

Primera edición

Estructuras de datos y algoritmos fundamentales

Víctor Manuel de la Cueva Hernández
Luis Humberto González Guerra
Edgar Gerardo Salinas Gurrión

Acerca de Editorial Digital



Estructura de datos y algoritmos fundamentales

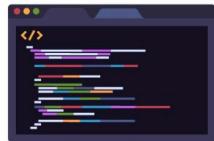
Víctor Manuel de la Cueva Hernández, Luis Humberto González Guerra
y Edgar Gerardo Salinas Gurrión

El objetivo principal del sello editorial del Tecnológico de Monterrey es divulgar el conocimiento y experiencia didáctica de la comunidad del Tecnológico de Monterrey a través del uso innovador de los recursos. Asimismo, apunta a contribuir con un modelo de publicación que integre en el formato de eBook, de manera creativa, las múltiples posibilidades que ofrecen las tecnologías digitales.

Con la Editorial Digital, el Tecnológico de Monterrey confirma su vocación emprendedora y su compromiso con la innovación educativa y tecnológica en beneficio del aprendizaje de los estudiantes dentro y fuera de la institución, así como del público en general.

D.R. © Instituto Tecnológico y de Estudios Superiores
de Monterrey, México 2020.

ebookstec@itesm.mx



Capítulo 1. Tipos de datos abstractos

01

Lograr un mayor nivel de abstracción ha sido uno de los puntos fundamentales en el desarrollo de los lenguajes de programación. La abstracción se puede dar en diferentes formas; dos de las principales son en los datos y en las estructuras de control. Las dos son sumamente importantes para la programación, pero la abstracción de datos siempre se ha dado en forma más natural, debido a que buscamos, constantemente, la mejor manera de organizarlos. Dentro de esta abstracción, una de las grandes aportaciones fue la idea de los tipos de datos abstractos, con el objetivo de poder tener acceso a ellos de forma eficiente.

1.1 Abstracción

La abstracción, en cualquier área, es un proceso que nos permite ocultar ciertos detalles para el usuario, dejando a la vista solo la información importante. Por ejemplo, en el lenguaje cotidiano, al hacer referencia a un animal llamado “perro”, estamos evitando describir cosas como su tamaño, color o raza; es decir, la palabra “perro” es una abstracción para el animal “perro”.

Podemos decir que la abstracción permite al usuario concentrarse en las cosas que le deben importar más y no tener distracciones en los detalles. Lo anterior da como resultado un mayor enfoque en lo que verdaderamente importa para entender o resolver un problema.

En una pieza de *software*, la abstracción permite concentrarse

más en qué es lo que hace dicha pieza, no tanto en entender cómo lo hace; es decir, es como una “caja negra” que esconde su implementación.

Sin lugar a dudas, la abstracción es el proceso que más ha influido en el desarrollo de los lenguajes de programación. Su objetivo principal, consiste en que el programador se concentre en el algoritmo para solucionar un problema, dejando en segundo plano la forma en la que se guardan los datos en la memoria, o la forma en la que se hace un proceso directamente en el procesador.

En lenguajes de programación, la abstracción se puede clasificar en dos tipos:

- **Abstracción de datos:** permite agrupar los datos en una forma más lógica, lo que la hace independiente de la forma en la que realmente se guardan en la memoria, ya sea principal o secundaria.
- **Abstracción de control:** permite ver los procesos ejecutándose por grupos y repeticiones de instrucciones, lo cual la hace independiente de la forma en la que se ejecutan las instrucciones en la unidad central de procesamiento, por medio de cálculo de direcciones y acceso a registros especiales.

Por razones naturales, la abstracción de datos es más entendible para los programadores, esto surge desde el primer momento en que se pensó hacer un programa.

La abstracción de datos permite definir:

- El dominio y la estructura que tienen los datos.
- El conjunto de atributos que caracterizan dichos datos.
- Las operaciones válidas que se pueden realizar con esos

datos.

1.2 Tipos de datos

En los lenguajes de programación existen muchos tipos de datos que se pueden manipular; por ejemplo: enteros, reales (de punto flotante), booleanos o *strings*. Para muchos de los lenguajes, sobre todo los compilados, es importante que el programador defina el tipo de datos que una determinada variable guardará, con el propósito de que pueda separar memoria suficiente para guardar el dato del tipo que se requiere; es decir, no se requiere la misma memoria para guardar un número entero que para guardar un número real.

La definición de los tipos de datos ya es en sí una abstracción. Al definir una variable entera, el sistema sabe que va a utilizar dos *bytes* para guardarla, en representación de complementos a dos, etc., pero esa información queda oculta para el usuario, por lo que este solo debe recordar que su variable debe contener un valor entero; esto, además de facilitar el uso de datos de un determinado tipo para el programador, hace que el programa sea independiente del sistema operativo o el *hardware* que se esté utilizando.

Al definir los tipos de datos, los programadores se dieron cuenta que algunos conceptos importantes estaban formados por diferentes tipos de datos. Por ejemplo, para definir el concepto de “alumno” y guardar su información, se requieren datos como: nombre, dirección, carrera, materias cursadas, calificaciones, etc. Estos datos, no son necesariamente del mismo tipo, ya que las características para describir un concepto son muy variadas, dependiendo en gran medida del concepto en sí, y de la tarea para la que dicho concepto se requiera. Así, para algunas tareas,

el concepto de “persona” deberá tener ciertas características y para otras se deberán agregar o quitar algunas, que no sean de tanta importancia. Otro ejemplo es: si estoy pensando en imprimir un directorio telefónico, el cual está formado por personas, se necesitará saber su nombre, dirección y teléfono; pero si estoy pensando en un archivo médico, tengo que agregar algunas características, como: peso, estatura o edad; y, posiblemente, puedo descartar el dato de la dirección.

Los diseñadores de lenguajes de programación, comenzaron a pensar de qué forma realizar una abstracción mayor (mayor que la de los tipos) para los datos. De esta forma nacieron los lenguajes que permitían al programador definir sus propios tipos basados en los tipos originales del lenguaje, llamados entonces tipos primitivos.

1.3 Tipos definidos por el usuario

Un mayor nivel de abstracción al de los tipos originales del sistema es el llamado “tipos primitivos”; si se tiene a los tipos primitivos como base, el usuario podía definir sus propios tipos, darles un nombre específico y declarar algunas variables que contuvieran datos de estos nuevos tipos. Un ejemplo de este tipo de abstracción lo son los registros (**récords**) del lenguaje de programación Pascal.

En la figura 1, inciso a, se puede observar un diagrama del concepto “persona” para un directorio telefónico, a su alrededor se muestran las características que lo describen. En la misma figura 1, pero en el inciso b, se observa una definición del tipo “persona”, con estas características, por medio de un **record**, dentro de la sección **type**.

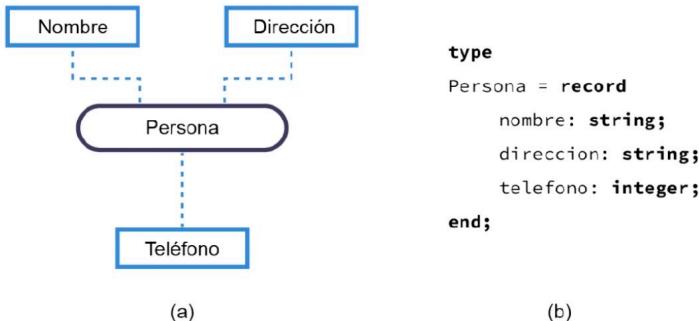


Figura 1.1 (a) El concepto “persona” en el centro con sus características que lo describen alrededor. (b) La implementación del concepto de “persona” en lenguaje de programación Pascal, definiendo el tipo Persona por medio de un registro (record).

Los registros en Pascal permitían al programador crear sus propios tipos; una vez creados, podían utilizarlos para definir variables de dichos tipos. Por ejemplo, utilizando la definición de la figura 1(b) se puede declarar una variable “Portero” que sea de este tipo “Persona” de la siguiente forma:

portero: Persona;

Al utilizar el operador “.” (operador punto) definido en Pascal, podemos referirnos al nombre del portero como:

portero.nombre

Donde, este nombre se puede tratar entonces como cualquier variable de tipo string, por ejemplo, si el nombre del portero es “Víctor”, podría hacer la asignación:

portero.nombre = “Víctor”;

Y si quiero escribir el nombre del portero lo puedo hacer con una instrucción writeln de la siguiente forma:

writeln(portero.nombre);

Las estructuras (**struct**) en el lenguaje de programación C jue-

gan exactamente el mismo papel que los registros en Pascal.

1.4 Tipos de datos abstractos (ADT)

El problema principal que se tiene con los datos definidos por el usuario, es que solo el programador sabe qué información está contenida en el tipo definido; es decir, es el único que sabe el tipo de dato primitivo al que pertenece cada característica definida. Esto implica que, por ejemplo, si un programador declaró la característica teléfono de un tipo persona como entero, se puede hacer la suma de dos teléfonos, cosa que no tiene mucho sentido; pero si lo declaró como **string**, puede entonces obtener el **substring** de los primeros 3 números del teléfono para encontrar la clave LADA (la clave LADA en México es la clave que identifica a un estado específico del país). Definitivamente, el más indicado para decidir qué puedo hacer con los datos del tipo que definió es el programador que hizo la definición.

En cuanto apareció la posibilidad de que el programador pudiera definir sus tipos, se muestra la necesidad de definir también las operaciones que se permiten hacer con las variables de dichos tipos; este nuevo nivel de abstracción es precisamente lo que se conoce como tipo de dato abstracto. Como es de esperarse, la representación de la información en la memoria está escondida para el usuario.

Un tipo de dato abstracto, como su nombre lo indica, es una abstracción de datos que encapsula en una entidad abstracta, tanto al tipo de datos que guarda la información requerida para su descripción, como a las operaciones que se puedan hacer sobre dichos datos; estas operaciones, normalmente, se especifican por medio de procedimientos o funciones definidas dentro

del tipo de dato abstracto. De esta forma, si el usuario de dicho tipo quiere definir una variable de ese tipo en su programa, se le proporciona el conjunto de operaciones que son las únicas que puede hacer sobre este tipo de variables.

Los tipos de datos abstractos (*Abstract Data Type*) fueron propuestos en 1974 por Barbara Liskov (premio Turing 2008) y Stephen N. Zilles, en el MIT, como parte del desarrollo del lenguaje de programación CLU. El lenguaje CLU es uno de los antecedentes más importantes para la creación de la Programación Orientada a Objetos (*Object Oriented Programming, OOP*), la cual tomó muchas de sus características, entre las que destacan, los ADT.

Cuando el programador crea un ADT se debe concentrar en especificar correctamente los datos que va a guardar, con todos los atributos que va a requerir. Una vez que se han definido correctamente los datos, el programador se puede concentrar en implementar las operaciones necesarias para manipular dichos datos; estas implementaciones se hacen por medio de funciones que regresan el resultado de la operación.

Por otro lado, cuando se utiliza un ADT, el usuario solo se tiene que concentrar en entender cuáles operaciones se pueden realizar sobre esos datos, pero se puede olvidar de la forma en la que esos datos están representados.

Con la aparición de los ADT, se hizo más fácil cumplir el sueño de la programación referente a la “reutilización del código”. El usuario que emplea un ADT creado por otro programador, solo se tienen que concentrar en conocer y entender las operaciones que se realizan sobre los datos, para lo cual bastará conocer tres cosas:

- el nombre de la operación,
- los datos que recibe como entrada y
- los datos que regresa después de hacer la operación.

En otras palabras, un ADT define una interfaz entre el usuario y el dato, por esta razón, al conjunto de operaciones de un ADT se le denomina comúnmente “Interfaz de programación de la aplicación” (API, *Application Programming Interface*), la cual podemos ver como un conjunto de nuevas instrucciones del lenguaje. API esconde por completo la implementación de dichas instrucciones, al utilizarlas, el programador cumple el objetivo de enfocar al programador en el problema.

1.5 Diseño de un ADT

La creación de un ADT tiene dos etapas: diseño e implementación. La implementación se hace por medio de un lenguaje de programación y es la que finalmente será distribuida y utilizada en el futuro; sin embargo, su diseño es totalmente independiente del lenguaje de programación en el que se va a implementar.

Para diseñar un ADT se deben tomar en cuenta tres factores:

- **Dominio:** conjunto de datos para los que se aplicará.
- **Atributos:** características que describen a cada uno de los datos. Se acostumbra definir el tipo de datos que es cada atributo.
- **Operaciones:** operaciones válidas que se pueden realizar con cada uno de los datos del dominio.

Como ejemplo, podemos seguir pensando en las personas en un

directorio telefónico. Para este caso el ADT se puede definir de la siguiente forma:

- Dominio: las personas que tengan una línea telefónica.
- Atributos de cada una de las personas del dominio:
 - Nombre: string
 - Dirección: string
 - Teléfono: entero (algunas personas lo ponen como string)
- Operaciones válidas para los datos del dominio:
 - Actualización de la dirección
 - Actualización del nombre
 - Actualización del teléfono
 - Eliminar persona
 - Insertar una nueva persona
- Otras ...

Se debe tener presente que el número de atributos y de operaciones depende en gran medida del diseñador y del problema que se esté resolviendo. Un ADT representa un concepto general, es decir, el concepto de “persona dueña de una línea telefónica”, cuando se utiliza para representar los datos de una persona en particular, se dice que esta persona es una instancia de dicho ADT.

Las personas que tienen experiencia en programación orientada a objetos, seguramente están pensando que esto se relaciona directamente con ella. La razón es que las clases en OOP cuentan con todos los elementos necesarios que se relacionan di-

rectamente para implementar un ADT, y por lo tanto, son la herramienta perfecta para implementarlos. Estos elementos, son los siguientes:

- ADT es equivalente a una clase.
- Atributos son equivalentes a las variables de instancia de la clase.
- Operaciones son equivalentes a los métodos de una clase.
- Instancia de un ADT es equivalente a un objeto de la clase.

Si las variables de instancia se declaran como privadas, estos datos solo se pueden manejar por medio de las operaciones definidas por medio de los métodos públicos. Los métodos controlan el acceso a los datos y establecen la forma en la cual estos datos pueden ser manejados y utilizados. La figura 2 muestra la clara relación que hay entre un ADT y una clase.

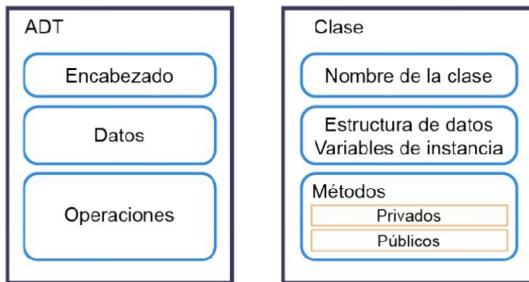


Figura 1.2. La definición de un ADT tiene una relación directa con los elementos de una clase en la OOP. La clase es la forma más simple de implementar un ADT

Una clase es la forma más simple de implementar un ADT, sin embargo, no es la única; a lo largo de la historia se ha utilizado el concepto de ADT y se han podido implementar sin que hubiera OOP, por ejemplo, por medio de paquetes en algunos lenguajes de programación como LISP o por medio de los módulos en len-

guajes como Haskell.

La principal ventaja que tiene la implementación de un ADT, por medio de una clase, es que verdaderamente los métodos y los datos quedan totalmente encapsulados en la estructura, lo que implica que ninguno de los métodos se puede usar para operaciones con datos para los que no fueron definidos.

1.5.1 Inicialización de un ADT

Generalmente, los ADT tienen una operación especial en su implementación, que se utiliza para realizar la inicialización de una instancia; a este método se le conoce como inicializador. En algunos lenguajes de la OOP, como Java y C++, el inicializador se llama constructor.

El constructor permite dar el espacio suficiente en la memoria para poder guardar todos los datos que definen el objeto que describen. En otras palabras, al momento de declarar un objeto, la operación del constructor lo crea y lo inicializa. Este proceso es totalmente transparente para el usuario, lo único que tiene que hacer es llamarlo, tal vez con algunos datos para colocarlos en algunos de los campos.

1.6 Ejemplo de diseño de un ADT

Uno de los ADTs más simples de hacer y más usados, es un conjunto de números a los que se les pueden realizar algunas operaciones. Veamos un ejemplo para el dominio de los números enteros no negativos, este se puede extender fácilmente al conjunto de los números enteros o al de los números reales.

El ADT que vamos a definir lo vamos a utilizar para guardar

y manipular un conjunto de números enteros no negativos. La estructura de datos física donde lo vamos a guardar será la más simple de todas ellas: un arreglo de enteros. Este arreglo, lo definiremos en su forma más simple, dándole una cantidad máxima de números que pueda contener.

Las operaciones que podemos hacer sobre los números en un arreglo son muy diversas y dependerán del uso que le demos al ADT. El ADT definido se podrá ir modificando, sobre todo en lo referente a las operaciones. Para los fines de este ejemplo, definiremos dos operaciones básicas: insertar un nuevo número en el arreglo y eliminar un número ya existente.

Si el arreglo se ve en forma vertical, se suele colocar el cero en la parte alta; si se ve en forma horizontal, se suele colocar el cero en la parte izquierda. Por eso, cuando se recorren los datos un lugar hacia el cero, es común decir “se recorren hacia arriba” o “se recorren hacia la izquierda”. El primer elemento de un arreglo siempre está en la posición 0. La definición del ADT sería:

Nombre: Arreglo

Dominio: Números enteros no negativos

Atributos:

- **MAX:** el número máximo de datos que caben en el arreglo.
- **datos:** el arreglo de enteros, inicia en el índice 0 y termina en el índice **MAX-1**.
- **tam:** el número de datos que tiene el arreglo, el cual inicia con 0.

Operaciones:

- **Insertar:** recibe el valor del número que se desea insertar, si

hay espacio disponible, se coloca al final del arreglo y se incrementa **tam**. Si ya no hay espacio disponible, se imprimirá un aviso “El arreglo está lleno, no se puede insertar el dato” y el número no se insertará.

- **Borrar:** si el arreglo no está vacío, borra el elemento del final del arreglo (el de la posición tam-1) y disminuye en 1 el tamaño **tam**, regresando el elemento que se borró. Si el arreglo está vacío, imprime el letrero “arreglo vacío” y regresa un -1.
- **Leer:** recibe un *string* que representa el nombre de un archivo que contiene los números al ser leídos. El primer renglón contiene el número de valores que contiene **tam**, los siguientes **tam** renglones contienen un valor por renglón. Si el archivo no existe, imprime el letrero “archivo inexistente”; si el archivo existe, lee todos los números contenidos en el archivo y los coloca en el arreglo del ADT, modificando **tam**.
- **Imprimir:** imprime todos los elementos del arreglo en forma horizontal, iniciando con el elemento en la posición 0.

Las dos operaciones que están definidas se pueden hacer de forma diferente; por ejemplo, se puede decidir insertar un elemento en la parte inicial del arreglo, esto es, siempre en la posición 0 o borrar el elemento que se encuentra en determinada posición. En capítulos posteriores, aumentaremos las operaciones sobre el arreglo y modificaremos las existentes.

Una vez que el ADT se tiene completamente definido se procede a colocarlo en código. En este libro usaremos el lenguaje C++.

1.6.1 Implementación del ADT como una clase

Iniciamos incluyendo las librerías necesarias (líneas 7 y 8) y

el nombre de espacio (línea 10) para definir la clase llamada **Arreglo** (línea 14). Además, colocamos sus atributos datos, un arreglo de tamaño **MAX**, y **tam**, un entero que se inicializa en 0 (líneas 16 y 17), como privados.

```
1. ///////////////////////////////////////////////////////////////////
2. // Implementación de un ADT Arreglo //
3. // Tecnológico de Monterrey      //
4. // Estructura de Datos        //
5. ///////////////////////////////////////////////////////////////////
6.
7. #include <iostream>
8. #include <fstream> // para el manejo de archivos
9.
10. using namespace std;
11.
12. #define MAX 50 // máximo tamaño del arreglo
13.
14. class Arreglo{
15. private:
16.     int datos[MAX];
17.     int tam = 0;
```

Ahora estamos listos para definir sus operaciones las cuales, al codificarlas, se colocan como métodos públicos de la clase.

1.6.2 *Implementación del ADT como un método*

Iniciaremos codificando la operación **Insertar** tal y como se definió, esto es, con un método llamado insertar, el cual recibe como parámetro el entero que se va a insertar en el arreglo, llamado **dato** (línea 3).

```

1. public:
2.     // recibe un dato entero y lo coloca al final del arreglo
3.     void insertar(int dato){
4.         if (tam < MAX){
5.             datos[tam] = dato;
6.             tam++; // se incrementa en 1 el tamaño del arreglo
7.         }
8.         else {
9.             cout << "El arreglo está lleno, no se puede insertar el dato" << endl;
10.        }
11.    }

```

Se verifica que haya espacio en el **Arreglo**, es decir, que **tam** sea menor que **MAX** (línea 4). Si hay espacio, se coloca el dato recibido como parámetro en siguiente posición vacía del arreglo, indicada por el número de datos que contiene, es decir, por **tam**. Se incrementa su tamaño **tam** en 1.

Si no hay espacio, se imprime el letrero “El arreglo está lleno, no se puede insertar el dato” y no se hace nada más.

1.6.3 *La operación borrar implementada como método*

La operación **Borrar** se implementa de acuerdo con su definición, como un método llamado **Borrar**, el cual no recibe parámetros y regresa un entero (línea 2).

```

1. // borra el elemento que se encuentra al final del arreglo
2. int borrar(){
3.     int dato;
4.     if (tam > 0){
5.         dato = datos[tam - 1];
6.         tam--;
7.     }
8.     else {
9.         cout << "El arreglo está vacío, no se puede borrar un elemento" << endl;
10.        dato = -1;
11.    }
12.    return dato;
13. }

```

Se define una variable entera llamada **dato** (línea 3), la cual va a contener el dato que se borró o -1. Si el arreglo no está vacío (línea 4), se guarda el dato a borrar, en este caso, el último dato que se encuentra en la posición **tam-1**, en la variable **dato** (línea 5), se disminuye el tamaño del arreglo en 1 unidad (línea 6). Si el arreglo está vacío (línea 8), se imprime el letrero “arreglo está vacío” (línea 9) y se coloca un -1 en la variable dato (línea 10). Finalmente, se regresa el dato guardado en la variable **dato**.

Es responsabilidad del usuario de este método verificar si el dato devuelto es un -1, lo cual significaría que no se pudo borrar ningún dato porque el arreglo estaba vacío o fue un entero no negativo; en cuyo caso lo puede usar para alguna actividad. El letrero no es necesario, solo se coloca para simplificar y hacer más claro este primer ejemplo.

Si se analiza el código, nos podemos dar cuenta que en realidad el último dato del arreglo no es borrado físicamente. Lo que en realidad se hace es un borrado “lógico”, es decir, al decrementar el tamaño del arreglo en uno, se deja disponible la casilla donde se encontraba el dato borrado, que en realidad todavía se encuentra, de tal forma que, en la siguiente inserción que se haga, el nuevo valor será escrito (sobrescrito) en dicho lugar.

1.6.4 Las operaciones auxiliares imprimir y leer

Las operaciones **Imprimir** y **Leer** se utilizan como funciones auxiliares para este ejemplo, con el propósito de facilitar la codificación para el programador.

```

1. // imprime el contenido del arreglo
2. void imprimir(){
3.     for (int i = 0; i < tam; i++){
4.         cout << datos[i] << " ";
5.     }
6.     cout << endl;
7. }
8.
9. // Lee un archivo de números enteros para colocarlos en el arreglo
10. // El archivo tiene el formato explicado en clase
11. void leer(string archivo){
12.     int n, numero;
13.     ifstream miArchivo(archivo);
14.     if (miArchivo.is_open())
15.     {
16.         miArchivo >> n;
17.         for (int i = 0; i < n; i++)
18.         {
19.             miArchivo >> numero;
20.             insertar(numero);
21.         }
22.         miArchivo.close();
23.     }
24.     else cout << "No se puede abrir el archivo";

```

Con el propósito de que el usuario no tenga que teclear dato por dato cuando se crea un arreglo, se creó el método **Leer**.

El método **Leer** recibe un **string** llamado archivo (línea 11), el cual contiene el nombre, con todo y extensión, del archivo del que se leerán los números que se colocarán en el arreglo. El archivo debe tener un formato específico, como se explicó en la definición del ADT, es decir, contiene un número por renglón. El primer número es el número de datos que tiene el arreglo (i.e. **tam**) y los siguientes **tam**, renglones, contienen los datos. El archivo de datos se relaciona con la variable de archivo llamada **miArchivo** (línea 13). Teniendo esto en consideración, el primer número leído, correspondiente al tamaño del arreglo, se guarda

en la variable local n (línea 16). Los “n” números en el archivo se leen entonces por medio de un ciclo for (línea 17 a 21), invocando en cada caso el método **Insertar** (línea 20), que coloca el número leído al final del arreglo.

No es necesario colocar el valor de n en tam, debido a que al invocar al método insertar se incrementa en uno el atributo (o propiedad) tam de la clase Arreglo; al final de la lectura se cierra el archivo (línea 22). Lo anterior se hace sólo si el archivo se puede abrir (línea 14); en caso contrario se escribe el letrero “no se puede abrir el archivo” (línea 24).

El método **Imprimir** no recibe parámetros ni regresa ningún valor (línea 2), tiene el objetivo de que el programador pueda ver lo que contiene el arreglo en todo momento. Cuando se invoca el método **Imprimir**, mediante un ciclo for (línea 3), se imprimen todos los números del arreglo en forma horizontal, separados por un espacio en blanco. Al terminar el ciclo, se imprime un salto de línea (línea 6).

1.6.5 La clase completa

La clase completa se puede ver a continuación:

```
1. ///////////////////////////////////////////////////////////////////
2. // Implementación de un ADT Arreglo //
3. // Tecnológico de Monterrey      //
4. // Estructura de Datos          //
5. ///////////////////////////////////////////////////////////////////
6.
7. #include <iostream>
8. #include <fstream> // para el manejo de archivos
9.
10. using namespace std;
11.
12. #define MAX 50 // máximo tamaño del arreglo
13.
14. class Arreglo{
15. private:
16.     int datos[MAX];
17.     int tam = 0;
18.
19. public:
20.     // recibe un dato entero y lo coloca al final del arreglo
21.     void insertar(int dato){
22.         if (tam < MAX){
23.             datos[tam] = dato;
24.             tam++; // se incrementa en 1 el tamaño del arreglo
25.         }
26.         else {
27.             cout << "El arreglo está lleno, no se puede insertar el dato" << endl;
28.         }
29.     }
30.
31.     // borra el elemento que se encuentra al final del arreglo
32.     int borrar(){
33.         int dato;
34.         if (tam > 0){
```

```
35.         dato = datos[tam - 1];
36.         tam--;
37.     }
38.     else {
39.         cout << "El arreglo está vacío, no se puede borrar un elemento" << endl;
40.         dato = -1;
41.     }
42.     return dato;
43. }
44.
45. // imprime el contenido del arreglo
46. void imprimir(){
47.     for (int i = 0; i < tam; i++){
48.         cout << datos[i] << " ";
49.     }
50.     cout << endl;
51. }
52.
53. // Lee un archivo de números enteros para colocarlos en el arreglo
54. // El archivo tiene el formato explicado en clase
55. void leer(string archivo){
56.     int n, numero;
57.     ifstream miArchivo(archivo);
58.     if (miArchivo.is_open())
59.     {
60.         miArchivo >> n;
61.         for (int i = 0; i < n; i++)
62.         {
63.             miArchivo >> numero;
64.             insertar(numero);
65.         }
66.         miArchivo.close();
67.     }
68.     else cout << "No se puede abrir el archivo";
69. }
70.
71.};
```

1.6.6 Uso del ADT implementado

El ADT queda completamente implementado en la clase **Arre-**

glo (atributos y métodos). Con el objetivo de manipular datos numéricos guardados en un arreglo, es posible utilizar el ADT por medio de sus métodos. Para explicar mejor el ejemplo siguiente, supongamos que la clase está guardada en el archivo llamado “Arreglo.h”.

La clase definida se puede usar de dos formas. La primera es abrir el archivo “Arreglo.h” y al final del mismo, después de la línea 71, colocar la función **main** y guardarla con extensión **cpp** para que se pueda compilar. La otra opción es abrir un archivo nuevo con extensión **cpp** (por facilidad, suponga que el archivo que contiene la clase y el nuevo, están colocados en el mismo folder). En este caso, la primera instrucción del nuevo archivo debe ser la que incluya la clase a ser utilizada, es decir, #include “Arreglo.h”. Ahora sí, proceda a escribir la función **main**, como se indica a continuación.

```

1. int main(){
2.     Arreglo miArreglo;
3.     int dato;
4.
5.     miArreglo.leer("numeros.txt");
6.     cout << "Los números leídos son: ";
7.     miArreglo.imprimir();
8.     dato = miArreglo.borrar();
9.     cout << "El elemento borrado es: " << dato << endl;
10.    cout << "El nuevo arreglo es: ";
11.    miArreglo.imprimir();
12. }
```

Para poder usar la clase **Arreglo**, la función **main** debe iniciar declarando un objeto de dicha clase, digamos **miArreglo** (línea 2). La variable entera **dato** se usa para guardar algún valor borrado en el **arreglo**.

Lo primero que tenemos que hacer es llenar el **arreglo** con datos. Para esto invocamos el método **Leer** del objeto **miArreglo** y le damos como parámetro, como se indicó en la definición y en

la implementación, el nombre del archivo que contiene los datos (línea 5), en este caso “numeros.txt” (sin acento porque es un nombre de archivo). Supongamos que el archivo contiene los siguientes datos:

4
0
1
2
3

El primer número indica que el archivo contiene 4 elementos, los elementos son los dígitos del 0 al 3, que se encuentran en los siguientes 4 renglones.

Una vez leídos los datos, invocamos el método **Imprimir** (línea 7, con un letrero en la línea 6 para saber lo que estamos imprimiendo) y obtenemos como salida:

0 1 2 3

Luego, invocamos el método **Borrar** que regresará el valor borrado o -1, el cual es guardado en la variable **dato** (línea 8). Recorriendo que se borra el último elemento del arreglo, el letrero en la línea 9, imprimirá lo siguiente:

El elemento borrado es: 3

Si en este momento volvemos a invocar al método **Imprimir** (línea 11, con un letrero en la línea 10 para saber lo que estamos imprimiendo), la salida obtenida será:

0 1 2

Es decir, se borró el último elemento del arreglo. De esta forma podemos seguir utilizando los métodos definidos por medio del objeto **miArreglo**.

Ejercicios



- 1.** Explique cómo implementaría un ADT en un lenguaje de programación que no es orientado a objetos, por ejemplo, con Pascal.
- 2.** Diseñe un ADT para establecer y manejar puntos en un plano cartesiano. Es decir, cada punto es un dato (objeto). Se deben definir los datos necesarios para guardar la información de un punto, así como las operaciones para manipular los puntos, dependiendo de la tarea que se quiere hacer con ellos.
- 3.** Defina un ADT para establecer y manejar rectángulos en un plano cartesiano. En este caso, cada rectángulo es un dato (objeto). Defina los atributos y las operaciones que requiere para manipularlos.
- 4.** Modifique la definición del ADT usado en el ejemplo del directorio telefónico para que pueda manejar varios teléfonos para una misma persona.
- 5.** Implemente el ADT del ejercicio 2 en su lenguaje orientado a objetos favorito.
- 6.** Implemente el ADT del ejercicio 3 en su lenguaje orientado a objeto favorito.
- 7.** Implemente el ADT del ejercicio 4.



Capítulo 2. Recursión

02

Cuando realizamos programas, es común que algunos métodos llamen a otros métodos para realizar su función, pero, cuando un método se llama a sí mismo de manera directa o indirecta se le conoce como recursión o recursividad.

En este capítulo estudiaremos este tipo de algoritmos, las implicaciones que tienen y cuándo es útil emplearlos.

2.1 Recursividad

Un algoritmo es recursivo cuando está definido en términos de una versión más simple de sí mismo. Para encontrar la solución final, es necesario encontrar primero la de problemas más pequeños de la misma naturaleza.

Frecuentemente, la recursión trabaja utilizando el principio divide y vencerás; es decir, divides el problema en varios problemas similares, pero de menor complejidad. Supongamos, por ejemplo, que deseas ordenar una lista; para hacer esta tarea primero ordenas sublistas más pequeñas, y al final ordenas la lista total. Aunque ordenar una lista de 1000 elementos y ordenar una lista de 2 elementos son de la misma naturaleza, el nivel de complejidad es muy diferente.

Otro ejemplo muy común para ilustrar la recursividad es calcular el factorial de un número. Sabemos que $5!$ es igual a $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, o sea igual a 120, pues el factorial de un número natural n , lo podemos calcular como

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1 \text{ para } n > 0.$$

También sabemos que matemáticamente definimos $0!$ Igual a 1. De esta misma manera podemos definir:

$$n! = \begin{cases} 1, & n = 0 \\ n * (n - 1)!, & n \geq 1 \end{cases}$$

Esto nos lleva a que podemos calcular $5!$ como $5 * 4!$, es decir $5 * 24$ y esto también es igual a 120.

2.1.1 Consideraciones para emplear la recursión

Cuando diseñemos o implementemos un algoritmo recursivo, es importante tener en cuenta las siguientes consideraciones para que el algoritmo nos resulte útil. Adicional al caso recursivo debe existir al menos un caso llamado base, el cual se resuelve de manera directa sin emplear recursión. Cada llamada recursiva debe progresar hacia el o los casos base.

Retomando el caso del factorial, podemos observar que el caso base es $n=0$, pues este por definición es 1. Para el resultado de factorial de 0 no necesitamos calcular ningún otro factorial previamente, sino que de manera directa podemos obtener el resultado 1. Por otro lado, para cualquier valor n mayor a 0 necesitamos conocer primero el valor de $(n-1)!$.

Podemos observar que cuando queremos calcular el factorial de un valor $n > 0$, la siguiente llamada de factorial se va aproximando al caso base pues la siguiente llamada a factorial, n se decremente en una unidad y así irá disminuyendo hasta que queramos calcular el factorial de 0, el cual es nuestro caso base y no necesitamos alguna llamada adicional.

La implementación de la función factorial podría quedar de la siguiente manera.

```

1. unsigned long factorial(int n){
2.     if (n==0)
3.         return 1;
4.     else
5.         return n*factorial(n-1);
6. }

```

A continuación, se muestra un diagrama que ilustra cómo ocurren las llamadas para calcular el factorial de 5.

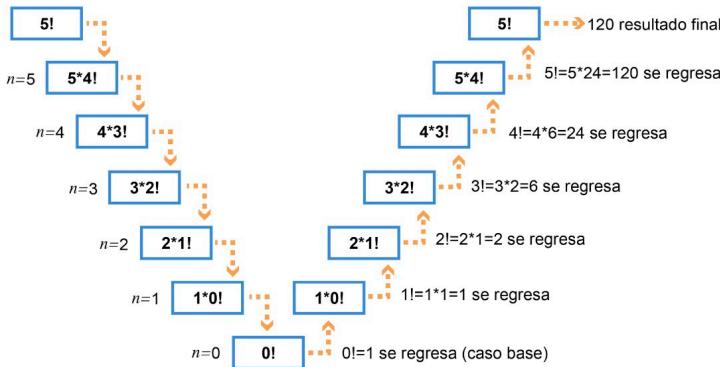


Figura 2.1 Representación de factorial(n).

Ahora cabe la pregunta, ¿qué sucede con la llamada de la función $\text{factorial}(5)$ cuando esta llama a la función $\text{factorial}(4)$? La llamada de $\text{factorial}(5)$ se queda en espera en una sección del programa llamada pila o *stack*, una vez que termine su ejecución, la llamada a $\text{factorial}(4)$ regrese su resultado, entonces $\text{factorial}(5)$ continúa en el punto donde se había detenido.

Un error común al implementar un algoritmo recursivo es no considerar las variables locales de cada llamada recursiva como independientes. Es decir, al modificar el valor de una variable de la función recursiva, no podemos pensar que también se ha modificado para las llamadas recursivas anteriores que están en espera, o que la llamada a la siguiente función recursiva, tendrá en sus variables los mismos valores que la llamada actual. Por ejemplo, en el método factorial, la variable n en el caso base toma el valor de 0 y regresa como resultado 1, es entonces que se re-

gresa el control a la ejecución de factorial(1); para esta llamada, su variable n sigue valiendo 1, independientemente de que en factorial(0) la variable n haya tomado el valor de 0.

2.1.2 La sucesión de Fibonacci

La sucesión de Fibonacci inicia de la siguiente manera:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Si observas un poco podrás darte cuenta que, a partir de 3er término de la serie, cada término se puede obtener como la suma de los 2 términos que le anteceden; mientras que los 2 primeros términos los definimos ambos como el 1; es decir, el 3er término es 2 debido a que los dos términos anteriores son el 1 y la suma de $1+1$ es 2; el cuarto término es el 3, debido a que los dos términos anteriores son el 1 y el 2 y, la suma de ambos, es 3.

Esta sucesión se puede generar muy fácilmente con recursión teniendo 2 casos base, el primer y segundo término de la sucesión son el 1. Los siguientes n-ésimos términos los calculamos como la suma del término n-1, más el término n-2.

La implementación de la función Fibonacci podría quedar de la siguiente manera: donde n representa el n-ésimo término de la serie que queremos calcular. Por cuestiones prácticas nuestro método toma como el primer elemento de la sucesión el elemento 0.

```

1. long fibonacci(int n){
2.     if (n==0 || n==1)
3.         return 1;
4.     else
5.         return fibonacci(n-1)+fibonacci(n-2);
6. }
```

Si vemos este programa, resalta la simpleza y lo natural que resulta este código. En estas líneas “podemos” calcular cualquier término de la sucesión sin necesidad de algún ciclo. Cuando po-

demos obtener códigos fáciles de interpretar y que su lectura resulta natural, solemos decir que es una solución elegante.

Si llamáramos a $\text{fibonacci}(4)$, nos daría como resultado 5; pero analicemos las llamadas que se realizan para obtener este resultado.

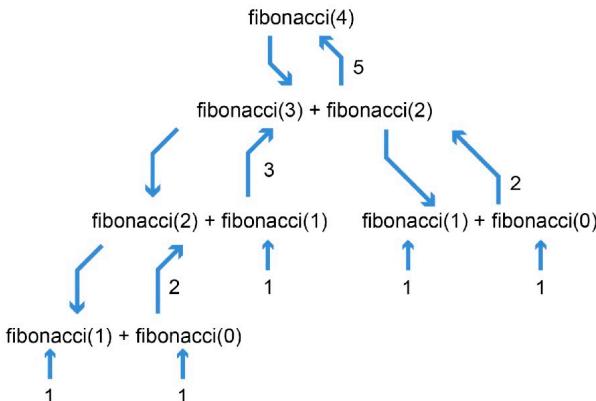


Figura 2.2 Obtención de un resultado con Fibonacci.

Podemos observar con este algoritmo que para calcular el 5to elemento de la sucesión, o sea $\text{Fibonacci}(4)$, se tienen que hacer 9 llamadas al método Fibonacci, resulta ser un número alto si consideramos el término que estamos calculando. Es fácil darse cuenta que para obtener el sexto elemento se necesitan 15 llamadas a la función Fibonacci, poco menos del doble del número de llamadas para calcular el término anterior. Como consecuencia, el número de operaciones a realizar también crece y, de igual manera, el tiempo que tarda el programa en encontrar la solución.

Si sabes utilizar lo básico en una hoja de cálculo, en un minuto o menos podrías encontrar los primeros 100 números de la serie de Fibonacci. Si lo hicieras a mano con ayuda de una calculadora que tuviera suficientes cifras significativas, podrías calcular estos mismos números en menos de una hora. Intenta

adivinar en cuánto tiempo una computadora promedio actual se tardaría utilizando el algoritmo que escribimos previamente. ¿Ya tienes un número en mente? Quizás para sorpresa tuya, una computadora actual tardaría alrededor de cientos o miles de millones de años. Suena increíble, ¿no?

De esta situación podemos mencionar varias cosas:

- Cuidado con lo bello, pues no siempre es lo mejor.
- Gracias al análisis de algoritmos podrás darte cuenta de estas situaciones y así desechar estos algoritmos.
- Esto no significa que la recursión no sirva, de hecho, este algoritmo recursivo con una pequeña modificación tardaría tan sólo millonésimas de segundo en cualquier computadora actual promedio.
- En la figura anterior puedes ver que, casi cualquier llamada al método Fibonacci, resulta en dos llamadas al mismo método y esto no es deseable.

2.1.3 Recursión infinita

Cuando hablamos de la recursión hicimos un fuerte énfasis en que deben existir al menos un caso base y que las llamadas recursivas deben ir progresando hacia él. En caso de tener un método recursivo que no cuente con un caso base o que las llamadas recursivas no se vayan aproximando hacia él, dará como resultado que nuestro programa tendrá un desbordamiento de pila, comúnmente conocido como un stack overflow, esto significa que el tamaño de la pila se ha excedido.

Esto sucede debido a que la llamada al método recursivo hace otra llamada a sí mismo, este hace otra llamada a sí mismo y así

sucesivamente hasta que la pila de nuestro programa ya no tiene capacidad de guardar una llamada más al método. El resultado en la mayoría de los lenguajes de programación termina con una salida anormal del programa y ello provoca que no se puedan recuperar los datos del mismo.

Por ejemplo, ¿qué sucedería si en el método factorial no pusieramos el caso base no recursivo de si n es igual a 0 entonces debe regresar 1? El método volvería a llamar a la función factorial, pero ahora pasando como parámetro el valor de n igual a -1, y así continuaría hasta terminar con el tamaño de la pila.

Si usamos el método factorial tal cual está escrito arriba con su caso base, pero lo mandamos a llamar para que calcule el factorial de -5, ¿qué va a suceder?, en este caso las llamadas recursivas no van a progresar hacia el caso base, sino que se van a alejar; por lo tanto, también tendríamos un caso de recursión infinita, terminada cuando ocurre el desbordamiento de la pila.

2.2 Iteración vs. recursión

A continuación hablaremos un poco acerca de los pros y contras de utilizar algoritmos iterativos contra algoritmos recursivos. Lo primero que debemos dejar en claro es posible que exista algún algoritmo que se pueda resolver de manera recursiva, pero no de manera iterativa; la respuesta es no. No existe algún algoritmo que se resuelva de manera recursiva pero sí de manera iterativa, esto significa que cualquier algoritmo recursivo también se puede escribir de manera iterativa.

Tenemos que tomar en cuenta que la recursión paga un costo adicional en cada llamada recursiva, al dejar un método en espera, la pila del programa tiene que realizar tareas adicionales que no se tienen en un método iterativo.

Es por esto que la recursión generalmente la debemos utilizar cuando la solución al problema es de naturaleza recursiva, de manera que si hay una solución simple de manera iterativa debemos de irnos hacia esa solución.

Por ejemplo, la suma de los números del 1 al n, es una tarea que fácilmente podemos realizar de las dos maneras; sin embargo, al haber una solución natural y sencilla de manera iterativa, la recomendación será implementar la solución iterativa, ya que una posible implementación recursiva dejaría en espera en la pila n número de llamadas al método. Entonces, al hacer la suma del 1 al 100,000 implicaría que en la pila tendrían que estar guardados 100,000 registros con información del contexto de cada llamada al método recursivo, si nuestro stack no es tan grande, el programa terminaría de manera anormal con la pérdida de datos del mismo.

Existen muchos programas de naturaleza recursiva y, aunque ya lo mencionamos previamente, que se pueden escribir de manera iterativa, la mayoría de las veces no resultan con una gran mejora, pues las estructuras adicionales que se necesitan utilizar compensan la sobrecarga de la recursión. Algunos de estos algoritmos los conoceremos más adelante; también gracias al análisis formal de algoritmos podremos darnos cuenta si vale la pena pasar un algoritmo a su otra forma.

Ejercicios



- Escribe una función **long potencia** (int base, int n), la cual recibe como parámetro dos enteros. El método debe calcular de manera recursiva un número elevado a otra potencia. El primer parámetro representa la base y el segundo parámetro la potencia. Puedes considerar que la potencia pasada como parámetro es un valor entero mayor o igual a cero.
 - Los coeficientes de un binomio elevado a una potencia son representados por el triángulo de pascal en honor al matemático Blaise Pascal. Las primeras 10 filas de este triángulo son:

El 1, que se encuentra hasta arriba, sería el coeficiente de elevar $(a+b)^0$, que evidentemente es resultado es 1, la siguiente fila son los coeficientes de elevar $(a+b)^1$ es decir $a+b$, la tercer fila son los coeficientes de elevar $(a+b)^2$, los valores 1, 2, 1, son los coeficientes del polinomio resultante $a^2+2ab+b^2$ y así sucesivamente.

Escribe la función **int trianguloPascal(int r, int c)**, la cual debe calcular el número del triángulo de Pascal que se ubica en la fila r y columna c. La implementación de este método debe ser recursivo. Por ejemplo, si se llama esta función con los valores 7 y 2, entonces la función regresa el valor 21, dado que tanto las filas como las columnas comienzan con el valor 0.

Para probar tu función escribe la función **void triangulares-HastaFila(int n)**, la cual recibe como parámetro hasta cuál fila se quieren imprimir los números del triángulo de pascal.

Esta función llama a la función **trianguloPascal** previamente implementada. Por ejemplo, si se llama esta función con el valor 9 entonces imprimirá:

Pues imprime desde la fila 0 hasta la fila 9 de los números del triángulo de Pascal.

```

1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5   10  10  5   1
1   6   15  20  15  6   1
1   7   21  35  35  21  7   1
1   8   28  56  70  56  28  8   1
1   9   36  84  126 126 84  36  9   1

```

- 3.** Escribe la función recursiva **bool palindromo(char str[], int i,int f)**, la cual recibe tres parámetros. El primero es un arreglo de caracteres que representan un string, el segundo y tercer parámetro representan los índices del arreglo que delimitan el string a verificar si se trata de un palíndromo o no. La primera vez que se llama esta función, el segundo parámetro es siem-

pre 0 y el tercer parámetro sería la longitud del arreglo menos uno. La función regresa a true cuando el arreglo pasado como parámetro sí representa un palíndromo y false en otro caso.



que delimita la notación **Θ grande**, y estamos diciendo que tenemos una cota asintóticamente ajustada con respecto al tiempo de ejecución.

3.1.2 Notación O grande (Big-O)

La notación **Θ grande** sirve para acotar de manera asintótica el crecimiento de un tiempo de ejecución dentro de un rango, pero normalmente solo se desea saber el acotamiento superior. Por ejemplo, la búsqueda secuencial es **$\Theta(n)$** , pero no es correcto decir que ese tiempo es para todos los casos, ya que se puede encontrar el valor buscado en la primera posición, entonces se ejecuta en **$\Theta(1)$** . El tiempo de ejecución para la búsqueda secuencial nunca es peor que **$\Theta(n)$** , pero pudiera ser mejor que ese tiempo. Usaremos la notación O grande para las cotas superiores asintóticas que son entradas muy grandes; se dice que el tiempo de ejecución es “ O grande de $f(n)$ ”, “ O de $f(n)$ ” o simplemente “Orden de $f(n)$ ”. Entonces podemos decir que la búsqueda secuencial es **$O(n)$** . La figura 3.3 muestra la notación **$O(n)$** .

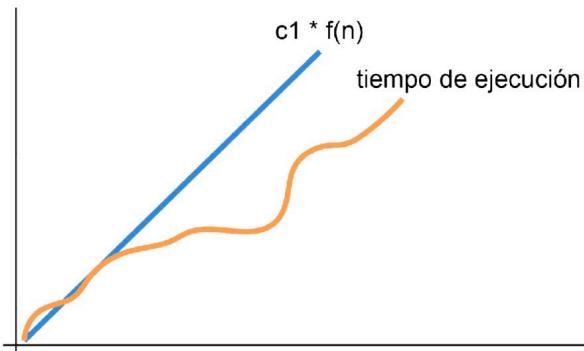


Figura 3.3 Notación $O(n)$.

Analizando la definición de la notación **Θ grande**, nos podemos dar cuenta que se parece a la notación O grande, excepto que la

notación **Θ grande** acota el tiempo de ejecución de los algoritmos por arriba y por abajo, en cambio la notación **O grande** solo acota por arriba.

Las principales notaciones O grande que se tienen en los algoritmos son:

- $O(1)$: Constante (que no depende de la entrada)
- $O(\log n)$: Logarítmica
- $O(n)$: Lineal
- $O(n \log n)$: Lineal * Logarítmica
- $O(n^2)$: Cuadrática
- $O(n^3)$: Cúbica
- $O(n^c)$: Polinomial (donde c es una constante)
- $O(m^n)$: Exponencial (donde m es una constante)
- $O(n!)$: Factorial

Esta notación es la que se estará utilizando durante todo el libro. En la figura 3.4 se puede apreciar el crecimiento de las notaciones **O grande (Big-O)** más básicas.

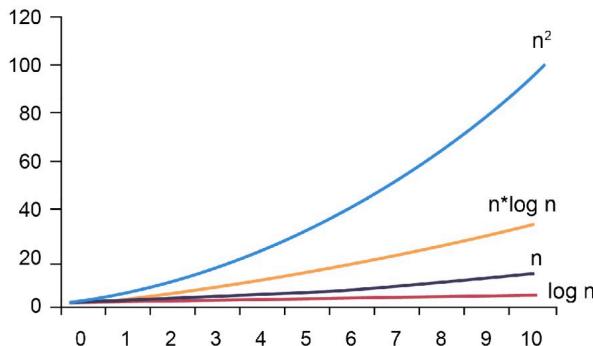


Figura 3.4 Crecimiento de las notaciones Big-O más básicas.

localiza o cuando el dato está en la última posición. Por cada entrada al ciclo, se realiza la comparación, la cual toma un tiempo constante, así como incrementar el valor y volver a comparar; si lo encuentra, se regresa el valor del índice. La suma de todas estas operaciones sería una constante llamémosla **a**; en el peor caso tendríamos que el ciclo toma $a * n$ y, como en caso de que no se encuentre tenemos que regresa -1; si asumimos que esto toma un valor constante que llamaremos **b**, nos quedaría que la función de búsqueda secuencial toma $a * n + b$. Para **n** muy grande, las constantes **a** y **b** se desprecian, por lo tanto, se dice que la búsqueda secuencial en su peor caso crece como el tamaño del arreglo, así, el tiempo de ejecución es $\Theta(n)$, y se dice que es “Theta grande de n” o simplemente “Theta de n”.

Si tenemos una función de tiempo de ejecución $f(n)$, cuando **n** es suficientemente grande, el tiempo de ejecución estará entre $c_1 * f(n)$ y $c_2 * f(n)$. Mientras existan constantes c_1 y c_2 que delimiten $f(n)$ para **n** muy grandes, decimos que el tiempo de ejecución del algoritmo es $\Theta(n)$. La figura 3.2 muestra esta notación.

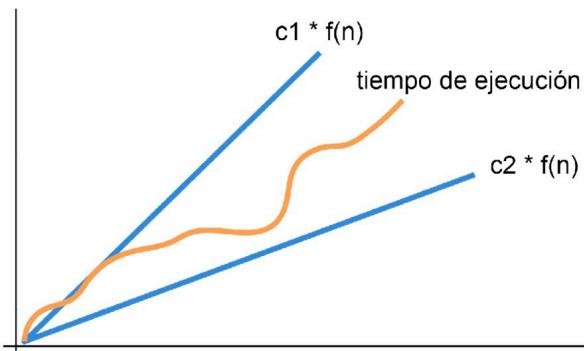


Figura 3.2 Notación $\Theta(n)$.

Se eliminan los factores constantes y términos de orden inferior, por lo cual podemos decir que el grado del polinomio es el

Capítulo 3. Análisis de algoritmos

03

Un algoritmo es un procedimiento que ayuda paso a paso a solucionar un problema. En programación, un algoritmo se puede expresar en pseudocódigo y este es la solución a un problema.

Para un problema, pueden existir varias soluciones, se debe analizar cada una de ellas y elegir la más eficiente. La eficiencia de los algoritmos se puede medir desde la perspectiva del tiempo de ejecución y del uso de memoria. En tiempo de ejecución, se habla en términos del tamaño de la entrada (llámémosle n), naturalmente se harán análisis para n muy grandes. Por otro lado, el uso de memoria requiere del algoritmo para ejecutarse; desde este punto de vista, se mide la cantidad extra de memoria que se necesita para poder realizar el algoritmo. En este capítulo, nos enfocaremos en la eficiencia del tiempo de ejecución y no tanto la eficiencia con el uso de memoria adicional, aunque si lo estaremos mencionando.

3.1 Notación asintótica

Dado un algoritmo que tiene como entrada n datos y cuyo tiempo de ejecución, en instrucciones máquina, se puede expresar con el polinomio $3n^2 + 87n + 230$, el término $87n + 230$ crece más lentamente que el término $3n^2$ conforme va aumentando el tamaño de los datos de entrada y, aunque para entradas chicas el término $3n^2$ tiene un peor comportamiento, siempre se tomarán en cuenta los escenarios de entradas grandes. En la figura 3.1 se puede apreciar el comporta-

miento de ambos términos.

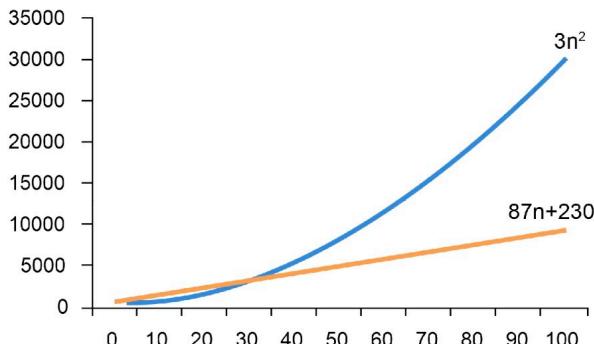


Figura 3.1 Crecimiento de los términos.

En la figura 3.1 se puede apreciar que el término que crece más rápidamente es el más significativo, al descartar los términos menos significativos y las constantes que contenga el término más significativo, estaremos utilizando la notación asintótica. Existen notaciones:

Notación Θ grande (Big-Θ)

Notación O grande (Big-O)

Notación Omega Ω grande (Big-Ω).

3.1.1 Notación Θ grande (Big-Θ)

Dado un algoritmo básico, la búsqueda secuencial de un dato sobre un arreglo de enteros de tamaño n , que regrese el índice donde se localiza el dato o -1 en caso de que no lo encuentre, el algoritmo sería:

```

1. int busqSecuencial(int arreglo[], int n, int dato){
2.     for (int i=0; i<n; i++) {
3.         if (arreglo[i] == dato){
4.             return i;
5.         }
6.     }
7.     return -1;
8. }
```

El ciclo **for** entra en su peor caso **n** veces, esto sería cuando no lo

3.1.3 Notación Ω grande (Big- Ω)

La notación **Ω grande** se usa cuando se desea decir que un algoritmo toma por lo menos cierta cantidad de tiempo, sin querer ofrecer la cota superior. Ya que la notación **Ω grande** es para mostrar los límites asintóticos inferiores. La figura 3.5 muestra el comportamiento de la notación **$\Omega(n)$** .

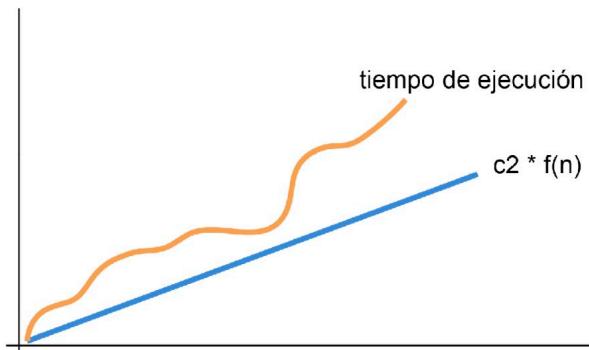


Figura 3.5 Notación $\Omega(n)$.

3.2 Análisis de algoritmos iterativos

La notación de **Big-O** de los algoritmos iterativos normalmente son polinomiales, para los cuales se tiene que hacer un análisis del algoritmo desde lo mas interno hasta lo mas externo; siguiendo tres reglas fundamentales: Secuencia, Condicional y Ciclos.

3.2.1 La secuencia

Cuando se tiene una secuencia de estatutos de códigos, se selecciona la que impacta con mayor fuerza a la secuencia de instrucciones, es decir a la que tenga un orden mayor. Por ejemplo: si tuviéramos 3 ciclos en forma secuencial, ciclo1 con **O(n)**, ciclo2 con **O(n · log n)** y ciclo3 con **O(log n)**, el orden de la secuencia es

$O(n \cdot \log n)$.

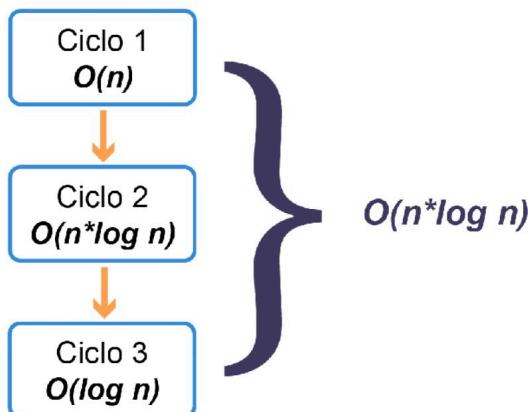


Figura 3.3 Representación de secuencia.

3.2.2 La condicional

Cuando existe la condicional (**if**), la acción a realizar se detona siempre y cuando la condición sea verdadera. Si la condición es falsa, siempre se tiene que escoger como el orden de la condicional la mayor de estas partes. Si solo existe la parte verdadera, se toma directamente como el orden de la condicional. Ejemplo: si se tuviera una condicional que tiene en la parte verdadera un Orden **$O(\log n)$** y en la parte falsa un Orden **$O(n^2)$** , el orden de la condicional será **$O(n^2)$** .

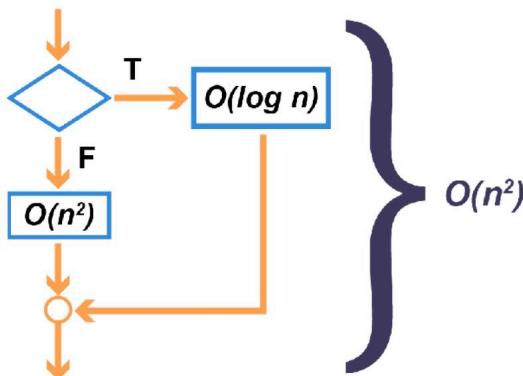


Figura 3.4 Representación de condicional.

3.2.3 Los ciclos

En los ciclos normalmente se tiene una condición y algo que hacer en repetidas ocasiones mientras la condición sea verdadera. El orden de los ciclos será la cantidad de veces que se realiza el ciclo por el orden de las operaciones internas que se encuentran dentro de él. Ejemplo: si se tuviera una ciclo que se realiza **n/2 veces** y las operaciones internas tienen un Orden **O(log n)**, el ciclo tendría un orden total de **O(n * log n)**.

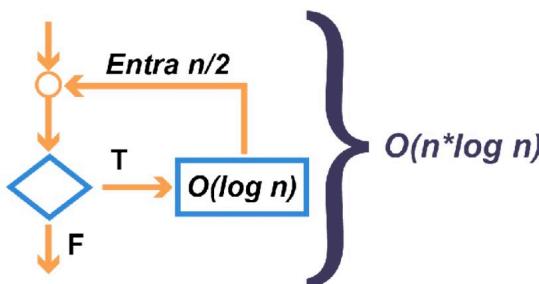


Figura 3.5 Representación de ciclo.

Cuando los ciclos tienen su rango de repetición en términos de **n**, entonces hay que revisar el comportamiento de la variable. La

variable de control y el orden normalmente será:

- **Lineal:** cuando a la variable de control se le suma o se le resta una constante.
- **Logarítmico:** cuando a la variable de control se le divide o se le multiplica una constante.

Al tener ciclos anidados se van multiplicando los órdenes, es muy recomendable analizar los órdenes de las instrucciones, desde las más internas hasta las más externas. Ejemplos:

```
1. for (int i=0; i<10; i++) {  
2.   instrucion;  
3. }
```

$O(1)$

```
1. for (int i=1234; i>0; i--) {  
2.   instrucion;  
3. }
```

$O(1)$

```
1. for (int i=0; i<n; i++) {  
2.   instrucion;  
3. }
```

$O(n)$

```
1. for (int i=n; i>0; i--) {  
2.   instrucion;  
3. }
```

$O(n)$

```
1. for (int i=0; i<n/2; i++) {  
2.   instrucion;  
3. }
```

$O(n)$

```
1. for (int i=n/5; i>0; i--) {  
2.   instrucion;  
3. }
```

$O(n)$

```
1. for (int i=1; i<=n; i+=4) {  
2.   instrucion;  
3. }
```

$O(n)$

```
1. for (int i=n; i>0; i-=5) {  
2.   instrucion;  
3. }
```

$O(n)$

```
1. for (int i=1; i<=n/2; i+=5) {  
2.   instrucion;  
3. }
```

$O(n)$

```
1. for (int i=n/7; i>0; i-=3) {  
2.   instrucion;  
3. }
```

$O(n)$

```

1. for (int i=1; i<=n; i*=2) {
2.   instrucion;
3. }

```

 $O(\log n)$

```

1. for (int i=n; i>0; i/=3) {
2.   instrucion;
3. }

```

 $O(\log n)$

```

1. if (n%2){
2.   for (int i=1; i<=n; i*=2){
3.     instrucion;
4.   }
5. } else{
6.   for (int i=4; i<100; i++){
7.     instrucion;
8.   }
9. }
10. }

```

 $O(\log n)$

```

1. for (int i=1; i<=n; i*=2){
2.   for (int j=4; j<n; j++){
3.     instrucion;
4.   }
5. }

```

 $O(n * \log n)$

```

1. if (n%2){
2.   for (int i=1; i<=n; i*=2){
3.     instrucion;
4.   }
5. } else{
6.   for (int i=4; i<n; i++){
7.     instrucion;
8.   }
9. }
10. }

```

 $O(n)$

```

1. for (int i=1; i<=n; i+=2){
2.   for (int j=1; j<n; j++){
3.     instrucion;
4.   }
5. }

```

 $O(n^2)$

3.3 Análisis de algoritmos recursivos

Para poder analizar la eficiencia de los algoritmos recursivos, se tiene que ver la cantidad de llamadas recursivas en ejecución que se realizan, así como el comportamiento del parámetro de control de la función recursiva. Normalmente se comporta de una de las siguientes formas:

- **$O(n)$:** cuando se tiene una sola llamada recursiva en ejecución y su parámetro de control se disminuye o incrementa en un valor constante.
- **$O(\log_b n)$:** cuando se tiene una sola llamada recursiva en ejecución y su parámetro de control se divide o se multiplica por un valor **b** constante.
- **$O(c^n)$:** cuando se tienen **c** llamadas recursivas en ejecución y su parámetro de control se incrementa o decrementa en una constante.
- **$O(n^{\log_b c})$:** cuando se tienen **c** llamadas recursivas en ejecu-

ción y su parámetro de control se divide o se multiplica por un valor **b** constante.

Ejemplos:

```
1. int sumaPares(int n){
2.     if (n <= 0)
3.         return 0;
4.     else if (n%2 == 0)
5.         return n+sumaPares(n-2);
6.     else
7.         return sumaPares(n-1);
8. }
```

$O(n)$

```
1. int fact(int n){
2.     if (n <= 0)
3.         return 1;
4.     else
5.         return n*fact(n-1);
6. }
```

$O(n)$

```
1. int contPot2(int n){
2.     if (n <= 0)
3.         return 0;
4.     else
5.         return 1+contPot2(n/2);
6. }
```

$O(\log_2 n)$

```
1. int contPot3(int n){
2.     if (n <= 0)
3.         return 0;
4.     else
5.         return 1+contPot3(n/3);
6. }
```

$O(\log_3 n)$

```
1. int algo(int n){
2.     if (n <= 0)
3.         return 400;
4.     else
5.         return algo(n-2)+algo(n-2)+algo(n-2)+algo(n-2);
6. }
```

$O(4^n)$

```
1. int algo(int n){
2.     if (n <= 0)
3.         return 123;
4.     else
5.         return algo(n-4)+algo(n-4)+algo(n-4);
6. }
```

$O(3^n)$

```
1. int algo(int n){
2.     if (n == 0)
3.         return 400;
4.     else
5.         return 1+algo(n/2)+algo(n/2)+algo(n/2)+algo(n/2);
6. }
```

$O(n^{\log_2 4}) = O(n^2)$

```
1. int algo(int n){
2.     if (n <= 0)
3.         return 123;
4.     else
5.         return 1+algo(n/4)+algo(n/4)+algo(n/4);
6. }
```

$O(n^{\log_4 3}) = O(n^{0.7924})$

3.3.1 Planteamiento de fórmulas de recurrencia

Para poder comprobar la eficiencia de los algoritmos recursivos, es necesario plantear el tiempo de ejecución en una fórmula de recurrencia en términos de $T(n)$, igualándola por un lado a la cantidad de llamadas recursivas en términos de la modificación de su parámetro de control. Por otro lado, el tiempo requerido del caso base, en función de la cantidad de operaciones básicas que llevaría su proceso.

Ejemplo 1

En la solución del factorial recursivo se tiene que, matemáticamente, el mínimo factorial que puede existir es el factorial de 0, que da como resultado 1; se sabe que el cálculo del factorial para cualquier entero positivo es la multiplicación de ese entero por la llamada recursiva de la función factorial en términos del entero original menos 1. Su planteamiento en fórmula recursiva sería por un lado su caso base, en donde solo toma 1 operación básica por el return 1. Por otro lado, tiene en su parte recursiva 1, la operación básica por el return; hay que agregarle el tiempo que toma la llamada a factorial para $n-1$. El algoritmo y la fórmula recursiva quedaría de la siguiente forma:

```
1. int factorial(int n){
2.     if (n == 0)
3.         return 1;
4.     return 1*factorial(n-1);
5. }
```

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + T(n - 1), & n > 0 \end{cases}$$

Ejemplo 2

Dado un algoritmo recursivo cuyo caso base tenga una operación básica que regrese un número cualquiera, y en su parte recursiva tenga 3 llamadas recursivas con un valor menor que su valor original de n . El algoritmo y la fórmula recursiva sería como sigue:

```

1. int a(int n){
2.     if (n == 0)
3.         return 123;
4.     return a(n-1)+a(n-1)+a(n-1);
5. }

```

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + 3T(n - 1), & n > 0 \end{cases}$$

Ejemplo 3

Otro ejemplo pudiera ser un algoritmo recursivo que, en su caso base, tenga una operación aritmética cualquiera, la cual regresaría y, en su parte recursiva, tendría 3 llamadas recursivas de la mitad de n. El algoritmo y la fórmula recursiva quedaría como se muestra:

```

1. int a(int n){
2.     if (n == 1)
3.         return 123*45;
4.     return a(n/2)+a(n/2)+a(n/2);
5. }

```

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

3.3.2 Solución de fórmulas de recurrencia

Cuando se tiene una fórmula recursiva o de recurrencia, lo que se desea es tenerla como una fórmula cerrada, sin recursividad. Para lograr esto, se tiene que realizar el proceso de estar iterando en su recursividad hasta lograr encontrar su patrón de comportamiento, y así llevarlo rápidamente al caso base, y obtener la fórmula cerrada.

Ejemplo 1

Retomando el ejemplo 1 del punto anterior, el algoritmo del factorial tiene una sola llamada recursiva con término del parámetro de control de n-1, por lo que debe de dar un O(n). La demostración se basa en la solución de su fórmula recursiva.

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + T(n - 1), & n > 0 \end{cases}$$

La base de fórmula recursiva será:

$$T(n) = T(n-1) + 1 \quad (3.1.1)$$

Entonces se tiene que buscar quién es $T(n-1)$, y se logra sustituyendo en la fórmula general (3.1.1) donde diga n , por $n-1$, se obtiene:

$$T(n-1) = T(n-2) + 1 \quad (3.1.2)$$

Ahora se sustituye la fórmula (3.1.2) en la fórmula base (3.1.1), queda:

$$T(n) = (T(n-2) + 1) + 1 \quad (3.1.3)$$

$$T(n) = T(n-2) + 2 \quad (3.1.4)$$

Ahora se buscará quien es $T(n-2)$, y se logra sustituyendo en la fórmula general (3.1.1) donde diga n por $n-2$ y se obtiene:

$$T(n-2) = T(n-3) + 1 \quad (3.1.5)$$

Ahora se sustituye la fórmula (3.1.5) en la fórmula general (3.1.1), queda:

$$T(n) = (T(n-3) + 1) + 2 \quad (3.1.6)$$

$$T(n) = T(n-3) + 3 \quad (3.1.7)$$

Se puede ver que si le rebaja 1 al término interno de T se le suma 1, si se le rebaja 2 se le suma 2 y si se le rebaja 3 se le suma 3, por lo tanto el patrón es que por cada constante que se le rebaje al valor interno de T , se le suma esa misma constante. Ahora bien, se tiene que llevar al caso base para que pare la recursividad, el caso base es $T(0)$, ahí ya no hay recursividad, y sabiendo que $T(0)$ es 1, por lo tanto queda:

$$T(n) = T(n-n) + n \quad (3.1.8)$$

$$T(n) = T(0) + n \quad (3.1.9)$$

$$T(n) = 1 + n \longrightarrow O(n)$$

Ejemplo 2

En el caso del ejemplo 2 del punto anterior, el algoritmo tiene 3 llamadas recursivas que rebajan la constante de 1 en cada una, por lo que debe de dar un $O(3n)$, la demostración se basa con la solución de su fórmula recursiva.

$$T(n) = \begin{cases} 1, & n = 0 \\ 1 + 3T(n - 1), & n > 0 \end{cases}$$

La base de fórmula recursiva será:

$$T(n) = 3T(n-1) + 1 \quad (3.2.1)$$

Entonces se tiene que buscar quién es $T(n-1)$, y se logra sustituyendo, en la fórmula general, (3.2.1) donde diga n , por $n-1$ y se obtiene:

$$T(n-1) = 3T(n-2) + 1 \quad (3.2.2)$$

Ahora se sustituye la fórmula (3.2.2) en la fórmula general (3.2.1), queda:

$$T(n) = 3(3T(n-2) + 1) + 1 \quad (3.2.3)$$

$$T(n) = 3^2T(n-2) + 3 + 1 \quad (3.2.4)$$

Ahora se buscará quién es $T(n-2)$, esto se logra en la fórmula general (3.2.1) donde diga n por $n-2$ y se obtiene:

$$T(n-2) = 3T(n-3) + 1 \quad (3.2.5)$$

Ahora se sustituye la fórmula (3.2.5) en la fórmula general (3.2.1), queda:

$$T(n) = 3^2(3T(n-3) + 1) + 3 + 1 \quad (3.2.6)$$

$$T(n) = 3^3T(n-3) + 3^2 + 3 + 1 \quad (3.2.7)$$

$$T(n) = 3^3T(n-3) + 3^2 + 3^1 + 3^0 \quad (3.2.8)$$

Se puede ver que por cada constante que se le rebaja al término interno de T se le suma 1 a la potencia de 3 que lo está multiplicando y se agrega el término 3 elevado a la constante -1 a la sumatoria que se está generando. Se tiene que llevar al caso base para que pare la recursividad, el caso base es $T(0)$ el cual tiene un valor de 1, y ahí ya no hay recursividad, por lo tanto queda:

$$T(n) = 3^nT(n-n) + 3^{n-1} + 3^{n-2} + \dots + 3^2 + 3^1 + 3^0 \quad (3.2.9)$$

Por lo que:

$$T(n) = \sum_{i=0}^n 3^i \quad (3.2.10)$$

y con el conocimiento de que:

$$\sum_{i=0}^n a^i = \frac{a^{n+1}-1}{a-1} \quad (3.2.11)$$

por lo tanto queda:

The diagram shows a mathematical derivation. On the left, a blue-bordered box contains the equation $T(n) = \frac{3^{n+1}-1}{3-1}$. An orange arrow points from this box to another blue-bordered box on the right, which contains the expression $O(3^n)$.

Ejemplo 3

Para el ejemplo 3 del punto anterior, el algoritmo tiene 3 llamadas recursivas de la mitad de la n , cada una de sus llamadas, por lo que debe de dar un $O(n \log 2^3)$, la demostración se basa con la solución de su fórmula recursiva.

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

La base de fórmula recursiva será:

$$T(n) = 3T(n/2) + 1 \quad (3.3.1)$$

Se recomienda que, cuando se tenga para la llamada recursiva el valor de la n dividido entre una constante, se sustituya la n por la constante elevada a la k , esto debido a que debe llegar al caso base en una determinada cantidad (k) de llamadas recursivas, por lo tanto:

$$\frac{n}{2^k} = 1 \text{ y despejando } n = 2^k \quad (3.3.2)$$

Por lo tanto sustituyendo n por 2^k en la fórmula (3.3.1) y sabiendo que $2^k/2 = 2^{k-1}$, queda:

$$T(2^k) = 3T(2^{k-1}) + 1 \quad (3.3.3)$$

Entonces se tiene que buscar quién es $T(2^{k-1})$, y se logra sustituyendo en la fórmula general (3.3.3) donde diga 2^k , por 2^{k-1} y se obtiene:

$$T(2^{k-1}) = 3T(2^{k-2}) + 1 \quad (3.3.4)$$

Ahora se sustituye en la fórmula general (3.3.3), queda:

$$T(2^k) = 3(3T(2^{k-2}) + 1) + 1 \quad (3.3.5)$$

$$T(n) = 3^2T(2^{k-2}) + 3 + 1 \quad (3.3.6)$$

Ahora se buscará quién es $T(3^{k-2})$, y se logra sustituyendo en la fórmula general (3.3.3) donde diga 2^k por 2^{k-2} y se obtiene:

$$T(2^{k-2}) = 3T(2^{k-3}) + 1 \quad (3.3.7)$$

Ahora se sustituye en la fórmula general (3.3.3), queda:

$$T(2^k) = 3^2(3T(2^{k-3}) + 1) + 3 + 1 \quad (3.3.8)$$

$$T(2^k) = 3^3T(2^{k-3}) + 3^2 + 3 + 1 \quad (3.3.9)$$

$$T(2^k) = 3^3T(2^{k-3}) + 3^2 + 3^1 + 3^0 \quad (3.3.10)$$

Se puede ver que por cada constante que se le rebaja al término interno de T se le suma 1 a la potencia de 3 que lo esta multiplicando, y se agrega el término 3 elevado a la constante -1 a la sumatoria que se esta generando. Ahora bien, se tiene que llevar al caso base para que pare la recursividad, el caso base es $T(1)$, ahí ya no hay recursividad, además sabiendo que $2^0 = 1$, queda:

$$T(2^k) = 3^kT(2^{k-k}) + 3^{k-1} + 3^{k-2} + \dots + 3^2 + 3^1 + 3^0 \quad (3.3.11)$$

entonces:

$$T(2^k) = \sum_{i=0}^k 3^i \quad (3.3.12)$$

se sabe que:

$$\sum_{i=0}^n a^i = \frac{a^{n+1}-1}{a-1} \quad (3.3.13)$$

por lo que queda:

$$T(2^k) = \frac{3^{k+1}-1}{3-1} = \frac{3*3^k-1}{2} \quad (3.3.14)$$

Al despejar la fórmula de $n = 2k$ (3.3.2) y la k queda que:

$$k = \log_2 n \quad (3.3.15)$$

Al sustituir la k queda:

$$T(n) = \frac{3*3^{\log_2 n}-1}{2} \quad (3.3.16)$$

por la propiedad de los logaritmos se sabe que:

$$x^{\log_b y} = y^{\log_b x} \quad (3.3.17)$$

por lo tanto queda:

$$T(n) = \frac{3n^{\log_2 3} - 1}{2} \rightarrow O(n^{\log_2 3}) = O(n^{1.5849})$$

3.4 Clasificación de problemas

La teoría de la clasificación de problemas está basada en qué tan difícil es resolver el problema y básicamente puede ser:

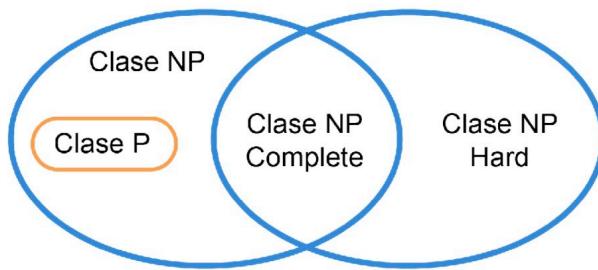


Figura 3.6 Clasificación de problemas.

- **Problemas de clase NP:** un problema que puede ser resuelto por un tiempo No-determinístico.

Un algoritmo No-Determinístico es un algoritmo que consta de dos fases: suponer y comprobar.

Si la complejidad temporal en la etapa de comprobación de un algoritmo no-determinístico es polinomial, entonces este algoritmo se denomina algoritmo polinomial no-determinístico.

- **Problemas de Clase P:** un problema es asignado a la clase P

(tiempo polinomial) cuando el tiempo de esta solución está en una potencia constante del tamaño de la entrada (n^k)

- **Problemas de Clase NP-Complete:** un problema B es NP-Complete si cumple con las siguientes dos características: ser un problema de Clase NP y para otro problema A en NP, A se reduce polinomialmente en B.
- **Problemas de Clase NP-Hard:** un problema se dice que es NP-Hard si solo cumple con la característica de que, para otro problema A en NP, A se reduce polinomialmente en B.

Reducción polinomial es una manera de relacionar dos problemas, que cuando se tiene un algoritmo **A** que se resuelve en tiempo polinomial, hay que ver la forma de que la salida de este pueda ser la entrada del algoritmo **B**, la idea es que la solución del algoritmo **A** apoye a la solución del algoritmo **B**.

Ejercicios



1. Encuentra la notación asintótica (**Big O**) para cada uno de los siguientes algoritmos, asumiendo un **O(1)** para *instrucción*:

- ```
for (int i=2; i<n; i++){
 instrucción;
}
```
- ```
int x=n;
while (x > 0){
    instrucción;
    x--;
}
```
- ```
for (int i=1; i<=n; i+=2){
 instrucción;
}
```
- ```
int x=n;
while (x > 0){
    instrucción;
    x-=3;
}
```

- ```
for (int i=2; i<n; i*=2){
 instrucción;
}
```
- ```
int x=n;  
while (x > 0){  
    instrucción;  
    x/=3;  
}
```
- ```
for (int i=1; i<=n; i+=2){
 for (int j=n; j>0; j-=2){
 instrucción;
 }
}
```
- ```
int x=n;  
while (x > 0){  
    for (int y=n; y>=0; y/=2){  
        instrucción;  
    }  
    x-=3;  
}
```
- ```
for (int i=1; i<=n; i*=2){
 for (int j=n; j>0; j-=2){
 instrucción;
 }
}
```

- ```
for (int i=1; i<=n; i*=2){  
    for (int j=n; j>0; j/=2){  
        instrucción;  
    }  
}
```
- ```
int algo(int n){
 if (n<=0)
 return 123;
 else
 return 1+algo(n-2);
}
```
- ```
int algo(int n){  
    if (n<=0)  
        return 123;  
    else  
        return 1+algo(n/2);  
}
```
- ```
int algo(int n){
 if (n<=0)
 return 123;
 else
 return 1+algo(n-2)+algo(n-2)+algo(n-2)+algo(n-2);
}
```

- ```
int algo(int n){  
    if (n<=0)  
        return 123;  
    else  
        return 1+algo(n/2)+algo(n/2)+algo(n/2)+algo(n/2);  
}
```

2. Encuentra la fórmula recursiva que representa el tiempo de ejecución de los siguientes algoritmos recursivos:

- ```
int algo(int n){
 if (n<=0)
 return 123;
 else
 return 1+algo(n-2)+algo(n-2)+algo(n-2)+algo(n-2);
}
```
- ```
int algo(int n){  
    if (n<=0)  
        return 123;  
    else  
        return 1+algo(n-5)+algo(n-5);  
}
```
- ```
int algo(int n){
 if (n<=0)
 return 123;
 else
 return 1+algo(n/2)+algo(n/2)+algo(n/2)+algo(n/2);
}
```

```
• int algo(int n){
 if (n<=0)
 return 123;
 else
 return 1+algo(n/5)+algo(n/5)+algo(n/5)
}
```

3. Encuentra la fórmula cerrada de las siguientes fórmulas recursivas:

- $T(n) = \begin{cases} 1, & n = 0 \\ 1 + 5T(n - 1), & n > 0 \end{cases}$
- $T(n) = \begin{cases} 1, & n = 0 \\ 1 + 8T(n - 1), & n > 0 \end{cases}$
- $T(n) = \begin{cases} 1, & n = 0 \\ 1 + 4T(n/2), & n > 0 \end{cases}$
- $T(n) = \begin{cases} 1, & n = 0 \\ 1 + 5T(n/3), & n > 0 \end{cases}$
- $T(n) = \begin{cases} 1, & n = 0 \\ n + T(n/2), & n > 0 \end{cases}$



# Capítulo 4. Algoritmos de búsqueda

04

**L**a búsqueda de información quizás sea una de las tareas más comunes que realizamos de manera cotidiana. Imagina cuántos productos pasan diario por las cajas de cobro en una tienda de autoservicio o cuántas búsquedas se realizan en sitio de compras por internet. Cuando hablamos de búsquedas de un valor en un listado, se habla principalmente de dos aproximaciones:

- búsqueda secuencial,
- búsqueda binaria.

En este capítulo abordaremos estas dos técnicas; conoceremos sus características, así como sus ventajas y desventajas.

## 4.1 Búsqueda secuencial

**I**magina que quieres abrir una puerta que fue cerrada con seguro, encuentras una caja llena de llaves y solo una abre la puerta. Todas las llaves son del mismo tipo, solo cambian las muescas y no hay manera visual de relacionar la cerradura con la llave maestra; solo te queda ir probando llave por llave hasta encontrar la que abre la puerta.

Cuando hacemos esto, estamos realizando una búsqueda secuencial, tomas una primera llave y pruebas si es la que buscas, si no es así, descartas esa llave y continuas con otra. De la misma manera podría suceder si tienes un folder lleno de recibos, si no están ordenados, no podrás localizar el recibo que estás buscando.

En programación, esta técnica es comúnmente empleada para buscar un valor dentro de un arreglo, cuando lo encontramos, frecuentemente se regresa a la posición del arreglo en el que se encuentra el valor buscado. Lo que hacemos es ver si el valor buscado se encuentra en la posición 0, en caso de no estarlo, continuamos con la siguiente posición y así hasta que hallemos el valor buscado

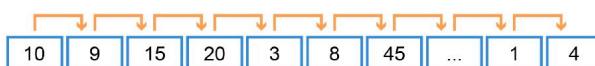


Figura 4.1 Representación visual de una búsqueda secuencial

Otra situación que puede suceder es que el valor que buscamos no se encuentre dentro de los datos en donde intentamos encontrarlo. En el caso de la búsqueda en el arreglo, nos podemos dar cuenta, cuando hemos llegado al final de los valores, que contiene el mismo y no hemos conseguido hallar el valor. En esta situación debemos regresar un valor que permita distinguir que el valor no fue hallado y comúnmente se regresa -1. Esto debido a que los índices del arreglo nunca son negativos pues comienzan en 0 y van incrementando de uno en uno, por lo tanto, regresar el valor -1, es una clara señal de que el valor no se ha encontrado.

Ya con esto en mente procedamos a ver una posible implementación de búsqueda secuencial.

```

9. int busqSecuencial(int arreglo[], int n, int dato){
10. for (int i=0; i<n; i++) {
11. if (arreglo[i] == dato){
12. return i;
13. }
14. }
15. return -1;
16. }
```

Esta implementación recibe 3 parámetros: el primero es el arreglo en donde vamos a realizar la búsqueda, el segundo es el número de valores que contiene el arreglo, es decir, aunque el arreglo tenga cierto tamaño puede ser que el arreglo no este

“ lleno” y solo contenga un número menor de elementos, por lo tanto, limitaremos la búsqueda hasta **n**. Finalmente el tercer parámetro es el valor que estamos buscando.

Además, podemos observar que existen dos posibles salidas para la función, la primera es dentro del cuerpo del *if*; cuando se encuentra el valor deseado se regresa a la posición en la que se localiza y la segunda es cuando se han terminado de recorrer todos los valores del arreglo y no se ha encontrado el valor, entonces, el control del programa sale del ciclo y regresa el valor -1.

## 4.2 Búsqueda binaria

**E**n los ejemplos de búsqueda anteriores recordarás que los datos no estaban ordenados de alguna manera, sin embargo, muchas veces los datos ya se encuentran ordenados o los podemos ordenar. Pensemos, por ejemplo, que te han llegado varios recibos telefónicos y estos están ordenados por número de teléfono y necesitas encontrar el recibo de un número particular; podrías buscar recibo por recibo, pero también se puede aprovechar que los recibos ya están ordenados y sacar ventaja de esta situación.

Algo que podrías hacer es ir a un recibo que se encuentre más o menos al centro del bonche y comparar el teléfono de ese recibo con el que estás buscando; si tuviste la fortuna de acertar en el recibo que buscabas pues listo, en caso contrario, si el número telefónico buscado es menor, entonces solo buscarás en la primera mitad del bonche de recibos; si es mayor, buscarás en la segunda mitad y repites este paso con lo que te quedó, hasta encontrar el recibo que buscas.

Si te das cuenta, en este proceso, si no encontraste el recibo a la primera oportunidad, en la siguiente búsqueda habrás descar-

tado la mitad de recibos en los cuales buscar. En el caso de tener 1000 recibos descartaste aproximadamente 500, si hubieras utilizado la búsqueda secuencial, solo habrías descartado 1.

En la siguiente iteración, en caso de no encontrar el recibo acabarías descartando lo que te quedaba y ya solo tienes que buscar en una cuarta parte de los recibos que hay, mientras que con búsqueda secuencial solo habrías descartado 2.

La técnica de búsqueda binaria está catalogada como del tipo reduce y vencerás, lo que significa reducir nuestro problema en uno más pequeño y simple que el original. Esta búsqueda aprovecha la ventaja de tener los datos ordenados y eso nos permite ir descartando la mitad de los elementos sobre los que se hace la búsqueda hasta que se localice el elemento de interés.

El algoritmo es sencillo y su implementación es muy común, tanto en su forma iterativa como recursiva. Su implementación iterativa regularmente utiliza tres variables auxiliares que nos ayudan a limitar la búsqueda: **min**, **max** y **avg**.

La variable **min** nos indica desde qué posición del arreglo vamos a buscar, **max** nos indica hasta qué parte del arreglo vamos a buscar, y **avg** indica el punto medio del rango en donde vamos a ver si ahí se localiza el elemento que busca. Si el elemento en la posición **avg** concuerda con el elemento buscado, entonces termina la búsqueda, si no, veremos si el valor buscado es menor al localizado en la posición **avg** y en ese caso **max** se iguala a **avg - 1**. En caso contrario **min** se iguala a **avg + 1** y se vuelve a buscar en el nuevo rango delimitado por **min** y **max**.

Mostremos un ejemplo de cómo se comportaría la búsqueda binaria en el siguiente arreglo, si se buscara el valor 56 dentro de él.

|   |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 |
| 8 | 10 | 13 | 18 | 26 | 29 | 32 | 38 | 56 | 61 | 77 | 80 | 87 | 92 |

| min | max | avg | arreglo (avg) | encontrado | valor es |
|-----|-----|-----|---------------|------------|----------|
| 0   | 13  | 6   | 32            | false      | mayor    |
| 7   | 13  | 10  | 77            | false      | menor    |
| 7   | 9   | 8   | 56            | true       |          |

Figura 4.2 Representación de búsqueda binaria en un arreglo.

Una implementación iterativa de búsqueda binaria sería la siguiente:

```

1. int busqSecuencial(int arreglo[], int min, int max, int dato){
2. int avg;
3. while (min < max) {
4. int avg = (min + max) / 2;
5. if (dato == arreglo[avg]){
6. return avg;
7. }else if (dato < arreglo[avg]){
8. max = avg - 1;
9. }else{
10. min = avg + 1;
11. }
12. }
13. return -1;
14. }
```

### 4.3 Eficiencia de búsqueda secuencial y búsqueda binaria

**C**omo ya vimos en el capítulo anterior, la búsqueda secuencial tiene una complejidad de tiempo de ejecución **O(n)**, es decir, en el peor de los casos necesitaremos **n** iteraciones para encontrar el valor buscado o para darnos cuenta de que el valor no se localiza en el arreglo.

Ahora analicemos la complejidad del algoritmo de búsqueda binaria. Lo que nos debe interesar es el número de iteraciones del ciclo *while*, cada iteración del ciclo se puede realizar en tiempo constante, en el peor de los casos, se va a ejecutar hasta que **min** sea igual a **max**; es decir, cuando el rango de búsqueda se haya re-

ducido a 1.

También observamos que el rango de búsqueda se reduce por mitad en cada iteración. Para simplificar la explicación tomemos un rango de búsqueda de tamaño **N** que sea una potencia de 2, por lo tanto, **N** lo podemos expresar como  $N=2^m$ , en la siguiente iteración el rango de búsqueda se reduce por mitad, entonces ahora tenemos un rango de búsqueda de tamaño  $N/2=2^{m-1}$ , en la siguiente iteración será  $N/4=2^{m-2}$  y así hasta llegar a un rango de tamaño 1, el cual es igual a  $2^0$ . Como  $m$  se ha ido reduciendo en una unidad cada iteración entonces se han realizado  $m+1$  iteraciones. Para expresar  $m+1$  en función de **N** multiplicamos  $N=2^m$  por dos y obtenemos  $2N=2^{m+1}$ ; después aplicamos logaritmo base 2 a cada término de la ecuación, entonces obtenemos que  $\log(2N)=m+1$ ; por lo tanto su complejidad temporal de ejecución es  $O(\log_2(n))$ , ya que como vimos anteriormente el factor 2 se puede despreciar.



# Capítulo 5. Algoritmos de ordenamiento

05

Es muy común para los usuarios que, al contar con un conjunto de datos, necesiten que estos cumplan con ciertas características para poder manipularlos de forma eficiente. Una de las características más deseables es que puedan estar guardados en la estructura de datos siguiendo un orden específico, es decir, que los datos estén ordenados.

En este capítulo estudiaremos los procedimientos más importantes que se han desarrollado para ordenar datos. Aunque los datos se pueden mantener ordenados en varios tipos de estructuras de datos, analizaremos los algoritmos para ordenar datos en un arreglo.

Los datos más simples de ordenar son aquellos que siguen una relación de orden; tal es el caso de los números, que tienen una relación natural. Sin embargo, se pueden crear relaciones de orden para cualquier tipo de datos; por ejemplo, las palabras se pueden ordenar alfabéticamente o los objetos, se pueden ordenar de acuerdo al dato contenido en alguno de sus atributos. Para facilitar las explicaciones, en la mayoría de los ejemplos se utilizarán números enteros, pero los algoritmos funcionan de igual forma para cualquier dato que se pueda ordenar.

## 5.1 Algoritmos de ordenamiento

En un ADT **Arreglo** (ver capítulo 1), que contiene números enteros desordenados, se puede establecer una operación como parte de sus métodos para que logre colocar los números en algún orden específico. Este método se puede progra-

mar siguiendo un procedimiento conocido como algoritmo de ordenamiento.

La idea fundamental de un algoritmo de ordenamiento es que logre cumplir su objetivo: ordenar los números, en la forma más eficiente posible. Esta definición de eficiencia debe ser establecida en base a un criterio, por ejemplo, tiempo de ejecución o números de operaciones básicas, memoria requerida o ambas (ver capítulo 3).

Un algoritmo de ordenamiento, en su forma más simple, se puede definir como un procedimiento que recibe, como entrada, un arreglo que contiene datos desordenados y regresa, como salida, un arreglo que contiene los mismos números colocados en un orden específico, ascendente o descendente. Para ahorrar memoria, el arreglo de salida puede ser el mismo que el que se recibió como entrada.

A lo largo de la historia, los científicos de la computación han inventado algoritmos que logran ordenar un arreglo de datos de forma cada vez más eficiente, sin embargo, hasta la fecha, no existe el método de ordenamiento óptimo, es decir, no existe un método de ordenamiento que se tarde lo menos posible con cualquier tipo de arreglo al que le demos entrada. Esto ha ocasionado que todos los algoritmos creados se mantengan vigentes y que se tengan que estudiar varios tipos, al menos los más importantes de cada clase para poder aplicar el algoritmo adecuado a una tarea específica con ciertas características.

Existen varios algoritmos de ordenamiento que trabajan sobre varios tipos de estructuras de datos. Estudiaremos una clase particular de ellos, que son los algoritmos de ordenamiento que trabajan sobre un arreglo y se basan en la comparación y en el intercambio de datos de una posición del arreglo a otra.

En cada caso de estudio se establecerá su orden Big-O (ver capítulo 3), tomando como operación básica la comparación.

## 5.2 El intercambio en un arreglo

**E**l tipo de algoritmo que vamos a estudiar es el que funciona ordenando datos en un arreglo y está basado en el intercambio de datos de una posición del arreglo a otra. Este intercambio se realizará como resultado de una comparación de los dos datos.

El intercambio entre dos posiciones de un arreglo es tan común que algunos ADTs que manejan arreglos lo tienen como una de sus operaciones (método en una clase, comúnmente llamado **swap**, como en C++), la cual recibe dos índices del arreglo, que indican las dos posiciones cuyos datos se desean intercambiar, y realizan el intercambio.

El procedimiento de intercambio entre dos posiciones de un arreglo requiere de una variable auxiliar, generalmente llamada **paso**, que nos ayuda para realizar el proceso.

Suponiendo que tenemos un arreglo **A**, una variable auxiliar **paso** y se desea intercambiar los datos en las posiciones **i** y **j**, el algoritmo de intercambio sería el siguiente:

- 1. paso**  $\leftarrow A[i]$ , pasar el dato en la posición **i** del arreglo **A**, a la variable auxiliar
- 2. A[i]  $\leftarrow A[j]$** , pasar el dato en la posición **j** del arreglo **A**, a la posición **i** del mismo arreglo **A**.
- 3. A[j]  $\leftarrow \text{paso}$** , pasar el valor guardado en la variable auxiliar (que era el que originalmente estaba en la posición **i** del arre-

glo **A**), a la posición **j** del arreglo **A**.

Gráficamente se vería como en la figura 5.1, en donde se muestra el arreglo inicial. Las siguientes 3 muestran los tres pasos del algoritmo descrito anteriormente.

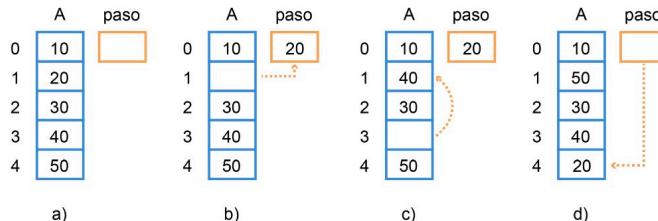


Figura 5.1 Proceso de intercambio de dos posiciones de un arreglo.

El proceso de intercambio se puede implementar como un método en el ADT **Arreglo** (ver capítulo 1) de la siguiente forma:

```

1. void intercambiar(int i, int j){
2. int paso;
3. paso = datos[i];
4. datos[i] = datos[j];
5. datos[j] = paso;
6. }
```

Si en la función **main**, declaramos un objeto **miArreglo**, de la clase **Arreglo** y este arreglo contuviera, al menos 8 números, podríamos cambiar el elemento en la posición 2 con el de la posición 7 de la siguiente forma:

```

1. int main(){
2. Arreglo miArreglo;
3.
4. miArreglo.intercambiar(2, 7);
5. }
```

Desde luego que haría falta leer los elementos del arreglo, hacer el intercambio y luego imprimirla, para poder observar el cambio realizado. Veamos ahora los principales algoritmos de ordenamiento basados en intercambio.

## 5.3 Selection Sort

**E**l algoritmo de selección (**Selection Sort**, en inglés) es el algoritmo de ordenamiento más simple de implementar y el más natural de imaginar. Si se le pide a los alumnos, que no saben nada de algoritmos de ordenamiento, que diseñen un algoritmo para ordenar un arreglo, es muy probable que terminen con algo similar a **Selection Sort**; sin embargo, es un algoritmo muy ineficiente.

Imaginemos un conjunto de números guardados en un arreglo y supongamos que lo queremos ordenar en forma ascendente: el más pequeño en la posición 0 y el más grande al final. La idea detrás del algoritmo es muy sencilla, se trata de recorrer todo el arreglo en busca de la posición que ocupa el elemento más pequeño, es decir, se recorre para seleccionar el elemento más pequeño, de ahí su nombre. Al encontrarlo, se intercambia el elemento en la posición 0 con el de la posición encontrada, la que contiene el más pequeño.

Al final de este proceso, en la posición 0 se encuentra el elemento más pequeño; repetimos el proceso, pero ahora desde la posición 1, y así sucesivamente hasta llegar a la penúltima posición. Solo se llega hasta la penúltima y no a la última debido a que al llegar a la penúltima, la última ya se encuentra en su posición correcta de forma automática. Si se quiere ordenar en forma ascendente basta con seleccionar el más grande en cada pasada. Un ejemplo dejará el proceso mucho más claro.

### *Ejemplo de Selection Sort*

Supongamos que tenemos el siguiente arreglo de 10 números (ahora con una representación horizontal con la posición 0 a la izquierda) y se quiere ordenar en forma ascendente:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
| 45 | 23 | 12 | 96 | 84 | 72 | 10 | 34 | 63 | 25 |

Vamos a usar una variable **menor** que guardará la posición del elemento más pequeño en cada pasada. También vamos a requerir un índice **i**, que indique en dónde se colocará el elemento más pequeño encontrado. Usamos un índice **j** que nos ayude a recorrer el resto del arreglo, desde **i+1** hasta la última posición (**tam-1**, donde **tam** es el número de elementos en el arreglo). La colocación inicial sería la siguiente:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 45 | 23 | 12 | 96 | 84 | 72 | 10 | 34 | 63 | 25 |
| i  | j  |    |    |    |    |    |    |    |    |

$$i=0, j=i+1=1, \text{menor}=0$$

Iniciamos el proceso con la iteración 1, preguntando si el elemento en la posición **j** (23) es menor que el de la posición menor (45). En este caso,  $23 < 45$ , y tenemos que colocar la variable **menor** a 1, indicando que la posición 1 contiene el menor, hasta ahora. Incrementamos **j**, para seguir buscando el menor:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 45 | 23 | 12 | 96 | 84 | 72 | 10 | 34 | 63 | 25 |
| i  |    | j  |    |    |    |    |    |    |    |

$$i=0, j=2, \text{menor}=1$$

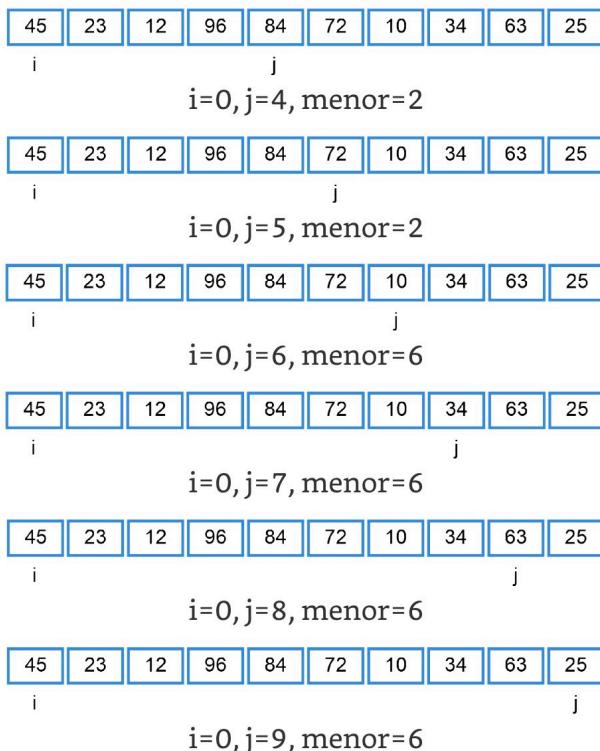
Volvemos a preguntar si el elemento en la posición **j** (12) es menor que el elemento en la posición **menor** (23). Como la respuesta es verdadera porque  $12 < 23$ , se coloca la variable **menor** a 2. Se incrementa la **j**:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 45 | 23 | 12 | 96 | 84 | 72 | 10 | 34 | 63 | 25 |
| i  |    | j  |    |    |    |    |    |    |    |

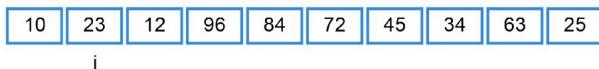
$$i=0, j=3, \text{menor}=2$$

Preguntamos ahora si el elemento en la posición **j** (96) es menor que el de la posición **menor** (12). Como la respuesta es falsa porque  $96 > 12$ , el valor en **menor** no se modifica; lo que

indica que el menor, hasta el momento, sigue estando en la posición 2. Se incrementa la **j** para continuar la búsqueda del menor. Los pasos restantes de la primera pasada (iteración 1) se muestran a continuación:



En ese momento se sabe que el menor de todos los elementos, desde la posición 0 hasta la posición 9, está en la posición 6, la cual se encuentra en la variable **menor**. Procedemos a intercambiar el elemento en la posición **i** (45) con el elemento en la posición menor (10), quedando el arreglo:



Hasta aquí es el fin de la primera iteración (pasada). En este

momento podemos garantizar que el menor de todos los elementos se encuentra en la posición 0.

Para iniciar la iteración 2, se incrementa la **i** en 1, ahora queda con un valor de 1, y se procede a recorrer el arreglo desde la posición 2 hasta la 9. Para esto, colocamos la **j** a **i+1**, es decir a 2, y se repite el proceso. La iteración 2 es la siguiente:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 23 | 12 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

**i****j**

$$i=1, j=i+1=2, \text{menor}=i=1$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 23 | 12 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

**i****j**

$$i=1, j=3, \text{menor}=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 23 | 12 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

**i****j**

$$i=1, j=4, \text{menor}=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 23 | 12 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

**i****j**

$$i=1, j=5, \text{menor}=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 23 | 12 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

**i****j**

$$i=1, j=6, \text{menor}=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 23 | 12 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

**i****j**

$$i=1, j=7, \text{menor}=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 23 | 12 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

**i****j**

$$i=1, j=8, \text{menor}=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 23 | 12 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

**i****j**

$$i=1, j=9, \text{menor}=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
| i  |    |    |    |    |    |    |    |    |    |

Al final de la iteración 2 se garantiza que los elementos antes de la posición 2 ya están ordenados. Continuemos con la iteración 3:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
| i  | j  |    |    |    |    |    |    |    |    |

$$i=2, j=i+1=3, \text{menor}=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
| i  |    | j  |    |    |    |    |    |    |    |

$$i=2, j=4, \text{menor}=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
| i  |    |    | j  |    |    |    |    |    |    |

$$i=2, j=5, \text{menor}=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
| i  |    |    |    | j  |    |    |    |    |    |

$$i=2, j=6, \text{menor}=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
| i  |    |    |    |    | j  |    |    |    |    |

$$i=2, j=7, \text{menor}=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
| i  |    |    |    |    |    | j  |    |    |    |

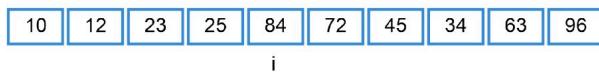
$$i=2, j=8, \text{menor}=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 96 | 84 | 72 | 45 | 34 | 63 | 25 |
| i  |    |    |    |    |    |    | j  |    |    |

i

El último arreglo es el que queda al final de la iteración 3. El 23 era el menor de los números que quedaban y ya estaba en su posición correcta, pero el algoritmo no se da cuenta de esto y hace el intercambio del elemento en la posición 2 con el elemento en la posición 2, quedando igual.

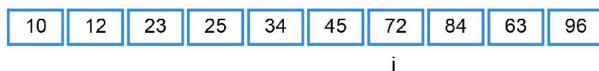
Los arreglos al final de las iteraciones restantes son:



final de iteración 4



final de iteración 5



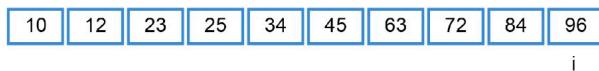
final de iteración 6



final de iteración 7



final de iteración 8



final de iteración 9

Se puede observar que al final de la iteración 9, el arreglo completo ya está ordenado. El último elemento queda ordenado en forma automática después de ordenar los anteriores, por lo que no es necesaria una iteración 10.

### 5.3.1 Algoritmo e implementación de Selection Sort

El algoritmo de **Selection Sort** se puede describir como se muestra a continuación.

|                      |               |                                                                                                        |
|----------------------|---------------|--------------------------------------------------------------------------------------------------------|
| <b>selectionSort</b> | Descripción   | Algoritmo de ordenamiento de arreglos                                                                  |
|                      | Entrada       | Arreglo <b>datos</b> a ser ordenado, de tamaño <b>tam</b>                                              |
|                      | Salida        | Arreglo ordenado                                                                                       |
|                      | Precondición  | Un arreglo base 0 (la primera posición es 0) válido como parte de los atributos del ADT <b>Arreglo</b> |
|                      | Postcondición | El arreglo ordenado es el mismo que recibió                                                            |

- La variable **menor** contiene el índice del menor elemento.
1. Para **i=0** hasta **tam-2** {
  2.     **menor = i**
  3.     Para **j = i + 1** hasta **tam-1**
  4.         **If datos[i] < datos[menor]**
  5.             **menor = j**
  6.     Intercambiar el elemento en la posición **i** con el de la posición **menor**
  7. }

Al tener el algoritmo, su implementación en cualquier lenguaje de programación es directa. Por ejemplo, en C++ la podemos implementar como una operación (método de una clase) del ADT **Arreglo**:

```

1. // Ordena el arreglo usando Selection Sort
2. void selectionSort(){
3. int menor, paso;
4. for (int i=0; i <= tam-2; i++){
5. menor = i;
6. for (int j=i+1; j <= tam-1; j++)
7. if (datos[j] < datos[menor])
8. menor = j; // posición del menor
9. // Hacer el intercambio
10. paso = datos[i];
11. datos[i] = datos[menor];
12. datos[menor] = paso;
13. }
14. }
```

Se puede utilizar el método **intercambiar** que escribimos en la sección 2.2, quedando entonces de la siguiente forma:

```

1. // Ordena el arreglo usando Selection Sort
2. void selectionSort(){
3. int menor, paso;
4. for (int i=0; i <= tam-2; i++){
5. menor = i;
6. for (int j=i+1; j <= tam-1; j++)
7. if (datos[j] < datos[menor])
8. menor = j; // posición del menor
9. // Hacer el intercambio
10. intercambiar(i, minimo);
11. }
12. }
```

La implementación es muy simple, solo requiere dos ciclos **for**. El primero, (línea 4) con un índice **i**, de 0 a **tam-2** (penúltima posición), para ir recorriendo el punto desde donde se inicia la selección en cada pasada. El segundo (línea 6), con un índice **j**, que va del valor de **i+1** hasta **tam-1** (la posición final) en busca del número menor.

En cada iteración se puede usar el método **Imprimir** para ver cómo va quedando el arreglo. Cuando se desee utilizar este método, se debe declarar un objeto de la clase **Arreglo**, digamos **miArreglo**, llenar el arreglo con los datos a ser ordenados y luego invocar el método **selectionSort**, simplemente con el operador punto, es decir, **miArreglo.selectionSort()**.

### *5.3.2 Complejidad del algoritmo Selection Sort*

La operación básica de este algoritmo de ordenamiento es la comparación. Su complejidad (orden) la daremos como una función del número de comparaciones que realiza en el peor de los casos.

Supongamos que el arreglo tiene **n** elementos a ser ordenados. El primer **for** es realizado **n-1** (desde 0 hasta **n-2**). El segundo **for**, que es el que contiene la comparación, se hace **n-1** veces (está dentro del primer **for**) y en cada una de ella hacer el siguiente número de comparaciones:

- **n-1** comparaciones la primera vez (el for va de 1 a **n-1**)
- **n-2** comparaciones la segunda vez (el for va de 2 a **n-1**)
- ...
- 1 comparación la **n-1** éSIMA vez (el for va de **n-2** a **n-1**)

Eso significa que el número de comparaciones en el peor caso

está dado por la suma:

$$\text{Comparaciones} = (n - 1) + (n - 2) + \cdots + 2 + 1$$

Y es lo mismo si lo escribimos al revés:

$$\text{Comparaciones} = 1 + 2 + \cdots + (n - 2) + (n - 1)$$

Por otro lado, la suma de Gauss nos dice que:

$$\sum_{k=1}^n k = \frac{n(n+1)}{2}$$

Entonces, la suma de las comparaciones es una suma de Gauss solo que el límite superior es **n-1**. Sustituyendo la **n-1** en el lugar de la **n**, podemos ver que el número de comparaciones es:

$$\text{Comparaciones} = \frac{(n-1)(n-1+1)}{2} = \frac{(n-1)n}{2} = \frac{n^2 - n}{2}$$

Por lo que el orden del algoritmo es:

$$O\left(\frac{n^2 - n}{2}\right)$$

Y aplicando las reglas del orden vistas en el capítulo 3 tenemos que:

$$O\left(\frac{n^2 - n}{2}\right) = O(n^2 - n) = O(n^2)$$

es decir, el orden del algoritmo **Selection Sort** es **O(n<sup>2</sup>)**.

## 5.4 Insertion Sort

El algoritmo de inserción (**Insertion Sort**, en inglés) es el siguiente algoritmo de ordenamiento más natural y simple de implementar, es tan ineficiente como **Selection Sort**.

Funciona de la siguiente manera: se recorren todos los elemen-

tos del arreglo desde la posición 0 hasta la última posición. En cada paso, el elemento de la posición actual se inserta (de ahí su nombre) en el lugar que le corresponde en un arreglo ordenado, el cual será conformado por todos los elementos del arreglo que estén antes que el actual; por lo que en cada paso, los elementos, desde la posición 0 hasta la actual, ya están ordenados, aunque se pueden mover al insertar los que faltan.

Para poder insertar el elemento actual en el lugar que le corresponde, será necesario irlo comparando con cada uno de los elementos que están arriba. Si el actual es menor que el que tiene arriba, se intercambian, si no es menor, tampoco será menor que el resto de los elementos que están antes, debido a que el arreglo está ordenado, es decir, si no es menor, el elemento actual está en su lugar correcto dentro del arreglo ordenado que está antes que él. Este proceso continúa con todos los elementos del arreglo.

Aunque arriba se comentó que el proceso debe iniciar en el primer elemento, es decir, en el que se encuentra en la posición 0, en realidad, no es necesario porque antes de él no hay ninguno, lo que significa que ya está colocado en su lugar adecuado. El proceso iniciará con el elemento en la posición 1, considerando que el de la posición 0 ya está ordenado. Vemos un ejemplo completo.

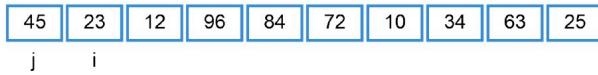
### *Ejemplo de Insertion Sort*

Supongamos que tenemos el siguiente arreglo de 10 números, ahora con una representación horizontal con la posición 0 a la izquierda, y se quiere ordenar en forma ascendente:

| 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
|----|----|----|----|----|----|----|----|----|----|
| 45 | 23 | 12 | 96 | 84 | 72 | 10 | 34 | 63 | 25 |

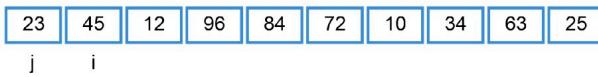
Vamos a requerir un índice **i**, que indique el elemento que estamos insertando en cada paso (por lo que va de **1** a **tam-1**, donde

**tam** es el número de elementos que contiene el arreglo), y otra variable índice **j**, que nos ayude a recorrer el arreglo ya ordenado, el de la parte de la izquierda de la posición actual, hasta llegar a la posición **0** o encontrar la posición del elemento actual (por lo que va desde **i-1** hasta **0** o hasta que encuentre su posición).



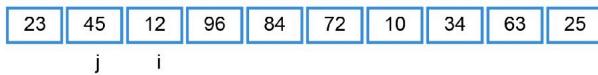
$$i=1, j=i-1=0$$

Iniciamos el proceso preguntando si el elemento en la posición **j+1** (23) es menor que el de la posición **j** (45). En este caso,  $23 < 45$ , y los intercambiamos, quedando el arreglo de la posición 0 a la 1, completamente ordenado:



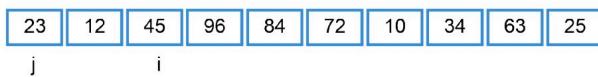
$$i=1, j=i-1=0$$

Como el índice **j** ya llegó a la posición 0, esta iteración termina. Entonces, incrementamos el índice **i** y colocamos **j** a **i-1**, para iniciar la siguiente iteración que colocará al elemento de la posición 2 en su lugar correcto, dentro del arreglo formado por los elementos anteriores a él:



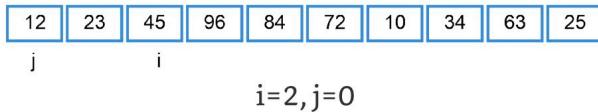
$$i=2, j=1$$

Al iniciar la segunda iteración, comparamos el elemento en la posición **j+1** (12) con el elemento en la posición **j** (45). Como  $12 < 45$ , los intercambiamos y como la **j** no ha llegado a **0**, la decrementamos:

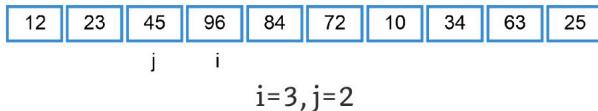


$$i=2, j=0$$

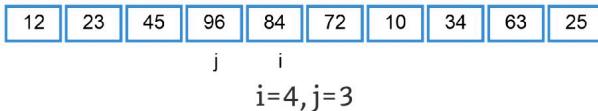
Comparamos otra vez el elemento en la posición **j+1** con el de la posición **j** y vemos que  $12 < 23$ , por lo que los intercambiamos:



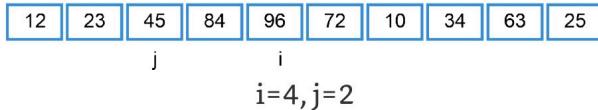
Como **j** ya llegó a la posición 0, indica que es el fin de la iteración. Incrementaremos **i** en uno y colocamos **j** a **i-1** para iniciar la tercera iteración:



En el inicio de la iteración 3, comparamos el 96 con el 45 y vemos que  $45 < 96$ , por lo que no se hace intercambio y eso indica el fin de la iteración. Debido a que sabemos que todos los elementos anteriores al 45 son menores al 45 y por lo tanto, menores al 96, lo que significa que el 96 está en el lugar correcto. Incrementaremos la **i** y colocamos la **j** a **i-1**.



En el inicio de la iteración 4, comparamos y vemos que  $84 < 96$ , y los intercambiamos. Decrementamos la **j**:



Comparamos y vemos que el  $45 < 84$ , lo que significa que el 84 ya está en su lugar correcto y termina la iteración. Pasamos a la iteración 5 y mostramos todos los pasos:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 84 | 96 | 72 | 10 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

j i

 $i=5, j=4$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 84 | 72 | 96 | 10 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

j i

 $i=5, j=3$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 72 | 84 | 96 | 10 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

j i

 $i=5, j=2$ 

Pasamos a la iteración 6:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 72 | 84 | 96 | 10 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

j i

 $i=6, j=5$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 72 | 84 | 10 | 96 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

j i

 $i=6, j=4$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 72 | 10 | 84 | 96 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

j i

 $i=6, j=3$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 10 | 72 | 84 | 96 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

j i

 $i=6, j=2$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 10 | 45 | 72 | 84 | 96 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

j i

 $i=6, j=1$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 10 | 23 | 45 | 72 | 84 | 96 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

j i

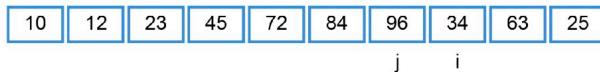
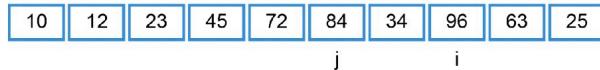
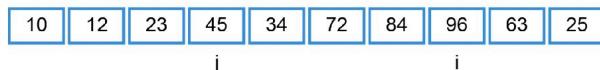
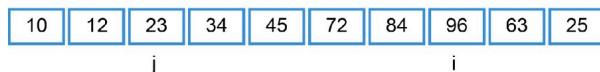
 $i=6, j=0$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 45 | 72 | 84 | 96 | 34 | 63 | 25 |
|----|----|----|----|----|----|----|----|----|----|

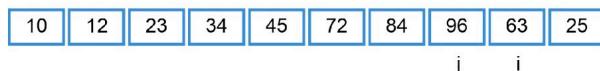
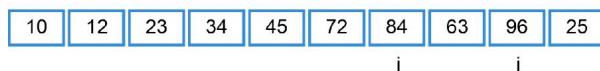
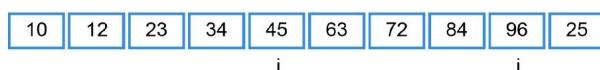
j i

$i=6, j=0$ 

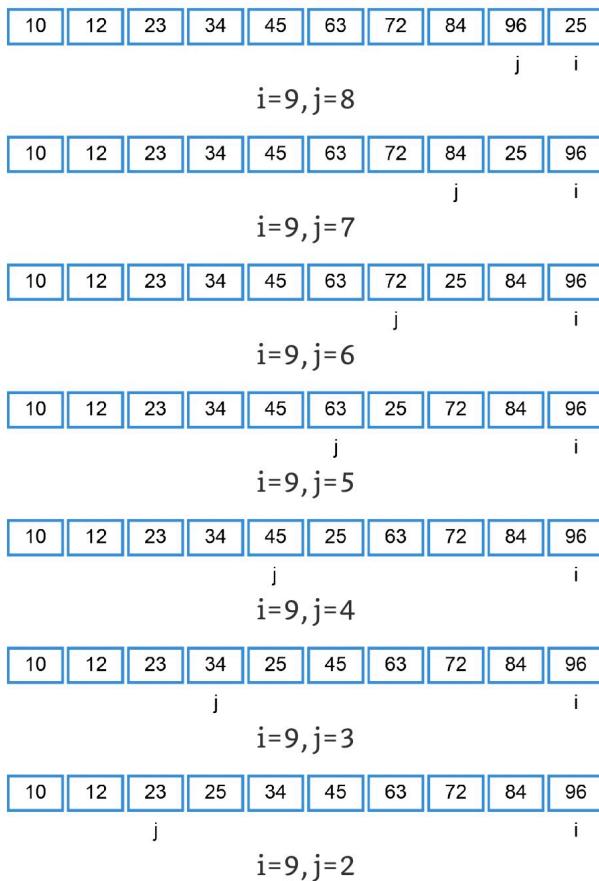
Iteración 7:

 $i=7, j=6$  $i=7, j=5$  $i=7, j=4$  $i=7, j=3$  $i=7, j=2$ 

Iteración 8:

 $i=8, j=7$  $i=8, j=6$  $i=8, j=5$  $i=8, j=4$

Iteración 9 y última:



Este es el fin de la última iteración y el arreglo termina ordenado.

### 5.4.1 Algoritmo e implementación de Insertion Sort

El algoritmo de **Insertion Sort** se puede describir como se muestra a continuación.

|                      |               |                                                                      |
|----------------------|---------------|----------------------------------------------------------------------|
| <b>insertionSort</b> | Descripción   | Algoritmo de ordenamiento de arreglos                                |
|                      | Entrada       | Arreglo <b>datos</b> a ser ordenado, de tamaño <b>tam</b>            |
|                      | Salida        | Arreglo <b>datos</b> ordenado                                        |
|                      | Precondición  | Un arreglo válido como parte de los atributos del ADT <b>Arreglo</b> |
|                      | Postcondición | El arreglo ordenado es el mismo que recibió                          |

1. Para **i=1** hasta **tam-1** {
2.   Para **j = i-1** 1 hasta 0 // decrementando
3.     **If** **datos[j+1] < datos[j]**
4.       Intercambiar el elemento en la posición **j+1** con el de la posición **j**
5.     **Else**
6.       Salir del ciclo // el elemento se encuentra en el lugar correcto
7. }

Al tener el algoritmo, su implementación en cualquier lenguaje de programación es muy directa. En C++ la podemos implementar como una operación (método) del ADT **Arreglo**:

```

1. // Ordena el arreglo usando Insertion Sort
2. void insertionSort(){
3. for (int i = 1; i < tam; i++){
4. for (int j = i-1; j >= 0; j--)
5. if (datos[j+1] < datos[j])
6. intercambiar(j+1,j);
7. else
8. break;
9. }
10. }
```

Se utilizan dos ciclos **for**. El primero (línea 3) es para recorrer todos los elementos del arreglo, excepto el primero, porque ya está ordenado. Con el fin de insertarlos en el lugar que les corresponde en el arreglo ordenado, por lo que va de 1 a **tam-1**. El segundo (línea 4) es el que recorre el arreglo ya ordenado, para colocar el elemento actual en el lugar que le corresponde, por lo que va de **i-1** hasta **0** o hasta que se encuentre un elemento menor antes que el actual. Si el elemento anterior al actual es menor (línea 5), se intercambian (líneas 6), si no (línea 7), se sale del segundo ciclo (líneas 8).

En cada iteración se puede usar el método **imprimir** para ir

viendo cómo va quedando el arreglo. Recuerda que, cuando se desea utilizar este método se debe declarar un objeto de la clase **Arreglo**, digamos **miArreglo**, llenar el arreglo con los datos a ser ordenados y luego invocar el método **insertionSort**, simplemente con el operador punto, es decir, **miArreglo.insertionSort()**.

### 5.4.2 Complejidad del algoritmo Insertion Sort

La operación básica de este algoritmo de ordenamiento también es la comparación. Su complejidad (orden) la daremos como una función del número de comparaciones que realiza en el peor caso.

En la iteración 1 solo hace 1 comparación, entre el elemento en la posición 0 y la 1. En la segunda iteración hace, máximo, dos comparaciones; esto es, el elemento a ordenar con el de la posición 0 y 1, y así sucesivamente, hasta llegar a la iteración **n-1** en donde hará máximo **n-1** comparaciones.

Si consideramos que el arreglo contiene **n** elementos, hará **n-1** iteraciones, ya que la inicial con solo el 0 no es necesaria. Esto significa que el número de comparaciones está dado por la suma:

$$\text{Comparaciones} = 1 + 2 + \dots + (n - 1)$$

Esta sumatoria vuelve a ser la suma de Gauss, pero con límite superior **n-1** en lugar de **n**. Es exactamente igual que el caso del **Selection Sort** (ver sección 5.3.3) y por lo tanto su complejidad es:

$$O(n^2)$$

**Selection Sort** e **Insertion Sort** tienen exactamente la misma complejidad.

## 5.5 Bubble Sort

**E**l algoritmo de burbuja (**Bubble Sort**, en inglés), no es muy natural de imaginar, pero sí es muy simple de implementar. Resulta igualmente ineficiente que los dos anteriores, sin embargo, se puede agregar una bandera para que termine en cuanto el arreglo esté ordenado y no siga haciendo comparaciones. Esta bandera solo nos ayuda a mejorar su complejidad en el caso promedio y no en el peor caso. Lo veremos en la implementación.

La idea es muy interesante, se inicia recorriendo todo el arreglo y comparando cada elemento con el que le sigue. En caso de que el que le sigue, sea menor que el actual, se intercambian, esto hace que al final de la primera pasada, el elemento más grande quede colocado al final del arreglo. Se vuelve a hacer otra pasada desde el inicio, pero ahora se omite el último puesto que ya está ordenado, y así sucesivamente.

Si pensamos en el proceso completo y en un arreglo colocado verticalmente, los elementos más grandes (más pesados) van al final del arreglo (al fondo), y los más pequeños (más ligeros) empiezan a moverse hacia el inicio del arreglo (hacia arriba). Esto da la idea del movimiento que tiene las burbujas en el agua (de ahí su nombre), donde van del fondo hacia arriba porque son más ligeras.

Veamos un ejemplo y pongamos atención al efecto de “burbujeo” de los elementos más pequeños.

### *Ejemplo de Bubble Sort*

Supongamos que tenemos el siguiente arreglo de 10 números (ahora con una representación horizontal con la posición 0 a la izquierda) y se quiere ordenar en forma ascendente:

| 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
|----|----|----|----|----|----|----|----|----|----|
| 45 | 23 | 12 | 96 | 84 | 72 | 10 | 34 | 63 | 25 |

Vamos a requerir un índice **i**, que indique el número de iteración que se está realizando. Se harán **tam-1** iteraciones (por lo que **i** va de 1 a **tam-1**, donde **tam** es el número de elementos que contiene el arreglo), y otra variable índice **j**, que nos ayude a recorrer el arreglo desde el inicio hasta el último elemento que todavía no esté ordenado (por lo que va desde 0 hasta **tam-i-1**). Iniciamos:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 45 | 23 | 12 | 96 | 84 | 72 | 10 | 34 | 63 | 25 |
| j  |    |    |    |    |    |    |    |    |    |

$$i=0, j=1$$

En la primera iteración comparamos el elemento en la posición **j** con el **j+1**. Como  $23 < 45$ , se intercambian. Se incrementa **j**:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 23 | 45 | 12 | 96 | 84 | 72 | 10 | 34 | 63 | 25 |
| j  |    |    |    |    |    |    |    |    |    |

$$i=1, j=1$$

Se compara nuevamente el **j** con el **j+1**. Como  $12 < 23$ , se intercambian. Se incrementa **j**:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 23 | 12 | 45 | 96 | 84 | 72 | 10 | 34 | 63 | 25 |
| j  |    |    |    |    |    |    |    |    |    |

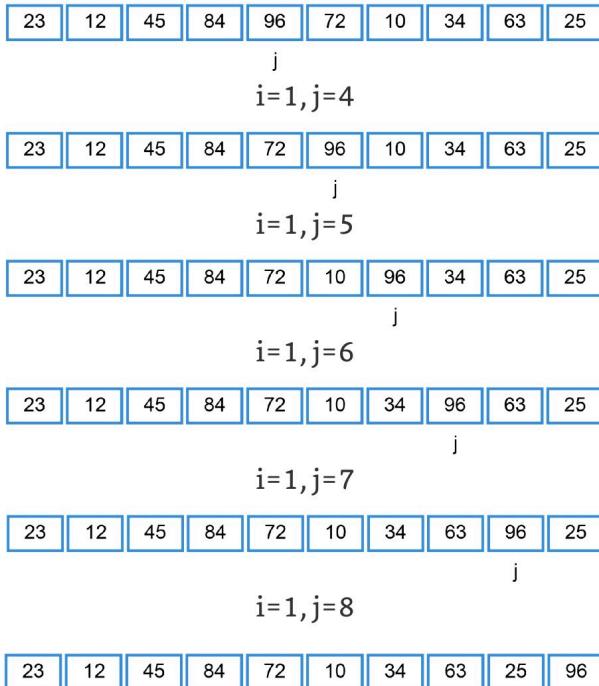
$$i=1, j=2$$

Se compara y se encuentra que  $96 > 45$ , por lo que no se intercambian. Se incrementa **j**:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 23 | 12 | 45 | 96 | 84 | 72 | 10 | 34 | 63 | 25 |
| j  |    |    |    |    |    |    |    |    |    |

$$i=1, j=3$$

El proceso continúa hasta el final:

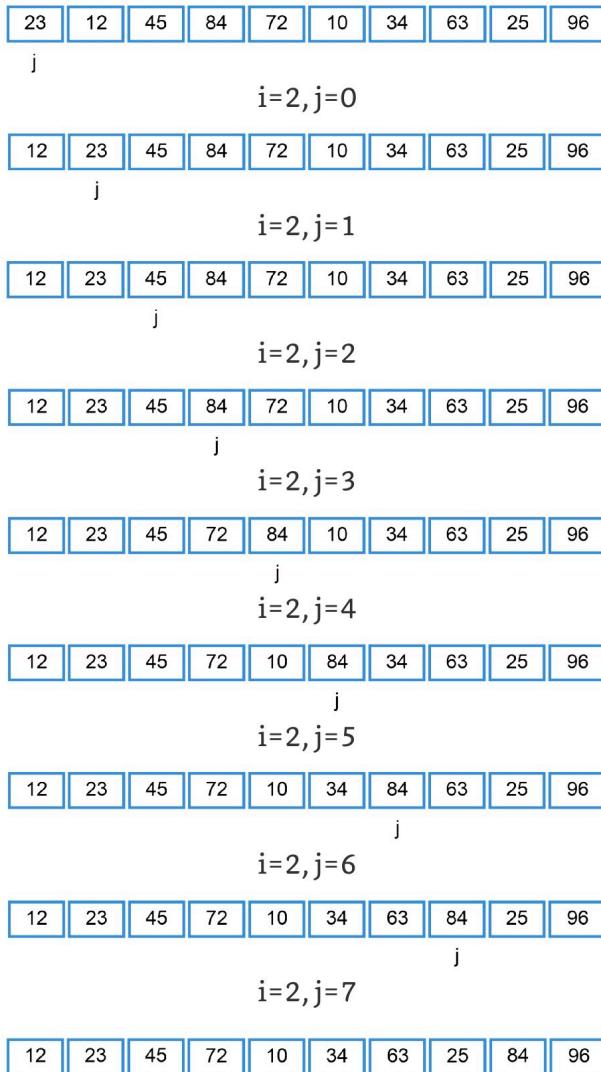


Como la **j** ya llegó a **tam-1**, la iteración **1** termina. Se incrementa **i** en **1**, se coloca **j** a **0** y se repite el proceso, pero ahora **j** llegará solo al elemento **7** ya que el **9** ya está ordenado.

El valor de **i** nos ayuda a saber en qué momento detenemos el incremento de **j**, lo cual sucederá al llegar al lugar **tam-i-1**. Un poco complicado entenderlo, pero, por ejemplo, para la iteración **1** (**i=1**), la **j** deberá llegar hasta el lugar  $10-1-1 = 8$ , para poderlo comparar con el **9**, en la iteración **2** la **j** deberá llegar hasta  $10-2-1 = 7$ , para poderlo comparar con el **8**, etc.

Se puede observar que al final de la iteración **1** se garantiza que el elemento más grande está al final, es decir, está ordenado (se fue al fondo). Se observa también que los elementos pequeños (ligeros) se empieza a mover hacia la izquierda y los grandes (pe-

sados) hacia la derecha, creando un efecto de que los más ligeros están “burbujeando” hacia la superficie (hacia la izquierda, si se viera en forma vertical sería hacia arriba, y parecerían más burbujas en agua). La iteración 2 se daría así:



Como la  $j$  ya llegó a 7, es el fin de la iteración 2. En este mo-

mento todos los elementos de la posición 8 en adelante, están ordenados. Se incrementa la **i**, se coloca la **j** a 0 y se inicia la siguiente iteración. Iteración 3

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 72 | 10 | 34 | 63 | 25 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

$$i=3, j=0$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 72 | 10 | 34 | 63 | 25 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

$$i=3, j=1$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 72 | 10 | 34 | 63 | 25 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

$$i=3, j=2$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 72 | 10 | 34 | 63 | 25 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

$$i=3, j=3$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 10 | 72 | 34 | 63 | 25 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

$$i=3, j=4$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 10 | 34 | 72 | 63 | 25 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

$$i=3, j=5$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 10 | 34 | 63 | 72 | 25 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

$$i=3, j=6$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 10 | 34 | 63 | 25 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

Termina la iteración 3, con todos los elementos de la posición 7 hacia la derecha completamente ordenados. Iteración 4

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 10 | 34 | 63 | 25 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

$$i=4, j=0$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 10 | 34 | 63 | 25 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

 $i=4, j=1$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 45 | 10 | 34 | 63 | 25 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

 $i=4, j=2$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 10 | 45 | 34 | 63 | 25 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

 $i=4, j=3$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 10 | 34 | 45 | 63 | 25 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

 $i=4, j=4$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 10 | 34 | 45 | 63 | 25 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

 $i=4, j=5$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 10 | 34 | 45 | 25 | 63 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

## Iteración 5

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 10 | 34 | 45 | 25 | 63 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

 $i=5, j=0$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 23 | 10 | 34 | 45 | 25 | 63 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

 $i=5, j=1$ 

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 10 | 23 | 34 | 45 | 25 | 63 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

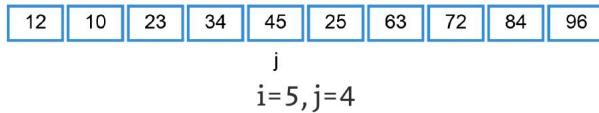
j

 $i=5, j=2$ 

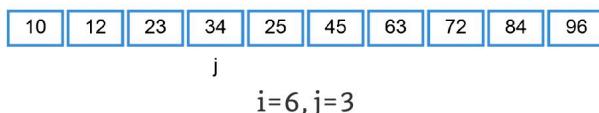
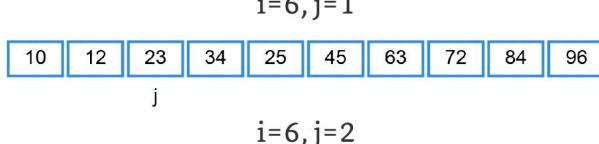
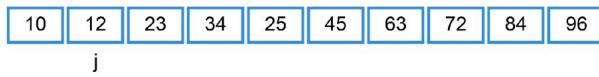
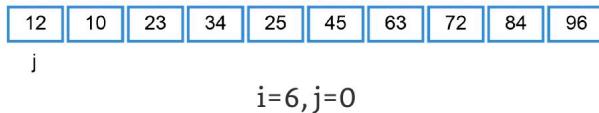
|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 12 | 10 | 23 | 34 | 45 | 25 | 63 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

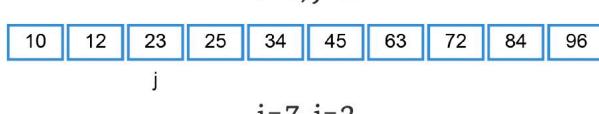
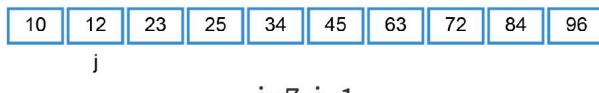
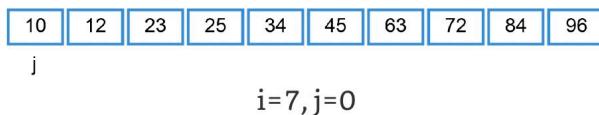
 $i=5, j=3$



Iteración 6



Iteración 7



|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 25 | 34 | 45 | 63 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

## Iteración 8

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 25 | 34 | 45 | 63 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

$$i=8, j=0$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 25 | 34 | 45 | 63 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

$$i=8, j=1$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 25 | 34 | 45 | 63 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

## Iteración 9

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 25 | 34 | 45 | 63 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

j

$$i=9, j=0$$

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 10 | 12 | 23 | 25 | 34 | 45 | 63 | 72 | 84 | 96 |
|----|----|----|----|----|----|----|----|----|----|

Después de la última iteración (**tam-1**) se garantiza que el arreglo está completamente ordenado.

Si se analizan detenidamente las iteraciones se observará que el arreglo estaba ordenado desde la iteración 6, este es uno de los principales problemas que tiene **Bubble Sort**, no se da cuenta cuando el arreglo ya está ordenado. Si se le da como entrada un arreglo ordenado, el algoritmo realizará todas las iteraciones necesarias sin hacer un solo intercambio.

Esto se arregla fácilmente con una bandera, condición que nos indica que el arreglo ya está ordenado y que en una iteración no se hace ningún intercambio; en este momento, se debe terminar el proceso de ordenamiento. Veremos esta pequeña modificación en la implementación.

### 5.5.1 Algoritmo e implementación de Bubble Sort

El algoritmo de Bubble Sort se puede describir como se muestra a continuación.

|                   |               |                                                                      |
|-------------------|---------------|----------------------------------------------------------------------|
| <b>bubbleSort</b> | Descripción   | Algoritmo de ordenamiento de arreglos                                |
|                   | Entrada       | Arreglo <b>datos</b> a ser ordenado, de tamaño <b>tam</b>            |
|                   | Salida        | Arreglo <b>datos</b> ordenado                                        |
|                   | Precondición  | Un arreglo válido como parte de los atributos del ADT <b>Arreglo</b> |
|                   | Postcondición | El arreglo ordenado es el mismo que recibió                          |

1. Para **i=1** hasta **tam-1**
2. Para **j = 0** hasta **tam-i-1**
3.     **If** **datos[j+1] < datos[j]**
4.         Intercambiar el elemento en la posición **j+1** con el de la posición **j**

En C++ podemos implementar este algoritmo como una operación (método) del ADT **Arreglo**:

```

1. // Ordena el arreglo usando Bubble Sort
2. void bubbleSort(){
3. for (int i = 1; i < tam; i++)
4. for (int j = 0; j < tam-i; j++)
5. if (datos[j+1] < datos[j])
6. intercambiar(j+1,j);

```

De la misma forma que los dos anteriores, se usan dos ciclos **for**. El primero (línea 3) nos indica el número de iteración que se va a realizar. El segundo (línea 4) es para recorrer la parte del arreglo que sigue desordenada, esto es, desde 0 hasta **tam-i-1**. Coloquemos ahora la bandera de salida (línea 3):

```

1. // Ordena el arreglo usando Bubble Sort
2. void bubbleSortBandera(){
3. bool bandera;
4. for (int i = 1; i < tam; i++){
5. bandera = false;
6. for (int j = 0; j < tam-i; j++){
7. if (datos[j+1] < datos[j]){
8. intercambiar(j+1,j);
9. bandera = true;
10. }
11. if (bandera == false){ // es lo mismo que if (!bandera)
12. cout << i << endl;
13. break;
14. }
15. }
16. }

```

Hay varias formas de colocarla. En este caso, la bandera antes de cada iteración es FALSE (línea 5), si en alguna comparación se hace un intercambio, la cambiamos a TRUE (línea 9). Al final de la iteración verificamos la bandera. Si es TRUE, continuamos y si es FALSE (línea 11), nos salimos.

### 5.5.2 Complejidad del algoritmo Bubble Sort

Seguramente surgió la pregunta sobre cómo afecta en su complejidad la bandera agregada a la segunda versión del **Bubble Sort**. La respuesta es que no afecta en nada debido a que el cálculo de la complejidad se hace para el peor caso y la bandera solo afecta el comportamiento promedio. Con esta aclaración, procedamos al cálculo de la complejidad para la primera versión del algoritmo, el que no tiene bandera.

Usaremos también la comparación como la operación básica. Supongamos que el arreglo contiene **n** elementos. En la iteración 1 se realizan **n-1** comparaciones, ya que se comparan todos los elementos con el que lo sigue, lo cual nos permite llegar hasta al elemento **n-1** para poderlo comparar con el **n**. En la iteración 2 se realizan **n-2** comparaciones, y así sucesivamente hasta que en la iteración **n-1** se realiza solo una comparación, la del elemento en la posición 0 con el de la posición 1. Esto significa que el número de comparaciones está dado por la siguiente suma:

$$\text{Comparaciones} = (n - 1) + (n - 2) + \dots + 1$$

Esta sumatoria vuelve a ser la suma de Gauss escrita al revés, pero con límite superior (**n-1**) en lugar de **n**. Es exactamente igual que el caso del **Selection Sort** y por lo tanto su complejidad es:

$$O(n^2)$$

Selection Sort, Insertion Sort, y Bubble Sort tienen exactamente la misma complejidad. Desde luego que el Bubble Sort con bandera, al recibir un arreglo casi ordenado, tardará mucho menos que los otros, y en el mejor de los casos, al recibir un archivo completamente ordenado, solo tarda una iteración.

Los tres algoritmos vistos hasta ahora son los más simples, pero también los más ineficientes. Desde luego que existen algoritmos de ordenamiento más eficientes, los cuales llegan a tener una complejidad casi lineal. Estos algoritmos son más complejos y se dificulta un poco más su implementación. A continuación, veremos dos de ellos.

## 5.6 Merge Sort

**M**erge Sort u ordenamiento por mezcla, es un algoritmo considerablemente más eficiente en tiempo de ejecución que los algoritmos previamente vistos. Es un método de ordenamiento basado en el principio de divide y vencerás, convirtiéndolo en un algoritmo bastante fácil de comprender.

La parte fundamental de **Merge Sort**, es realizar la mezcla; vamos a tratar de explicar este principio con una sencilla analogía. Supón que tienes dos filas de personas y que en cada una de ellas las personas están ordenadas por estatura, a continuación, quieres formar de esas dos filas, solo una fila con todas las personas ordenadas por estaturas.

Si aprovechamos el hecho de que cada fila ya está ordenada, el proceso de crear la fila es bastante más sencillo. Una manera muy sencilla de hacer esta tarea sería comparar a las personas que se encuentran al inicio de cada fila y ver quién es la persona menor, poner a esa persona al inicio de una nueva fila (cuidado,

solo pasas a uno, el menor). Repites la comparación con cada una de las personas que quedaron al inicio de la fila para ver quién es el siguiente al que le corresponde pasar a la nueva fila. Después de pasar a varias personas, una de las filas quedará vacía y la otra, al menos, tendrá a una persona, por lo que lo último que quedaría será pasar a las personas que aún están en una de las filas originales, al final de la lista nueva. Evidentemente la fila nueva tendrá a las personas de ambas filas, pero ordenadas de menor a mayor.

Este proceso es bastante sencillo, pero considera que las filas originales ya están ordenadas y esto es algo que no siempre contamos al inicio. Lo que hace Merge Sort es “partir” la lista original por mitad y tener dos listas, a su vez cada sublista se vuelve a partir y así sucesivamente hasta llegar a tener  $n$  listas de tamaño 1. Podemos decir que cada sublista se encuentra ordenada, pues son de tamaño 1 y entonces comenzamos a juntar las listas como se fueron partiendo, pero realizando el proceso de mezcla que se menciona arriba. Esto provocará que al final tengamos una lista ordenada.

### *Ejemplo de Merge Sort*

Tenemos los siguientes elementos que deseamos ordenar 10, 52, 20, 41, 50, 39, 25 y 86. El proceso de división y mezcla sería el siguiente.

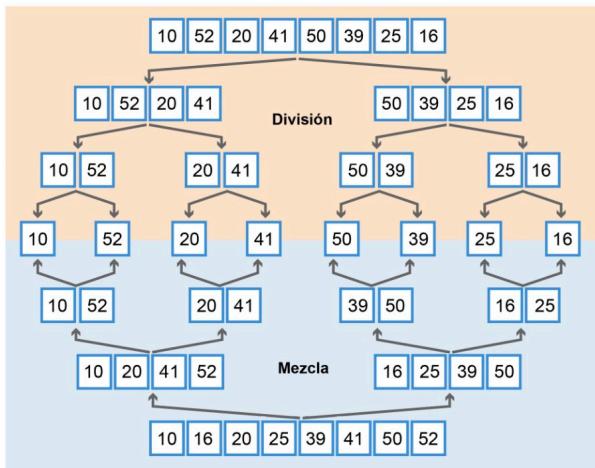


Figura 5.2 Proceso de división y mezcla en Merge Sort

Es importante mencionar que realmente no partimos el arreglo, sino que nos vamos a auxiliar de unos índices para indicar de dónde a dónde consideramos una mitad del arreglo y de dónde a dónde la otra mitad, es decir solo es una partición conceptual.

### 5.6.1 Algoritmo e implementación de Merge Sort

**Merge Sort** es de naturaleza recursiva, su implementación suele llevar una función para llamar a la función recursiva con los parámetros iniciales. Además, necesitamos realizar un segundo paso que consiste en la mezcla.

El algoritmo recursivo de **Merge Sort** se puede describir como se muestra a continuación

|                  |               |                                                                                                                                                                        |
|------------------|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>mergeSort</b> | Descripción   | Algoritmo de ordenamiento de arreglos                                                                                                                                  |
|                  | Entrada       | Arreglo <b>datos</b> a ser ordenado, de tamaño <b>tam</b><br>El índice <b>inicio</b> desde dónde queremos ordenar<br>El índice <b>fin</b> hasta dónde queremos ordenar |
|                  | Salida        | Arreglo <b>datos</b> ordenado                                                                                                                                          |
|                  | Precondición  | Un arreglo base 0 (la primera posición es 0) válido como parte de los atributos del ADT <b>Arreglo</b>                                                                 |
|                  | Postcondición | El arreglo ordenado es el mismo que recibió                                                                                                                            |

- Inicio representa el índice del arreglo a partir de donde queremos ordenar
- Fin representa el índice del arreglo hasta donde queremos ordenar

1. mergeSort (datos, inicio, fin)
2. Si (**inicio < fin**)
3.     centro = (**inicio + fin**) / 2
4.     mergeSort (datos, inicio, centro)
5.     mergeSort (datos, centro + 1, fin)
6.     mezcla (datos, inicio, fin)

|               |               |                                                                                                                                                 |
|---------------|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Mezcla</b> | Descripción   | Algoritmo de mezclar ordenadamente dos listas                                                                                                   |
|               | Entrada       | Arreglo <b>datos</b> a ser mezclado entre los índices <b>inicio</b> a <b>mitad</b> y de <b>mitad+1</b> a <b>fin</b>                             |
|               | Salida        | Arreglo <b>datos</b> ordenado de <b>inicio</b> a <b>fin</b>                                                                                     |
|               | Precondición  | Los elementos del arreglo de <b>inicio</b> a la <b>mitad</b> deben estar ordenados y de la <b>mitad + 1</b> a <b>fin</b> deben estar ordenados. |
|               | Postcondición | El arreglo ordenado es el mismo que recibió                                                                                                     |

- El parámetro **inicio** representa el índice a partir de donde queremos mezclar ordenadamente
- El parámetro **fin** representa el índice hasta donde queremos mezclar ordenadamente

1. mezcla (datos, inicio, fin)
2. Crear arreglo **datosTmp** donde quepan los datos contenido de **inicio** a **fin**. (Aquí colocaremos de manera temporal la lista ordenada con los elementos localizados entre **inicio** y **fin**)
3. **centro**= el punto medio entre **inicio** y **fin**.
4. **j = inicio**
5. **k= centro + 1**
6. Para **i=0** hasta **fin - inicio** y mientras **j <=centro** y **k<=fin**
7. Si **datos[ j ] < datos[ k ]**
8.     **datosTmp[ i ]=datos[ j++ ]**
9. Sino
10.     **datosTmp[ i ]=datos[ k++ ]**
11. Terminar de pasar los valores que quedan pendientes de pasar de una de las listas.

Implementaremos en C++ este algoritmo como una operación

(método) del ADT Arreglo:

```

1. void mezcla(int inicio, int fin){
2. int centro = (inicio + fin) / 2;
3. int j = inicio,
4. k = centro + 1,
5. size = fin - inicio + 1;
6. int datosTmp[size];
7.
8. for (int i = 0; i < size; i++){
9. if(j <= centro && k <= fin){
10. if(datos[j] < datos[k]){
11. datosTmp[i] = datos[j++];
12. }else {
13. datosTmp[i] = datos[k++];
14. }
15. }
16. else if(j <= centro){
17. datosTmp[i] = datos[j++];
18. }else{
19. datosTmp[i] = datos[k++];
20. }
21. }
22.
23. for (int m = 0; m < size; m++){
24. datos[inicio + m] = datosTmp[m];
25. }
26. }
27.
28. void mergesort(int inicio, int fin) {
29. if(inicio < fin) {
30. int centro = (inicio + fin) / 2;
31. mergesort(inicio, centro);
32. mergesort(centro + 1, fin);
33. mezcla(inicio, fin);
34. }
35. }
36.
37. void mergesort() {
38. mergesort(0, tam-1);
39. }
```

Como podrás darte cuenta los índices **inicio** y **fin** sirven para delimitar la parte del arreglo que vas a tratar de ordenar y posteriormente calcula el centro del arreglo para generar 2 sublistas de la original.

### 5.6.2 Complejidad del algoritmo Merge Sort

Para analizar la complejidad de **Merge Sort** revisemos primero la complejidad de la mezcla. En la función mezcla recorremos el arreglo de inicio a fin y sabemos que inicio al menos vale 0 y fin máximo vale **tam-1**. Luego lo volvemos a recorrer para regresar los datos copiados, por lo tanto la función mezcla es de **O(2n)** y por lo tanto es **O(n)**.

Si revisamos la función **mergesort**, lo que hace es partir por

la mitad la lista, dando como resultado dos sublistas, luego se llama recursivamente pasando cada una de las dos sublistas hasta llegar a una lista de tamaño 1. Aquí cabe preguntarse un numero **n**, ¿cuántas veces se le puede sacar mitad entera hasta llegar a 1? Evidentemente **log2(n)**, esto quiere decir que la función **mergesort** se va a llamar a si misma **log2(n)** veces y en cada llamada va a llamar a la operación mezcla que ya vimos que es de orden lineal. Por lo tanto, podemos concluir que su complejidad es:

$$O(n \log_2(n))$$

Si lo comparamos con los algoritmos previamente vistos, esto se traduce en un gran ahorro en tiempo de ejecución. pues seguramente ordenar una cantidad de valores que a **Merge Sort** le lleve poco menos de un minuto equivale a que un algoritmo como **Bubble Sort** le lleve bastantes horas.

## 5.7 Quick Sort

**Q**uick Sort es otro algoritmo de naturaleza recursiva que utiliza el principio de divide y vencerás. En el mejor de los casos resulta ser muy eficiente, a diferencia de **Merge Sort** e, igual que los algoritmos previamente vistos, funciona sin utilizar espacio adicional. Esto lo hace uno de los algoritmos de ordenamiento más utilizados.

El fundamento de Quick Sort para ordenar un arreglo consiste en utilizar un elemento del arreglo al cual llamaremos pivote. Posteriormente, colocaremos los elementos del arreglo que son menores al pivote en el inicio del arreglo, los elementos mayores al pivote al final de arreglo y finalmente dejar al pivote entre las dos sublistas que acabamos de generar. A este procedimiento se le conoce como partición.



Figura 5.3 Partición de un arreglo alrededor de un pivote p.

Observa que, al finalizar la partición, el pivote queda en la que será su posición final una vez que el arreglo esté ordenado. Luego, llamaremos de manera recursiva a Quick Sort para ordenar la lista que quedó al inicio del arreglo, la de los elementos menores al pivote, y para ordenar la lista que quedó después del pivote, o sea la de los elementos mayores al pivote.

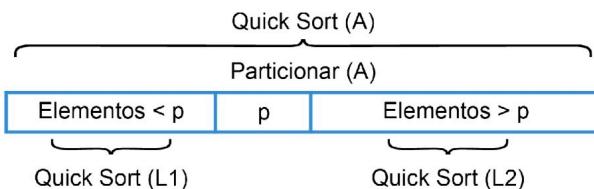


Figura 5.4 Pasos de Quick Sort para ordenar el arreglo A.

Aún no hemos hablado de cuál elemento es el que debemos seleccionar como pivote, de momento simplemente tomaremos el elemento que se localiza al inicio de la lista que queremos ordenar.

Tampoco hemos mencionado el procedimiento para hacer la partición, es posible hacerla de diferentes maneras y algunas más eficientes que otras. Tony Hoare, inventor de este algoritmo propuso tomar el primer elemento como pivote, comenzar a recorrer el arreglo de la segunda posición hacia atrás con un índice (*i*) y detenerse al encontrar un elemento mayor que el pivote; después comenzar a recorrer el arreglo de la última posición hacia adelante con otro índice (*j*) y detenerse al encontrar un elemento menor que el pivote. Luego, es necesario hacer un intercambio del elemento en la posición *i*, con el elemento de la posición *j*; continuar los recorridos y los intercambios hasta que

los índices **i** y **j** se cruzaran. Finalmente hacer un intercambio del pivote que está al inicio de la lista con la posición donde quedó el índice **j**. De esta manera quedaría nuestro arreglo particionado alrededor del pivote.

También existe otra manera que es la que explicaremos a fondo; consiste básicamente en tener en nuestro arreglo 4 secciones: en la primera posición de nuestra lista a ordenar estará nuestro pivote, luego tendremos algunos elementos que son menores a nuestro pivote, después algunos elementos que son mayores a nuestro pivote y finalmente algunos elementos que aún no hemos clasificado como menores o mayores. Para delimitar las secciones nos auxiliaremos de un par de índices **i** y **j**.

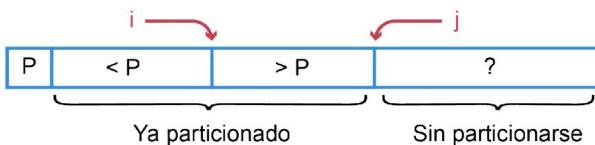


Figura 5.5 Secciones en el proceso de partición.

Al comenzar la partición solo tendremos nuestro pivote al inicio, todos los demás elementos estarán en la sección de elementos no particionados, o sea **i** y **j** haciendo referencia a la posición después del pivote. Vamos a ir recorriendo el arreglo **y**, en cada iteración, pasaremos un elemento a la sección particionada, siempre avanzaremos **j** una posición. Si el elemento que se acaba de incorporar a la sección particionada es menor que el valor del pivote, entonces realizaremos un intercambio de este elemento con el que se localiza en la posición **i**, e incrementaremos **i** en una unidad. Esto provocará que nuestro espacio particionado vaya incrementando, mientras que la sección de elementos sin particionar se reducirá hasta quedar vacío. Finalmente, para ubicar al pivote en su posición final, lo intercambiaremos con el elemento

localizado en **i-1**.

### Ejemplo de partición alrededor del pivote

Tenemos la siguiente lista de valores  $\{30, 52, 20, 41, 50, 39, 25, 86\}$  que deseamos particionar alrededor de elemento que se localiza al inicio.

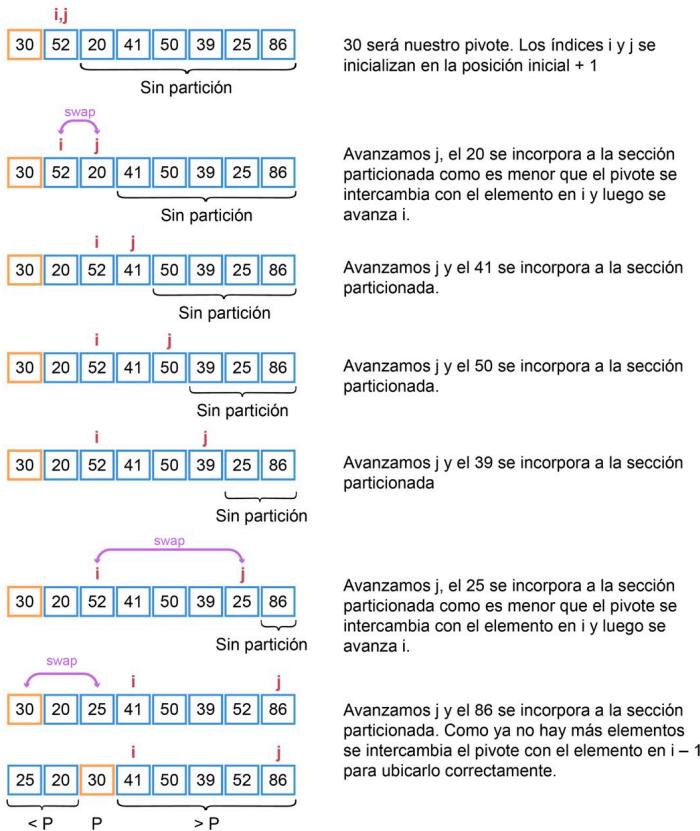
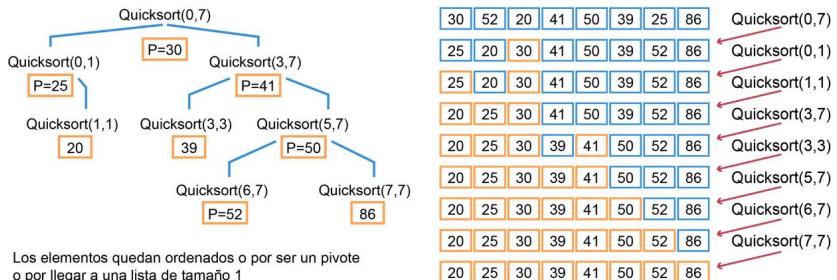


Figura 5.6 Ejemplo de partición de un arreglo alrededor de un pivote localizado al inicio.

### Ejemplo de ordenamiento con Quick Sort

Como ya mencionamos previamente, después de hacer la par-

tición se llama de manera recursiva **Quick Sort** para cada una de las dos sublistas que se generaron, la lista de los menores al pivote y la lista de los mayores al pivote, hasta llegar a una lista de tamaño 1.



### 5.7.1 Algoritmo e implementación de Quick Sort

| quickSort | Descripción   | Algoritmo de ordenamiento de arreglos                                                                                                                                  |
|-----------|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|           | Entrada       | Arreglo <b>datos</b> a ser ordenado, de tamaño <b>tam</b><br>El índice <b>inicio</b> desde dónde queremos ordenar<br>El índice <b>fin</b> hasta dónde queremos ordenar |
|           | Salida        | Arreglo <b>datos</b> ordenado                                                                                                                                          |
|           | Precondición  | Un arreglo base 0 (la primera posición es 0) válido como parte de los atributos del ADT <b>Arreglo</b>                                                                 |
|           | Postcondición | El arreglo ordenado es el mismo que recibió                                                                                                                            |

- Inicio representa el índice del arreglo a partir de donde queremos ordenar.
- Fin representa el índice del arreglo hasta donde queremos ordenar.

7. quickSort (**datos, inicio, fin**)
8. Si (**inicio < fin**)
9.     **posP** = particionar(**datos, inicio, fin**)
10.    quickSort (**datos, inicio, posP - 1**)
11.    quickSort (**datos, posP + 1, fin**)

|                    |               |                                                                                                                                                                                                    |
|--------------------|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>particionar</b> | Descripción   | Algoritmo de ordenamiento de arreglos                                                                                                                                                              |
|                    | Entrada       | Arreglo <b>datos</b> a ser ordenado, de tamaño <b>tam</b><br>El índice <b>inicio</b> desde dónde queremos considerar el arreglo<br>El índice <b>fin</b> hasta dónde queremos considerar el arreglo |
|                    | Salida        | Arreglo <b>datos</b> ordenado                                                                                                                                                                      |
|                    | Precondición  | Un arreglo base 0 (la primera posición es 0) válido como parte de los atributos del ADT <b>Arreglo</b>                                                                                             |
|                    | Postcondición | El arreglo ordenado es el mismo que recibió                                                                                                                                                        |

- Inicio representa el índice del arreglo a partir de donde queremos considerar el arreglo.
- Fin representa el índice del arreglo hasta donde queremos considerar el arreglo.

```

1. particionar(datos, inicio, fin)
2. piv=datos[inicio]
3. i = inicio + 1
4. para j=inicio + 1 hasta fin
5. Si datos[j] < piv
6. intercambiar A[i]yA[j]
7. i ++
8.
9. intercambiar A[inicio] y A [i - 1]

```

Implementaremos en C++ este algoritmo como una operación (método) del ADT **Arreglo**:

```

1. void intercambiar(int i,int j){
2. int paso=datos[i];
3. datos[i]=datos[j];
4. datos[j]=paso;
5. }
6.
7. int particionar(int primero,int ultimo) {
8. int pivot=datos[primero];
9. int i=primero+1;
10. for(int j=i;j<ultimo;j++) {
11. if(datos[j]<pivot) {
12. intercambiar(i++, j);
13. }
14. }
15. intercambiar(primero, i-1);
16. return i-1;
17. }
18.
19. void quicksort(int primero,int ultimo) {
20. if(primer<ultimo) {
21. int posPiv=particionar(primero, ultimo);
22. quicksort(primer, posPiv-1);
23. quicksort(posPiv+1,ultimo);
24. }
25. }
26.
27. void quicksort() {
28. quicksort(0, tam-1);
29. }

```

Como puedes observar, el método particionar además de hacer

la partición regresa la posición en la que dejó al pivote. De esta manera podemos hacer las llamadas recursivas a **Quick Sort** para que ordene las 2 sublistas generadas.

### 5.7.2 Complejidad del algoritmo Quick Sort

Ahora hablemos del desempeño que puede tomar Quick Sort. Primero veamos el número de comparaciones que hacemos en la **partición**, es claro que es del orden de **n**, dado que es un ciclo que va recorriendo el arreglo y en cada iteración se realizan operaciones de orden constante.

Para comprender la complejidad de **Quick Sort**, dependerá de cuántas veces se mande llamar la función recursiva. Primero supongamos que tenemos la mala fortuna de **siempre** seleccionar como pivote el elemento menor de la lista a ordenar.

$$n + (n - 1) + (n - 2) + \dots + 1$$

En ese caso obtendremos una sublista vacía y la otra se habrá reducido solo en 1 elemento. Entonces debido a la partición el número de comparaciones que tendríamos serían

$$n + (n - 1) + (n - 2) + \dots + 1$$

y lo que ya sabemos que es igual a

$$\frac{n(n + 1)}{2}$$

y por lo tanto es de

$$O(n^2).$$

Este sería el peor de los casos, curiosamente lo tendríamos al

intentar ordenar una lista que ya se encuentra ordenada y que tomamos como pivote el primer elemento de la lista a ordenar o el último elemento.

Ahora, supongamos que tenemos una función que mágicamente siempre selecciona como pivote el elemento que divide por mitad a los elementos del arreglo, al igual que como lo hacíamos en **Merge Sort**, en este caso el número de comparaciones iniciales serían  $n$  (por la partición del arreglo completo), a continuación, se hacen  $n/2$  comparaciones a las dos sublistas (por la partición a las 2 sublistas de tamaño  $n/2$ ) y luego 4 ( $n/4$ ) comparaciones y así hasta llegar a  $n(1/n)$  y esto lo hacemos un total de  $\log_2(n)$  veces, es decir:

$$1(n) + 2\left(\frac{n}{2}\right) + 4\left(\frac{n}{4}\right) + \dots + n\left(\frac{1}{n}\right) \text{ y esta suma la hacemos } \log_2(n) \text{ veces.}$$

Por lo tanto su complejidad en el mejor de los casos es  $O(n \log_2(n))$ .

Como podemos ver, la eficiencia del **Quick Sort** depende de la calidad del pivote, es decir, qué tan bien nos divide la lista en listas de un tamaño similar; para seleccionar el pivote existen diferentes formas de hacerlo. Una manera es tomar el valor mediano entre tres valores: el primer elemento de la lista, el último valor de la lista y el valor localizado al centro de la lista. Sin embargo, experimentos han mostrado que Quick Sort se comporta con una eficiencia cercana a  $O(n \log_2(n))$ , cuando la listas quedan divididas entre un 25% y un 75 % de sus elementos, es decir, la mitad de los valores de la lista son buenos candidatos para ser pivotes, por lo que suele dejarse el tomar cualquier valor de la lista como pivote. Un error sería hacer una función que seleccione el pivote y que esta función tuviera una alta complejidad de orden lineal en lugar de constante, esto automáticamente estaría per-

judicando la eficiencia del algoritmo.

Finalmente, en el caso promedio Quick Sort tiene también una complejidad  $O(n \log_2(n))$ . En resumen, la complejidad de Quick Sort es:

- Peor de los casos  $O(n^2)$ .
- Mejor y caso promedio  $O(n \log_2(n))$ .

## Ejercicios



1. ¿Cómo modificarías el algoritmo **Insertion Sort** si quieres ordenar los números en forma descendente?
2. ¿El algoritmo **Selection Sort** funcionaría también al usar números negativos? ¿Por qué?
3. ¿El algoritmo **Bubble Sort** funcionaría también si el arreglo contiene números repetidos? ¿Por qué?
4. Los algoritmos de ordenamiento funcionarían bien para otro tipo de datos, tal como se comentó anteriormente. Escriba tres ejemplos de datos que podría contener el arreglo para ordenarlo.
5. Al ordenar el siguiente arreglo de entrada, utilizando el algoritmo **Insertion Sort**:

|    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|
| 45 | 73 | 82 | 31 | 41 | 27 | 16 |
|----|----|----|----|----|----|----|

- a. Escriba el arreglo que se tendría al finalizar la iteración 1.
- b. Escriba el arreglo que se tendría al finalizar la iteración 4.
- c. Escriba el arreglo que se tendría al finalizar el segundo paso de la iteración 5. Recuerde que un paso es una ejecución del ciclo más interno.
6. Al ordenar el siguiente arreglo de entrada, utilizando el algoritmo **Selection Sort**:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| f | u | n | c | i | o | n | a |
|---|---|---|---|---|---|---|---|

- a.** Escriba el arreglo que se tendría al finalizar la iteración 2.
  - b.** Escriba el arreglo que se tendría al finalizar la iteración 4.
  - c.** Escriba el arreglo que se tendría al finalizar el segundo paso de la iteración 6. Recuerde que un paso es una ejecución del ciclo más interno.
- 7.** Al ordenar el siguiente arreglo de entrada, utilizando el algoritmo **Bubble Sort**:
- |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 67 | 21 | 11 | 42 | 28 | 75 | 39 | 19 |
|----|----|----|----|----|----|----|----|
- a.** Escriba cómo quedaría el arreglo al final de cada una de las iteraciones, usando la bandera para detenerlo.
  - b.** Escriba el arreglo que se tendría al finalizar el segundo paso de la iteración 6. Recuerde que un paso es una ejecución del ciclo más interno.
- 8.** Modifique el método **Insertion Sort** para poder ver cómo quedan los arreglos al final de cada iteración. Use un letrero para saber a qué iteración corresponde cada arreglo impreso. Pruébelo con el arreglo del ejercicio 5 y verifique las respuestas que obtuvo en el mismo.
- 9.** Modifique el método **Selection Sort** para poder ver cómo va el arreglo en cada paso de cada iteración. Use un letrero para saber el paso y la iteración de cada arreglo. Pruébelo con el arreglo del ejercicio 6 y úselo para comprobar sus respuestas a dicho ejercicio.

10. Al ordenar el siguiente arreglo de entrada, utilizando el algoritmo **Merge Sort**:

|   |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|
| 4 | 25 | 42 | 15 | 88 | 36 | 51 | 17 |
|---|----|----|----|----|----|----|----|

a. Dibuje el árbol en el proceso de división.

b. Dibuje el árbol en el proceso de mezcla.

11. Modifique el método **Merge Sort** para poder ver cómo va el arreglo en cada llamada recursiva. Use un letrero para conocer el número de llamada. Pruébelo con el arreglo del ejercicio 10 y úselo para comprobar sus repuestas a dicho ejercicio.

12. Al ordenar el siguiente arreglo de entrada, utilizando el algoritmo **Quick Sort**:

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 32 | 13 | 76 | 26 | 92 | 53 | 19 | 63 |
|----|----|----|----|----|----|----|----|

Dibuje el árbol que se forma en el proceso de partición con cada llamada recursiva.

13. Modifique el método **Quick Sort** para poder ver cómo va el arreglo en cada llamada recursiva. Use un letrero para conocer el número de llamada. Pruébelo con el arreglo del ejercicio 12 y úselo para comprobar sus repuestas a dicho ejercicio.



# Capítulo 6. Manejo de memoria

06

**M**uchos lenguajes de programación tienen la capacidad de manejar dos tipos de memorias principales:

- Estática: es la memoria que se utiliza tradicionalmente, normalmente se asocia a una variable u objeto fijo y permanece durante el tiempo de vida de la variable. Además, se conocen como automáticas porque son creadas automáticamente por el compilador al encontrar su declaración.
- Dinámica: este tipo de memoria se “crea” y se “destruye”, según las indicaciones del programador durante la ejecución del programa. Las variables y objetos creados dinámicamente son almacenados en un área llamada HEAP, administrada por el sistema operativo. El manejo de este tipo de memoria favorece su uso eficiente en la ejecución de un programa.

## 6.1 Apuntadores y referencias.

**U**n apuntador es una referencia indirecta a un elemento, ya que almacena la dirección de memoria donde el elemento en cuestión estará almacenado. Para poder declarar un apuntador se antepone un asterisco:

**tipo \*nombre\_apuntador;**

Ejemplos:

**int ent, \*ap\_entero;**

**char ch, \*ap\_char;**

Donde **ent** es una variable tipo **int**, **ap\_entero** es un apuntador

a **int**, ch es una variable tipo **char** y ap\_char es un apuntador a **char**.

Ya dentro del cuerpo del programa, existen dos operadores que se pueden utilizar:

- **Operador de indirección.** Se representa por asterisco (\*), accede al contenido de lo apuntado.
- **Operación de dirección.** Se representa por ampersand (&), accede a la dirección donde esta lo apuntado.

Ejemplo:

```

1. int a, *b; // a es una variable int, b es un apuntador a int.
2. a = 5; // la variable a contiene el valor de 5.
3. b = &a; // el apuntador b apunta a la dirección de la variable a.
4. *b = 20; // el contenido de lo que apunta b contiene ahora 20.
5. cout << a << " " << *b << endl ;
6.
7. //imprime 20 20, ya que b apunta a la memoria de a.

```

En la figura 6.1 se puede apreciar el comportamiento paso a paso del ejemplo anterior.

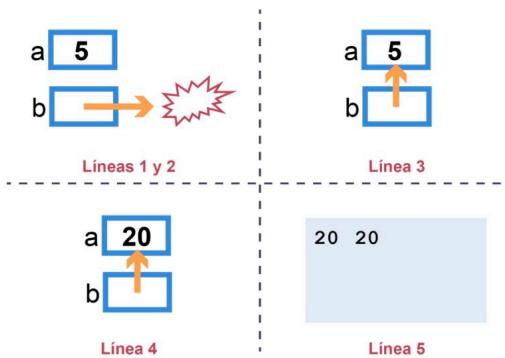


Figura 6.1 Ejemplo gráfico de apuntadores.

## 6.2 Administración de memoria dinámica.

**E**l espacio de almacenamiento dinámico se va creando y destruyendo durante la ejecución del programa, se realiza utilizando los operadores:

- **new.** Regresa la dirección del espacio de memoria que se acaba de crear, en caso de que no hubiera espacio regresa el valor de nullptr. La sintaxis para llamarlo es:

**apuntador = new tipo\_de\_dato;**

El apuntador toma el valor de la dirección y apunta al nuevo espacio.

- **delete.** Libera el espacio apuntado, dejando el apuntador con un valor indefinido. La sintaxis para llamarlo es:

**delete apuntador;**

El apuntador libera el espacio de memoria ocupado

Ejemplo:

```
1. int a, *b; // a es una variable int, b es un apuntador a int.
2. a = 5; // la variable a contiene el valor de 5.
3. b = new int; // el apuntador b apunta a una nueva posición de mem.
4. *b = 20; // el contenido de lo que apunta b contiene ahora 20.
5. cout << a << " " << *b << endl;
6.
7. //imprime 5 20
```

En la figura 6.2 se pude apreciar el comportamiento paso a paso del ejemplo anterior.

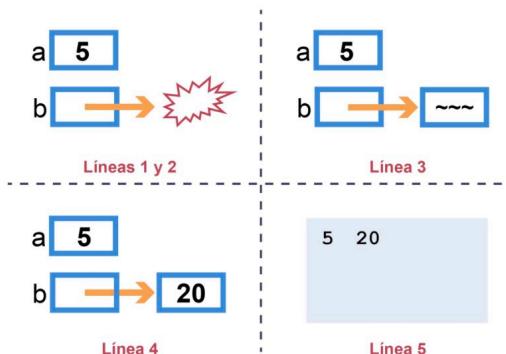


Figura 6.2. Ejemplo gráfico de apuntadores con memoria dinámica.

Para que el uso de la memoria dinámica funcione adecuadamente hay que seguir las siguientes reglas:

- Por cada ejecución de un comando **new**, se deberá ejecutar un comando **delete** antes de terminar la ejecución del comando.

En caso de que se realicen comandos **delete** adicionales, el programa tendrá un error de ejecución.

En caso de que no se realicen suficientes comandos **delete**, el programa dejaría memoria que el sistema operativo la considera ocupada sin estarlo.

- Un **delete** opera liberando el espacio de memoria apuntado, independientemente de que existan más apuntadores apuntando al mismo espacio.

Ejemplo:

```
1. int *p, *q, *r; // p q r son apuntadores a int.
2. p = new int; // el apuntador p apunta a una nueva pos. de mem.
3. q = r = p; // q y r apuntan a la misma dirección de p.
4. delete r; // liberas la memoria que r apuntaba.
```

En la figura 6.3 se puede apreciar el comportamiento paso a paso del ejemplo anterior.

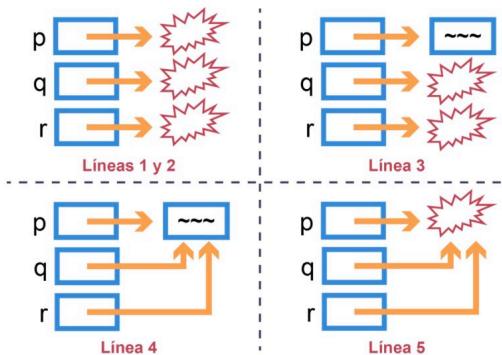


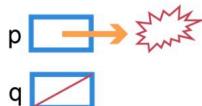
Figura 6.3 Ejemplo gráfico de múltiples apuntadores señalando un mismo espacio de memoria.

- Un apuntador local, que señala una función, se destruye al terminar la ejecución del mismo tal como cualquier variable local, sin importar a que espacio de memoria haga referencia. Es importante hacer el **delete** correspondiente antes de salir de la ejecución de la función.
- Basura es diferente de nada.

### Ejemplo:

```
1. int *p; // p es apuntador a int que apunta a basura.
2. int *q = nullptr; // q es apuntador a int que apunta a nullptr.
```

En la figura 6.4 se puede apreciar el comportamiento paso a paso del ejemplo anterior.



Líneas 1 y 2

Figura 6.4. Ejemplo gráfico de la diferencia de nullptr y basura.

- Hacer un **delete** a un apuntador que no tiene una referencia de memoria dinámica provocará un error de ejecución.
- Hacer un acceso a un apuntador cuyo valor sea **nullptr** provocará un error de ejecución.
- Asignar un valor a un apuntador con la operación **new**, perderá el contenido anterior sin importar lo que esté apuntando.
- Para asignarle un valor a un apuntador no siempre se necesita hacer con la operación **new**, también puede hacerse a través de el operador de **&** o asignándole el valor de otro apuntador.
- Los valores de los apuntadores solo se pueden comparar si apuntan a donde mismo. Solo se permite compararlos con el

operador de igualdad == o el operador de desigualdad !=.

- A un dato referenciado por un apuntador se le pueden hacer todas las operaciones que el dato permita.

# Ejercicios



- 1.** Para cada uno de los ejercicios, escribe lo que desplegaría en pantalla.

```
1. int a=10,*p;
2. p = &a;
3. cout << a << " " << *p << endl;
```

```
1. int a=10,*p *q;
2. p = q = &a;
3. *q++;
4. cout << *q << " " << *p << endl;
```

```
1. int a=1, b=2;
2. int *p = *q = &b;
3. *q++;
4. cout << b << " " << a << endl;
```

```
1. int a=1, b=2, *p, *q;
2. p = &a;
3. q = &b;
4. *q++;
5. cout << b << " " << a << endl;
```

```
1. nt a=10, b=20, *p, *q;
2. p = &a;
3. q = p;
4. *q+=30;
5. cout << *p << " " << *q << endl;
```

```
1. int a=10, b=20, *p, *q;
2. p = new int(5);
3. q = &a;
4. *q+=30;
5. cout << *p << " " << *q << " " << a << " " << b << endl;
6. delete p;
```

```
1. int a=10, b=20, *p, *q;
2. p = new int(5);
3. q = p;
4. *q+=30;
5. cout << *p << " " << *q << " " << a << " " << b << endl;
6. delete q;
```

```
1. int a=10, b=20, *p, *q;
2. p = new int(5);
3. q = new int(10);
4. *q+=30;
5. b = *q + *p;
6. cout << *p << " " << *q << " " << a << " " << b << endl;
7. delete q;
8. delete p;
```

```

1. int a=10, b=20, *p, *q;
2. p = new int(15);
3. q = &b;
4. *q+=30;
5. cout << *p << " " << *q << " " << a << " " << b << endl;
6. delete p;

```

```

1. int a=8, b=17, *p, *q;
2. p = &a;
3. q = new int(10);
4. *q+=a;
5. cout << *p << " " << *q << " " << a << " " << b << endl;
6. delete q;

```

**2.** Para cada uno de los ejercicios, escribe si habría o no error de ejecución, y si deja memoria volando y por qué.

```

1. int a=10,*p;
2. p = new int(10);
3. cout << a << " " << *p << endl;
4. delete p;

```

```

1. int a=10,*p *q;
2. q = new int(10);
3. *q++;
4. cout << *q << " " << *p << endl;
5. delete p;

```

```

1. int a=1, b=2;
2. int *p = new int(10);
3. int *q = &b;
4. *q++;
5. cout << b << " " << a << endl;
6. delete p;
7. delete q;

```

```

1. int a=1, b=2;
2. int *p = new int(10);
3. int *q = &b;
4. *q++;
5. cout << b << " " << a << endl;
6. delete p;
7. delete q;

```

```

1. int a=8, b=17, *p, *q;
2. p = new int(30);
3. q = new int(10);
4. *q+=a;
5. cout << *p << " " << *q << " " << a << " " << b << endl;
6. delete q;

```



## Capítulo 7. Introducción a las estructuras de datos

07

**U**no de los elementos más importantes en todo sistema computacional es el conjunto de datos con el que va a funcionar. Por ejemplo, un sistema bancario deberá contar con las transacciones realizadas por sus usuarios; un sistema de aprendizaje automático requerirá de los casos de ejemplo con los que se va a entrenar el sistema, un sistema escolar requiere de los registros de todos sus alumnos, etc.

Los datos siempre deben estar disponibles para que el sistema computacional los utilice. Si los datos son muy numerosos, será importante que su manejo se haga en forma eficiente para que el sistema computacional no tenga retrasos o tiempos muy largos para emitir una respuesta al usuario.

El sistema computacional siempre debe ser capaz de hacer ciertas operaciones sobre esos datos, las cuales normalmente incluyen: agregar datos nuevos, borrar datos que ya no se utilicen, modificar datos que ya están capturados y consultar datos que se tienen guardados.

A estas actividades se les conoce como **ABCC**: **A**ltas, **B**ajas, **C**ambios y **C**onsultas. Es importante que se entienda cuáles son las operaciones que se hacen “sobre” los datos y las que se hacen “con” los datos. Las operaciones ABCC, se hacen sobre los datos, por lo que el sistema computacional usa los datos disponibles para hacer operaciones con ellos. Las operaciones con los datos siempre incluyen la operación de búsqueda o consulta sobre los datos.

Con la separación anteriormente hecha de las operaciones, podemos concentrarnos en cómo guardar los datos y las operaciones que se hacen sobre ellos. De esto es de lo que tratan precisamente las estructuras de datos y el diseño de algoritmos, materia principal de este libro.

Podemos entonces definir una **estructura de datos** como una unidad especial para guardar los datos, la cual nos permite mantenerlos y utilizarlos de forma eficiente. Los procedimientos para mantener y utilizar los datos, como en cualquier sistema computacional, se definen usando algoritmos. Estos algoritmos deben ser muy eficientes para que el sistema funcione correctamente.

El **diseño y análisis de algoritmos** es la rama de las ciencias computacionales que nos permite diseñar algoritmos eficientes para una tarea específica, además de analizar su eficiencia formalmente (matemáticamente).

Podemos ahora entender completamente la famosa frase de Niklaus Wirth, gran científico de la computación, premio Turing 1984, quién afirmó que:

**Algoritmos + Estructuras de datos = Programas**

## 7.1 Estructura de datos

La forma física de guardar los datos es en la memoria de la computadora, la cual es un conjunto de celdas de datos consecutivas. Estas celdas están numeradas con un número que es la dirección de memoria, ese número es el que nos permite tener acceso a dichas celdas.

Las estructuras de datos nos permiten acomodar de formas diferentes los datos en la memoria. La forma más simple consiste

en guardar los datos en sitios consecutivos de la memoria, lo cual corresponde a la estructura de datos más famosa y utilizada de todos los tiempos: el arreglo.

En un arreglo, todos los datos guardados deben ser del mismo tipo, o la misma clase, hablando en programación orientada a objetos. Existen otras formas para guardar los datos en la memoria, las cuales dan ciertas ventajas para ciertos tipos de datos y en ciertos tipos de aplicaciones donde se utilicen dichos datos. Estas estructuras de datos las iremos estudiando conforme avance este libro. Por otro lado, los datos guardados en las estructuras tienen cierta relación. Por ejemplo, una relación de orden, una relación de predecesores y sucesores, etc. De acuerdo con esto, una posible clasificación de las estructuras de datos, de acuerdo con la relación de predecesores y sucesores entre los datos, es la que se muestra en la figura 7.1.

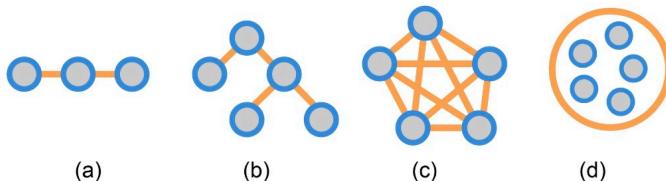


Figura 7.1. Relaciones entre los datos: (a) lineal, (b) jerárquica, (c) red, (d) sin relación

La figura 7.1(a) muestra una relación lineal entre los datos, en la cual, cada dato tiene un predecesor y un sucesor; excepto el primero, que solo tiene sucesor, y el último, que solo tiene predecesor. Es una relación uno a uno.

La figura 7.1(b) muestra una relación jerárquica de los datos, en donde cada dato puede tener muchos sucesores, pero solo un predecesor, como un organigrama. Es una relación uno a muchos.

La figura 7.1(c) muestra una relación de red entre los datos, en la cual, cada dato puede tener muchos predecesores y muchos sucesores. Es una relación muchos a muchos.

Finalmente, la figura 7.1(d) muestra un grupo de datos sin relación alguna. Este tipo de estructura no es muy usado debido a que la relación entre los datos es muy importante para diseñar sus procesos de acceso y modificación.

De acuerdo con criterios particulares, cada una de las clasificaciones anteriores puede tener subclasificaciones; por ejemplo, la posición física que cada dato tiene en la memoria. Si en una relación lineal, los datos se encuentran en posiciones consecutivas de la memoria física, la estructura de datos es un arreglo; si en la misma estructura lineal, los datos se encuentran en posiciones no continuas de la memoria física, se trata de una lista ligada.

El tipo de estructura también afecta la información adicional que debe guardar cada dato, además del dato en sí. Por ejemplo, si se trata de un arreglo, se sabe que el siguiente dato está en la siguiente celda (o conjunto de celdas) de la memoria, están contiguos. Sin embargo, en una lista ligada, cada dato deberá guardar la posición que su sucesor tiene en la memoria, debido a que no están contiguos. En los casos en que un dato debe contener información adicional para ir al siguiente dato en la estructura, se acostumbra a referirse al dato como nodo, el cual contiene un conjunto de información y no un solo dato. Una representación gráfica de este tipo de estructura de datos se puede observar en la figura 7.2.

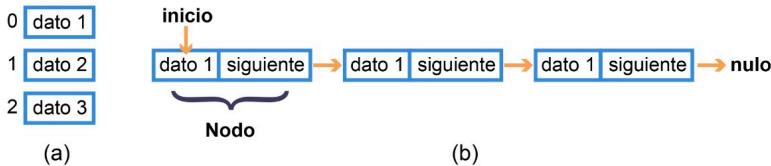


Figura 7.2. Diferentes implementaciones de una estructura lineal: (a) arreglo, (b) lista ligada.

## 7.2 Tipo de datos abstracto

Cuando se define una estructura de datos, no basta con definir la forma en la que se van a guardar los datos en la memoria; es muy importante definir también las operaciones que se pueden hacer sobre los datos guardados. Esto nos lleva al concepto de Tipo de Dato Abstracto (ADT, por sus siglas en inglés).

En el capítulo 1 ahondaremos en el concepto de tipo de dato abstracto, podemos adelantar que es una entidad que consta de los datos que guarda y las operaciones que se pueden hacer sobre esos datos. Para las personas que ya han programado en un lenguaje orientado a objetos, se les hará muy familiar este conjunto. Un tipo de dato abstracto se implementa en un lenguaje orientado a objetos como una clase, en sus variables de instancia se define la estructura de datos física, o la información del nodo, y las operaciones que se realizan sobre los datos se implementan por medio de los métodos. La figura 7.3 muestra un esquema de un tipo de datos abstracto como una clase.



*Figura 7.3 Relación entre un ADT y una Clase: (a) ADT, (b) Clase.*

De ahora en adelante, siempre que hablemos de una estructura de datos, nos estaremos refiriendo a una ADT que engloba tanto los datos mismos, como las operaciones que se hacen sobre dichos datos. Este ADT, lo implementaremos en C++ como una clase. Se estudiarán los algoritmos para hacer el acceso a los datos muy eficiente y dichos algoritmos se implementarán en los métodos de la clase como las operaciones que se pueden hacer sobre los datos que guarda la estructura.



# Capítulo 8. Estructura de datos lineales

08

Las estructuras de datos se pueden clasificar de acuerdo al número de predecesores y sucesores que tiene sus elementos. Existen dos tipos de estructuras: lineales y no lineales. Una estructura de datos lineal es aquella en donde sus elementos tienen un predecesor, excepto el primero y el último, que no tienen.

Su nombre lo obtiene debido a que, si se colocan todos los datos que contiene la estructura, cada uno seguido de su sucesor, iniciando con el primero, se forma una estructura de una dimensión como la línea recta.

A continuación, estudiaremos a fondo este tipo de estructuras, las cuales son muy útiles para guardar y operar cierto tipo de datos en forma eficiente, y nos ayudarán a entender fácilmente cómo se implementa en código una estructura de datos.

## 8.1. Listas ligadas

Las listas ligadas o lista encadenadas se pueden ver como la generalización de los arreglos. Primero recordemos qué es un arreglo, en un arreglo los datos se encuentran uno seguido del otro en direcciones de memoria contiguas. Por ejemplo, si suponemos que en una dirección de memoria cabe un dato, y la memoria inicia en la dirección 0, entonces un arreglo de 5 elementos iniciando en la dirección 0, tendrá su primer dato precisamente en esa dirección de memoria; su segundo dato en la dirección 1 y así sucesivamente hasta su quinto dato que estará en la dirección 4.

Al darle un nombre a ese arreglo lo único que se requiere guardar en dicho nombre, es la dirección de memoria donde se encuentra el primer dato, como se puede observar en la figura 8.1.

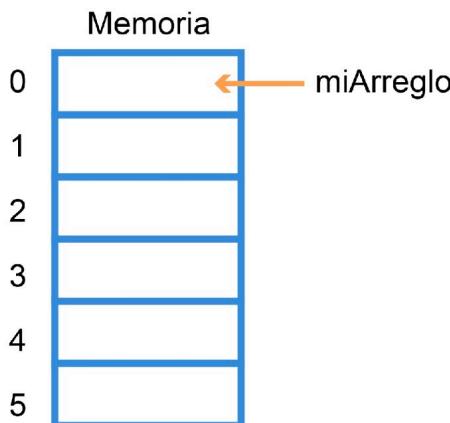


Figura 8.1. Ejemplo de un arreglo en memoria de 6 elementos.

Las listas ligadas son como un arreglo, pero sus datos no necesariamente están en direcciones de memoria contiguas. Sus datos están guardados en una entidad llamada nodo, el cual puede estar en cualquier dirección de memoria asignada por el sistema al solicitar espacio (memoria dinámica). Eso significa que cada nodo debe tener información adicional para poder localizar su nodo sucesor. Es decir, un nodo, además del dato, debe guardar la dirección de memoria en donde se encuentra su nodo sucesor. En muchos lenguajes de programación, como es el caso de C++, una dirección de memoria se guarda en una variable tipo apuntador (*pointer*), la cual se indica colocando un asterisco antes del nombre. Lo veremos en la codificación más adelante.

Como el último nodo no tiene sucesor, pero sí tiene una variable que guarda la dirección de su sucesor, es necesario que apunte a “nada”, que en muchos lenguajes de programación se

representa con la palabra “null”. A esta estructura también se le pone un nombre, como en el caso del arreglo, el cual guarda la dirección de su primer elemento, tal como se muestra en la figura 8.2.



Figura 8.2. Ejemplo de una lista ligada de 4 elementos.

En la figura 8.2 también aparecen los nodos uno detrás del otro, sin embargo, en la memoria pueden quedar en cualquier lugar, no necesariamente están en direcciones contiguas. Para efectos gráficos, siempre se acostumbra poner los elementos de una lista uno detrás de otro, cada uno de ellos apuntando a su sucesor, formando una línea.

### 8.1.1 Listas vs. arreglos

Una lista y un arreglo son muy similares. Se puede decir que el arreglo es un caso especial de una lista, donde los nodos se encuentran en direcciones contiguas de memoria.

Los dos tipos de estructuras de datos se pueden usar para cosas similares; sin embargo, ambas tienen ventajas y desventajas que debemos tomar en cuenta. A continuación, se comentan algunas de ellas.

El arreglo requiere guardar menos información porque no necesita guardar la dirección de los sucesores, ya que siempre están en la dirección siguiente a la actual. Esto los hace más rápidos a la hora de buscar datos en él, debido a que solo se requiere conocer el número de dato al que se desea acceder. Una simple multiplicación nos dará de inmediato la dirección de memoria en la que se encuentra el dato, partiendo de la dirección de memoria del dato inicial, la cual se conoce porque es la que está guardada

en la variable tipo arreglo.

- El arreglo requiere que se conozca el número de datos que guardará. Ciertamente, en algunos lenguajes de programación se puede extender en ejecución el tamaño del arreglo, pero esto implica crear otro arreglo y copiar todos los elementos del arreglo anterior en el nuevo, lo que lo hace muy tardado.
- Una lista ligada puede guardar tantos datos como la memoria lo permita ya que no se requiere especificar su longitud desde el inicio, cada vez que se requiere un dato, se crea y se inserta un nodo en ella. Desde luego, esto la hace más lenta.
- En ambas estructuras se pueden agregar o eliminar datos en cualquier posición de la misma; sin embargo, en algunos casos estas acciones son más rápidas en un arreglo y, en otro, son más rápidas en una lista. Analizaremos su orden con mayor detalle más adelante.

En resumen, es de suma importancia conocer el funcionamiento y las características de las listas y los arreglos, así como identificar la tarea para la que se está utilizando la estructura de datos, todo esto para determinar cuál se adapta mejor a las necesidades de la actividad en cuestión.

### **8.1.2 Definición e implementación de una lista ligada**

Es claro que para definir un ADT para una estructura de datos de tipo lista ligada, lo primero que se requiere es la definición de un ADT para un nodo, el cual contendrá dos variables de instancia: el del dato que guardará, de tipo genérico **T**, al cual llamaremos **info**, y el apuntador a su sucesor, el cual llamaremos **next**. En acciones solo requerirá un constructor que reciba el dato que guardará, que lo coloque dentro del nodo en la variable info y

haga **next** igual a **null**, como se muestra a continuación:

**NodoLista(T)**

**info: T**

**next: \*NodoLista(T)**

**Constructor:** Recibe el dato a ser guardado y regresa un apuntador a un nodo con el dato guardado en su atributo info, y el apuntador a su sucesor igual a nulo.

Si es necesario, se pueden agregar otros métodos para acceso y modificación de los miembros de la clase

Una vez que se cuenta con el ADT Nodo, es fácil crear un ADT Lista para definir una lista ligada. Este ADT contará solo con una variable de instancia llamada **primero**, la cual apuntará al primer nodo de la lista, que guardará la dirección en la memoria del primer nodo. Al inicio, la lista creada estará vacía, por lo que su constructor solo hará que la variable **primero** sea **nulo**. Posteriormente, se le podrán agregar más acciones como por ejemplo la de insertar o eliminar un cierto nodo de la lista. Su definición quedaría de la siguiente forma:

**Lista(T)**

**next: \*NodoLista(T)**

**Constructor:** Cuando se invoca, regresa un apuntador a una lista nula, es decir, con su variable **primero** apuntando a nulo.

Otros métodos de acceso y modificación de los miembros de la clase.

Ahora podemos hacer la codificación en lenguaje C++, basada en los ADT **Nodo** y **Lista**. Aunque la lista puede contener datos de cualquier tipo, para facilitar el ejemplo crearemos listas que

guardan valores enteros, de la siguiente forma:

```

1. ///
2. // Implementación de la clase NodoLista y Lista //
3. ///
4.
5. #include <iostream>
6.
7. using namespace std;
8.
9. class NodoLista{
10. public:
11. int info;
12. NodoLista *next;
13.
14. NodoLista(int dato){
15. info = dato;
16. next = nullptr;
17. }
18. };
19.
20. class Lista{
21. private:
22. NodoLista *primero;
23. public:
24. Lista(){
25. primero = nullptr;
26. }
27. };

```

- En la línea 11 se observa que el dato que contiene el nodo debe ser entero. Se hizo así para facilitar el ejemplo, pero puede ser del tipo que sea, incluyendo, un objeto de una clase.
- En la línea 12 está la definición de la variable **next** que guarda la dirección de su nodo sucesor. Desde luego, es un apuntador (se observa el `*` antes de la variable) a un **NodoLista**.
- En la línea 14 se encuentra la codificación del constructor, la cual recibe un dato y lo guarda en la variable **info** del **nodo** (línea 15). Por otro lado, hace nulo el apuntador al siguiente (línea 16).
- La clase lista (línea 20) solo tiene un apuntador a un nodo, el cual es el primer nodo de la lista y por eso se guarda en la variable **primero** (línea 22). Su constructor (línea 24) solo hace nulo el primero, lo cual hace que se cree una lista vacía.

### *8.1.3 Operaciones de la clase lista*

Ahora se deben definir las operaciones de la clase **Lista**, es decir, las acciones del **ADT Lista**. Estas operaciones son muy variadas y dependen del problema para el que se utilice la lista, sin embargo, existen algunas acciones básicas que deben hacer:

- Buscar un elemento.
- Insertar un nuevo elemento.
- Eliminar un elemento.
- Borrar la lista completa.

### *Buscar un elemento*

Para facilitar la explicación pensemos que el dato contenido en cada nodo de la lista es único, aunque fácilmente lo podemos generalizar a listas con datos repetidos. Tomemos como referencia la lista de la figura 8.3.



Figura 8.3. Ejemplo de una lista ligada de enteros con cuatro nodos.

Una lista, en general, no se encuentra ordenada (aunque podría estarlo), por lo que la búsqueda se tendrá que hacer en forma secuencial, iniciando con el primer elemento y moviéndose al siguiente hasta encontrar el elemento buscado o terminar la lista sin encontrarlo.

En la figura 8.3, se puede deducir que se tiene un objeto **miLista** de la clase **Lista** de la siguiente forma:

**Lista miLista;**

La variable de instancia **primero** del objeto **miLista** está apuntando al primer elemento de la lista, es decir, al nodo 1, para este ejemplo el que contiene el número 5. El nodo que contiene el 5

apunta al nodo que contiene el 2 y así sucesivamente. El último nodo apunta a **null**, indicando que ya no hay más nodos después de él, entonces es el fin de la lista.

Una vez analizado cómo está formada la lista, vemos que el algoritmo para buscar en la lista debe ir nodo por nodo, iniciamos en el primer nodo y nos movemos al siguiente hasta encontrar el nodo que contiene el dato buscado, o terminamos la lista sin haberlo encontrado. Si logramos encontrar el nodo que contiene el dato buscado, regresamos como valor su dirección en memoria, si no lo encontramos, regresamos **null**. Para ir marcando el nodo en el que estamos buscando actualmente, vale la pena generar una variable adicional de tipo **NodoLista**, que vaya guardando el apuntador al nodo actual. Realmente, el algoritmo es muy simple y se muestra a continuación:

**Entrada:** Busca sobre una lista (**primero** apunta al primer elemento de la lista) y recibe el **dato** que se quiere buscar.

**Salida:** Si encuentra el **dato**, regresa el apuntador al nodo que lo contiene, si no, regresa **null**.

1. Hacer **actual = primero**

2. Preguntar si el nodo actual es nulo.

2.1. Si la respuesta es verdadera regresar **null** (no se encontró el dato).

2.2. Si la respuesta es falsa, preguntar si el **dato** es igual a la **info** contenida en el nodo actual.

2.2.1. Si la respuesta es verdadera, regresar **actual**, que contiene el apuntador al nodo con el dato buscado (la

búsqueda tuvo éxito).

**2.2.2.** Si la respuesta es falsa, ir al siguiente nodo haciendo **actual = actual.next** y regresar al punto 2.

La codificación del algoritmo de búsqueda en listas en C++ es directa y se muestra en el siguiente código:

```

1. public:
2. NodoLista* buscar(int dato){
3. NodoLista *actual;
4. actual = primero;
5. while (actual != nullptr){
6. if (actual->info == dato)
7. return actual;
8. else
9. actual = actual->next;
10. }
11. return nullptr;

```

La variable **actual** debe ser un apuntador a un nodo, es decir, a un objeto de la clase **NodoLista** (línea 3). Debemos recordar que, en C++, si se tiene un apuntador a un objeto, el acceso a sus variables de instancia se obtiene por medio del operador flecha (**->**).

El orden de este proceso es O(n), debido a que, en el peor de los casos, es necesario buscar en toda la lista para encontrar un elemento o saber qué no se encuentra en la lista.

### *Insertar un nuevo elemento*

La inserción en una lista puede hacerse en cualquier lugar de la misma, esto es, al inicio, al final o en medio. Si se inserta al inicio o al final, la información que debe recibir es solo el dato que se desea insertar; si se inserta en medio, se le debe dar alguna referencia sobre en dónde se quiere insertar el dato que reciba. Esta referencia puede ser, por ejemplo, en el lugar 3 (el nuevo elemento se convertiría en el tercer nodo de la lista), pero también puede ser un nodo que contenga un dato específico, por ejemplo, insertar después del nodo que contenga el dato 5, o in-

sertar antes del nodo que tenga el dato 2. Esto implica que debemos tener no uno, sino cinco procedimientos para insertar un elemento a una lista:

1. Insertar al inicio.
2. Insertar al final.
3. Insertar después del nodo que contiene el dato k.
4. Insertar antes del nodo que tiene el dato k.
5. Insertar en el lugar n.

Vamos a tratar los casos del 1 al 3, que son los más comunes y dejaremos los dos finales como ejercicio. El proceso para insertar un nuevo nodo al inicio es el más simple de todos y se despliega de esta forma:

**Entrada:** recibe el **dato** que se va a insertar en la lista.

**Salida:** coloca el apuntador al nuevo primer elemento de la lista en la variable **primero**

1. Crear un nuevo nodo conteniendo el **dato** que se desea insertar y guardar su dirección en un apuntador llamado **nuevoNodo**.
2. Hacer que el **next** del **nuevoNodo** apunte al **primero** de la lista.
3. Hacer que el **primero** de la lista apunte al **nuevoNodo**.

Como ejemplo, tratemos de insertar un nodo que contiene un 1 al inicio de la lista de la figura 8.3. Los pasos del algoritmo se pueden observar en la figura 8.4.

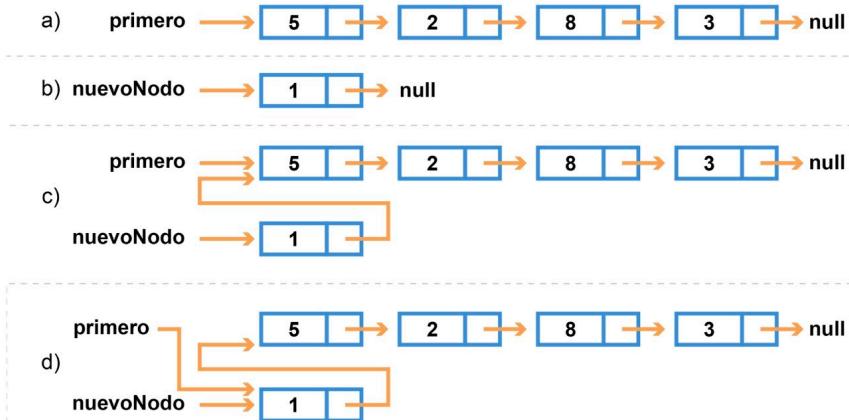


Figura 8.4. Ejemplo de inserción de un nuevo nodo al inicio de una lista.

La lista inicial se muestra en la figura 8.4a. El primer paso es generar un nuevo nodo con el dato que se desea insertar (figura 8.4b). El segundo paso es hacer que el **next** del nuevo nodo apunte al **primero** de la lista (figura 8.4c). Como paso final, el **primero** de la lista se hace ahora apuntar al nuevo nodo (figura 8.4d), convirtiéndose así en el nuevo primero de la lista. Desde luego que no pasa nada si la variable **nuevoNodo** se deja apuntando también al primero de la lista.

La implementación en lenguaje C++ se muestra a continuación:

```

1. public:
2. void insertarInicio(int dato){
3. NodoLista *nuevoNodo = new NodoLista(dato);
4. nuevoNodo->next = primero;
5. primero = nuevoNodo;
6. }

```

La inserción de un nuevo nodo al inicio de la lista tiene un orden constante **O(1)**, es decir, sin importar cuántos elementos contiene la lista, la inserción al inicio siempre tardará lo mismo.

Para insertar un nuevo nodo al final de la lista, las cosas

cambian en cuanto a la rapidez y facilidad, debido a que ahora tenemos que recorrer toda la lista para colocarnos en el nodo final y entonces poder insertar el nuevo. Su complejidad es **O(n)**, porque se tienen que recorrer todos los **n** elementos de la lista.

El algoritmo debe abarcar dos casos. Cuando la lista está vacía, el insertar un nuevo nodo bastará con hacer que **primero** apunte al nuevo nodo, pero cuando la lista tiene al menos un elemento, habrá que recorrerla primero para colocarse en el nodo final. Una vez que se está en el nodo final, la inserción es muy directa, ya que se hace que el **next** del nodo final apunte al nuevo nodo y listo (recuerda que el **next** del nuevo nodo ya apunta a **null** por lo que no habría que hacer nada más). El procedimiento completo se observa a continuación:

**Entrada:** recibe el **dato** que se va a insertar en la lista.

**Salida:** coloca el apuntador **next** del nodo final al nuevo nodo, convirtiéndose así en el nuevo nodo final.

1. Crear un nuevo nodo conteniendo el **dato** que se desea insertar y guardar su dirección en un apuntador llamado **nuevoNodo**.
2. Recorrer la lista desde el inicio, hasta colocarse en el nodo final.
3. Hacer que el **next** de nodo final apunte al **nuevoNodo**.

Un ejemplo gráfico se puede observar en la figura 8.5. La lista original está en la figura 8.5a, se crea un nuevo nodo en la figura 8.5b y se recorre la lista para llegar al nodo final y se hace que el **next** de dicho nodo final apunte al nuevo nodo en la figura 8.5c.

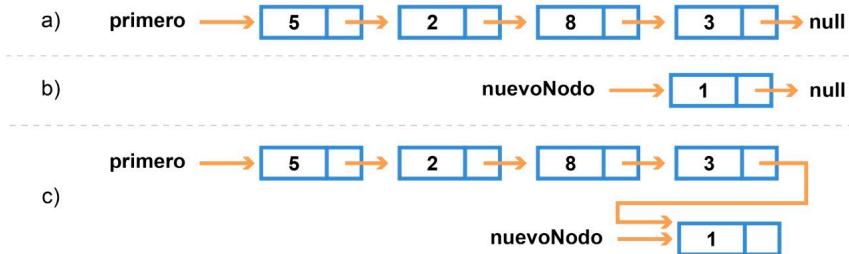


Figura 8.5. Ejemplo de inserción de un nuevo nodo al final de una lista.

Su implementación en código de C++ es:

```

1. public:
2. void insertarFinal(int dato){
3. NodoLista *nuevoNodo = new NodoLista(dato);
4. NodoLista *actual;
5. actual = primero;
6. if (primero == nullptr)
7. // la lista está vacía
8. primero = nuevoNodo;
9. else{
10. while (actual->next != nullptr)
11. actual = actual->next;
12. // al terminar, actual apunta al nodo final
13. actual->next = nuevoNodo;
14. }
15. }

```

La inserción en algún lugar intermedio de la lista es la más complicada de las 3. Como se comentó anteriormente, debemos dar alguna referencia, y en este caso, será el dato que contiene el nodo después del cual se quiere insertar el nuevo nodo.

El algoritmo consiste en recorrer la lista hasta encontrar el nodo que contiene el dato después del que se quiere insertar, llamémoslo nodo actual. El **next** del nuevo nodo lo hacemos apuntar al **next** del nodo actual. Finalmente, el **next** del nodo actual lo hacemos apuntar al nuevo nodo. Para facilitar el algoritmo supondremos que el nodo referencia siempre se encuentra en la lista, pero se puede modificar fácilmente en caso de que no lo encuentre. El proceso completo se muestra en el siguiente recuadro:

**Entrada:** recibe el **dato** que se va a insertar en la lista y el dato **k** que contiene el nodo después del que se quiere insertar.

**Salida:** inserta el nuevo nodo en el lugar solicitando, modificando los apuntadores correspondientes.

1. Crear un nuevo nodo conteniendo el **dato** que se desea insertar y guardar su dirección en un apuntador llamado **nuevoNodo**.
2. Recorrer la lista desde el inicio, hasta colocarse en el nodo que contiene el dato k al que llamaremos nodo actual.
3. Hacer que el **next** de nuevo nodo apunte al **next** del nodo actual.
4. Hacer que el **next** del nodo actual apunte al nuevo nodo.

Un ejemplo grafico se puede ver en la figura 8.6, en la que se inserta el nodo nuevo conteniendo un 1 después del nodo que contiene el dato 2.

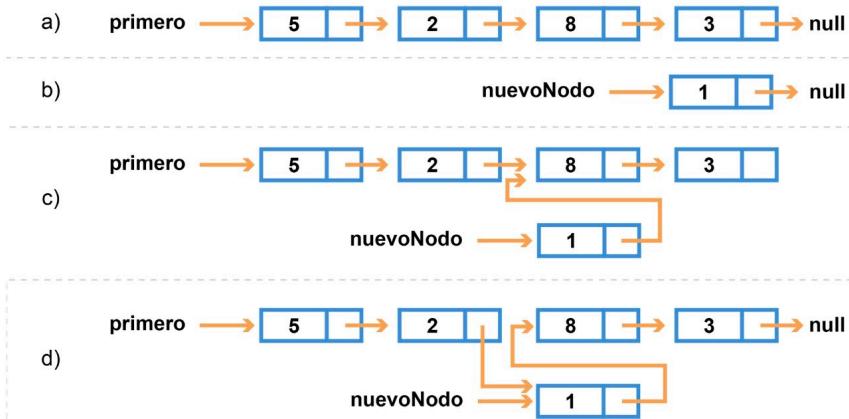


Figura 8.6. Ejemplo de inserción de un nuevo nodo en el medio de la lista.

La lista original se muestra en la figura 8.6a. Se aprecia cómo se crea un nuevo nodo con el dato 1 (figura 8.6b) y se recorre la lista hasta colocarse en el nodo que contiene un 2, al que llamaremos nodo actual; se logra que el **next** del nuevo nodo apunte al **next** del nodo actual en la figura 8.6c, es muy importante que este paso se haga antes de actualizar el **next** del nodo actual porque de otra forma se perdería el nodo al que apunta el nodo actual; finalmente, se hace que el **next** del nodo actual apunte al nodo nuevo en la figura 8.6d.

El código en C++ de este algoritmo es el siguiente:

```

1. public:
2. void insertarDespues(int dato, int k){
3. NodoLista *nuevoNodo = new NodoLista(dato);
4. NodoLista *actual;
5. actual = primero;
6. while (actual->info != k)
7. actual = actual->next;
8. // al terminar, actual apunta al nodo que contiene el valor k
9. nuevoNodo->next = actual->next; // muy importante que esto se haga primero
10. actual->next = nuevoNodo;
11. }

```

El orden de este proceso es igual al procedimiento de búsqueda, esto es **O(n)**, porque en el peor de los casos se debe recorrer toda

la lista para encontrar el lugar donde se desea insertar el nuevo nodo. Los dos casos restantes se dejan como ejercicio.

### *Eliminar un elemento*

El proceso para eliminar un elemento puede recibir el dato que se desea eliminar o el número de nodo de la lista. También se podría tener un proceso para eliminar el primer elemento de la lista y otro para eliminar el último elemento, que serían casos similares a los vistos para insertar un elemento.

La primera opción es la más común y es en la que nos centramos inicialmente para definir esta operación. Como es posible que el elemento que se desea eliminar no se encuentre en la lista, este procedimiento debe regresar una indicación de si se pudo eliminar o no. Normalmente es una bandera (booleano) que es verdadera si se pudo realizar y falsa en caso contrario.

El proceso es similar al que usamos para agregar un nodo después de un cierto elemento que ya se encontraba en la lista, explicado en el algoritmo 8.4. Si el nodo a eliminar no se encuentra en la lista, debe regresar una indicación de que el proceso falló (normalmente, false). Si el nodo sí se encuentra en la lista, debemos reconocer dos casos.

El primero es cuando el nodo a eliminar es el primero, para lo cual simplemente hacemos que el inicio de la lista (**primero**) apunte al siguiente (**next**) del primero. Si no es así, significa que el nodo a eliminar estará del segundo nodo en adelante (segundo caso), para lo cual recorremos la lista hasta que el siguiente del nodo actual sea el nodo para eliminar. Hacemos que el siguiente del actual apunte al siguiente de su siguiente y liberamos la memoria que ya no se utilice, en caso de que no haya recolector de basura. En este caso, debemos indicar que la operación fue

exitosa (normalmente, **true**). La operación completa es como se indica a continuación.

**Entrada:** dato que contiene el nodo a eliminar.

**Salida:** quita de la lista el nodo que contiene el dato, si es que existe en la lista.

1. Si el nodo a eliminar es el primero, hacer nodo eliminado igual a primero, hacer primero igual a **next** del nodo eliminado, ir a 5.
2. Si no es el primero, recorrer la lista hasta estar en el nodo cuyo nodo siguiente sea el que contiene el dato a eliminar. A ese nodo lo llamaremos nodo actual. En caso de terminar la lista y no haber encontrado el nodo a eliminar, regresar una indicación de que el proceso falló.
3. Hacer nodo eliminado igual al **next** del nodo actual.
4. Hacer **next** del nodo actual igual al **next** del nodo eliminado.
5. Liberar la memoria del nodo eliminado.
6. Regresar indicación de que la operación fue exitosa.

Un ejemplo de este proceso, cuando el nodo a eliminar sí está en la lista se muestra en la figura 8.7, en donde se borra el nodo que contiene el dato 2 de la lista.

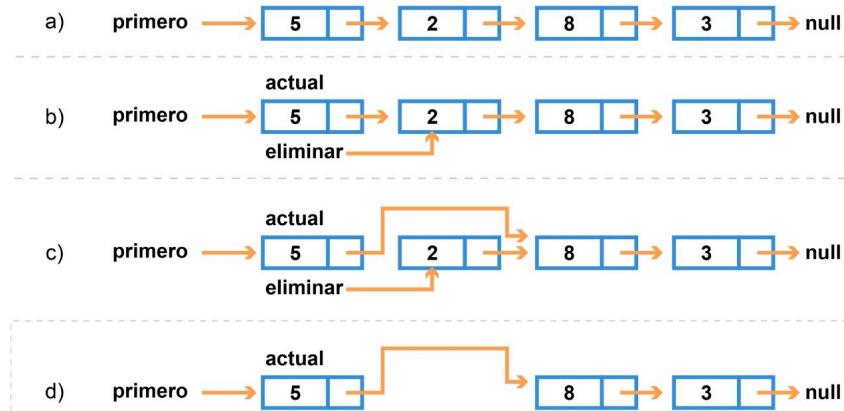


Figura 8.7. Eliminación de un nodo de la lista.

La figura 8.6a muestra la lista original. Se recorre la lista hasta encontrar el nodo actual, cuyo siguiente es el nodo a eliminar en la figura 8.6b; se hace el **next** del nodo actual igual al **next** del nodo a eliminar en la figura 8.6c y, finalmente, se libera la memoria del nodo eliminado en la figura 8.6d.

El código en lenguaje C++ del algoritmo de eliminación queda de la siguiente forma:

```

1. public:
2. bool eliminar(int dato){
3. NodoLista *actual, *elimina;
4. actual = primero;
5. if (actual == nullptr)
6. return false; // la lista está vacía
7. if (actual->info == dato){
8. primero = actual->next;
9. delete(actual);
10. return true; // era el primero y se borró con éxito
11. }
12. while ((actual->next != nullptr) && (actual->next)->info != dato)
13. actual = actual->next;
14. if (actual->next == nullptr)
15. return false; // el dato no se encuentra en la lista
16. elimina = actual->next;
17. actual->next = elimina->next;
18. delete(elimina);
19. return true; // el dato se borró con éxito
20. }

```

Recuerde que C++ no tiene recolector de basura, por lo que la

memoria se debe liberar a mano (líneas 9 y 18).

El orden de esta acción es similar al de búsqueda, es decir, **O(n)**, porque en el peor de los casos se debe recorrer toda la lista para encontrar el elemento que se desea eliminar o saber que no se encuentra.

La eliminación del primer nodo de la lista es una acción muy simple. Bastará con hacer el primero de la lista igual al **next** del **primero**, sin embargo, se debe indicar que la acción falló. Si la lista no está vacía, se debe retener el nodo a eliminar para poder liberar la memoria. El orden de este proceso es constante, **O(1)**. Su código en C++ puede quedar como sigue:

```

1. public:
2. bool eliminarInicio(){
3. NodoLista *elimina;
4. if (primero == nullptr)
5. return false; // la lista está vacía
6. else{ // lista no vacía
7. elimina = primero;
8. primero = primero->next;
9. delete(elimina);
10. return true; // la operación fue exitosa
11. }

```

La eliminación del último es muy simple, pero necesitamos recorrer toda la lista hasta colocarnos en el nodo anterior al último. Si la lista está vacía, regresamos una indicación de que la operación falló. Si la lista tiene al menos un elemento, necesitamos verificar dos casos: si la lista sólo cuenta con un elemento, solo tenemos que hacer **primero** igual a **nulo**, pero si tiene más de uno, tenemos que recorrer toda la lista hasta colocarnos en el penúltimo elemento. Retenemos el último con una variable auxiliar para luego liberar la memoria y finalmente hacemos el **next** del penúltimo igual a nulo. En este caso se debe regresar una indicación de que el proceso fue exitoso. El código es el siguiente:

```

1. public:
2. bool eliminarFinal(){
3. NodoLista *actual, *elimina;
4. actual = primero;
5. if (primero == nullptr)
6. return false; // la lista está vacía y el proceso falló
7. if (actual->next == nullptr){ // la lista sólo tiene un elemento
8. elimina = actual; // retenemos el nodo a eliminar
9. primero = nullptr;
10. }
11. else { // la lista tiene más de un elemento
12. while ((actual->next)->next != nullptr)
13. actual = actual->next; // la recorremos hasta el penúltimo
14. elimina = actual->next; // retenemos el nodo a eliminar
15. actual->next = nullptr;
16. }
17. delete(elimina);
18. return true; // el procedimiento fue exitoso

```

### Borrar la lista completa

En un lenguaje que tenga recolector de basura, bastará con borrar el apuntador al primero de la lista (primero), sin embargo, si el lenguaje no cuenta con recolector de basura, como es el caso de C++, se debe ir borrando nodo por nodo, desde el inicio para no dejar **leaks** en memoria.

El procedimiento completo consiste en recorrer la lista hasta el último nodo e ir borrando uno por uno. Sin embargo, es muy importante que antes de borrar un nodo guardemos su apuntador al siguiente (**next**) porque de otra forma se perdería la lista. El proceso completo se muestra a continuación:

**Entrada:** la lista a ser borrada (primero).

**Salida:** el **primero** de la lista apuntando a nulo, con la memoria liberada.

**1.** Mientras **primero** sea diferente de nulo:

**1.1.** Hacer **siguiente** igual al **next** de **primero**.

**1.2.** Liberar la memoria a la que apunta **primero**.

**1.3.** Hacer **primero** igual a **siguiente**.

El código de este algoritmo en lenguaje C++ queda como sigue:

```

1. public:
2. void borrar(){
3. NodoLista *siguiente;
4. while (primero != nullptr){
5. siguiente = primero->next;
6. delete(primero);
7. primero = siguiente;
8. }
9. }

```

El orden del borrado de la lista completa es **O(n)** cuando el lenguaje no tiene recolector de basura, como C++, ya que se tiene que eliminar nodo por nodo. Si el lenguaje cuenta con recolector de basura, como Java, el borrado de la lista tiene orden constante **O(1)**, porque bastará hacer que el primero deje de apuntar al primer elemento de la lista (hacerlo **nulo**) para que el recolector tome toda la memoria de la lista y la regrese al **heap**.

#### 8.1.4 Lista circular

La lista circular, como su nombre lo indica, es una lista en donde todos sus elementos están colocados en forma circular. Esto se logra fácilmente haciendo que el último elemento apunte al primero, como se muestra en la figura 8.7.



Figura 8.8. Un ejemplo de una lista circular.

Las listas circulares tienen la ventaja de que todos sus nodos son iguales en cuanto a predecesores y sucesores. En este caso, todos los nodos de una lista circular tienen un sucesor y un predecesor, ya no existe alguno que no tenga predecesor, como el primero de las listas lineales, y tampoco uno que solo tenga predecesor, como el último de las listas lineales. Aquí todos cumplen con la condición de tener un predecesor y un sucesor. Esta condición nos ayuda a manejar en forma más eficiente algunos tipos de acciones. Por ejemplo, al momento de insertar un nodo

no tenemos que hacer cuatro procedimientos diferentes, bastará con tener dos, uno para insertar antes de un nodo dado y otro para insertar después de un nodo dado.

Las listas circulares fueron creadas para resolver de forma más eficiente cierto tipo de problemas, por ejemplo, si se hacen búsquedas continuas de datos, no es necesario iniciar la búsqueda del dato desde el inicio para cada dato buscado, en su lugar, la búsqueda del siguiente dato puede iniciar donde se quedó la del dato anterior.

Sin embargo, este tipo de listas también tiene desventajas, por ejemplo, es más complicado saber cuándo ya se han recorrido todos los nodos de la lista en una búsqueda (recuerda que antes terminábamos cuando el **next** de un nodo apuntaba a **nulo**). Pero son cosas que se pueden solucionar fácilmente; una forma sería tener siempre un nodo como referencia que apuntará a algo que se le podría considerar el primer nodo (figura 8.8). De esta forma, cuando hayamos regresado a este nodo de referencia, significaría que ya terminamos de recorrer la lista. Es cierto que esta solución trae consigo algunas otras complicaciones, por ejemplo, qué pasa con ese nodo de referencia cuando es eliminado. Seguramente, ya se están imaginando alguna solución, la cual podría ser, poner como nodo de referencia el siguiente. ¿Y si la lista está vacía?, bueno pues el nodo de referencia apuntaría a nulo.



Figura 8.9. Lista circular con un nodo referencia llamado inicio.

Algunos programadores han optado por colocar un nodo de referencia fijo, el cual no contenga ningún tipo de información, por lo que nunca se podría borrar; de esa forma, la fila nunca estaría

vacía. Esa solución tiene la pequeña desventaja de requerir memoria aún cuando la lista esté vacía, pero solo es un nodo.

Las operaciones fundamentales en las listas circulares son las mismas que en las listas lineales, pero, como ya se comentó, no habría tantos casos especiales:

- 1.** Buscar un elemento.
- 2.** Insertar un elemento.
- 3.** Eliminar un elemento.
- 4.** Borrar lista completa.

### **8.1.5. Lista doblemente encadenada**

Por ejemplo, se ha podido observar que, al eliminar el último elemento de una lista, tengo que recorrerla toda, pero debo ir guardando el elemento anterior para asegurar que, cuando llegue al último elemento, en realidad esté colocado en el penúltimo. En otras palabras, para eliminar el último elemento de una lista siempre requiero colocarme en el penúltimo elemento para iniciar las operaciones de eliminación. Para hacer esto, es necesario tener dos apuntadores para este proceso: uno que apunte al actual y otro al siguiente del actual, en cada paso.

Esto se podría evitar logrando tener solo un apuntador para recorrer la lista, si al llegar al último elemento me pudiera regresar al elemento anterior. Con la definición actual del nodo de una lista ligada, esto es imposible ya que se tiene un apuntador al predecesor, solo se tiene uno al sucesor (**next**). Para poder hacerlo necesitamos modificar el nodo de una lista y agregar un apuntador al predecesor (**prev**). Su ADT se muestra a continuación:

**NodoListaDoble(T)****info:** T**prev:** \*NodoListaDoble(T)**next:** \*NodoListaDoble(T)

**Constructor:** Recibe el dato a ser guardado y regresa un apuntador a un nodo con el dato guardado en su atributo **info**, y los apuntadores a su sucesor y predecesor los hace nulos.

Si es necesario, se pueden agregar otros métodos para acceso y modificación de los miembros de la clase

A este nuevo tipo de listas se les llama listas doblemente encadenadas. Su ADT, basado en el del nodo anteriormente definido, es el siguiente:

**ListaDoble(T)****primero:** \*NodoListaDoble(T)

**Constructor:** Cuando se invoca, regresa un apuntador a una lista nula, es decir, con su variable **primero** apuntando a nulo.

Otros métodos de acceso y modificación de los miembros de la clase.

Una representación gráfica de una de estas listas se puede ver en la figura 8.9.



Figura 8.10. Ejemplo de una lista doblemente encadenada de 4 elementos.

Las operaciones sobre estas listas son las mismas para las listas ligadas simples, solo que ahora se deben modificar más apunadores.

Por ejemplo, para eliminar el último nodo del código es necesario recorrer toda la lista desde el inicio, hasta colocarnos en el nodo actual cuyo **next** sea **nulo**, que indicaría que es el último nodo. Estando ahí, hacemos un nuevo apuntador igual al **prev** del último y procedemos a eliminar el actual. En este caso el código sería:

```

1. public:
2. bool eliminarFinal(){
3. NodoListaDoble *actual, *penultimo;
4. actual = primero;
5. if (primero == nullptr)
6. return false; // la lista está vacía
7. while (actual->next != nullptr)
8. actual = actual->next;
9. // al salir del while actual apunta al último nodo
10. penultimo = actual->prev;
11. penultimo->next = nullptr;
12. delete(actual);
13. return true;
14. }

```

El resto de los códigos de las operaciones son similares a los de las listas simples y se dejan como ejercicio para el lector.

## 8.2. Pila

**L**a pila (**stack** en inglés) es otro tipo de estructura de datos lineal en el que se pueden insertar y borrar los datos que contiene. En la pila, los datos siempre se agregan por uno de sus lados, digamos por el inicio y se eliminan por el mismo lado. Esto es, los últimos elementos en entrar serían los primeros en ser eliminados, por lo que se le conoce como estructura **LIFO** (por sus siglas en inglés, **Last-In, First-Out**).

Un ejemplo real es una pila de platos. Los platos se colocan en la pila por arriba y al tomarlos, tomamos el que está hasta arriba. La figura 8.10 es una representación gráfica de una pila. Los datos contenidos pueden ser de cualquier tipo, pero para facilitar la explicación utilizaremos una pila de enteros.

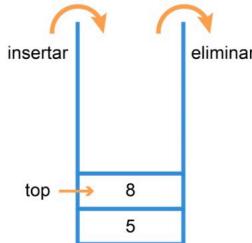


Figura 8.11. Ejemplo de una pila que contiene dos enteros.

La parte alta de la pila que contiene el elemento de hasta arriba, se le conoce como **top**. La pila tiene dos acciones principales, donde ambas modifican el **top** de la pila, además de su contenido:

- **push(dato)**: inserta el dato que recibe en la parte alta de la pila.
- **pop()**: elimina el elemento que está en la parte alta de la pila.

Tiene una tercera acción básica que no modifica ninguno de los valores de la pila:

- **getTop()**: regresa el valor que se encuentra en la parte alta de la pila.

En la figura 8.11 se muestra un ejemplo de varias acciones sobre una pila vacía. Las acciones están colocadas debajo de la pila y la pila muestra el resultado después de cada acción.

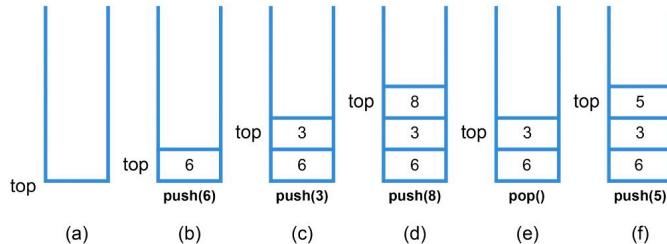


Figura 8.12. Ejemplo de una pila y algunas acciones

realizadas sobre ella.

En las acciones de la figura 8.11 se puede observar que cada **push** deja el dato en la parte alta de la pila, hace que se mueva el **top** un lugar hacia arriba; cada **pop** borra el elemento en el **top** de la pila y mueve el **top** un lugar hacia abajo. Si en la pila de la figura 8.11c hacemos un **getTop**, nos regresará un 3 y si lo hacemos en la figura 8.11f, no regresará un 5.

Tomando en cuenta lo anterior podemos diseñar el ADT Pila de la siguiente forma:

### Pila(T)

**top:** int

**espacioPila:** Una estructura de datos que contenga los datos de tipo T de una pila. Esta puede ser un arreglo o una lista ligada.

**Constructor:** Hace top = -1, para indicar que la lista está vacía.

**push(dato):** inserta el dato en la pila.

**pop():** saca un dato de la pila.

**getTop():** obtiene el elemento de la pila que esté en la parte superior.

Algunos otros métodos que se requieran.

El constructor puede inicializar la variable **top**, pero también se puede inicializar directamente en su definición de variable de instancia, con lo que no haría falta redefinir el constructor.

Es necesario contar con alguna estructura de datos, a la que llamaremos **espacioPila**, como lo indica el ADT, que nos permita guardar los datos que contiene la pila. Una posibilidad es usar un arreglo y otra es usar una lista ligada. Veamos primero el caso de un arreglo.

### 8.2.1 La pila implementada con un arreglo

Se puede utilizar un arreglo de tipo **T** para guardar los datos de una pila. Como se trata de un arreglo, es necesario definir su tamaño desde el inicio de la ejecución. A este tamaño lo llamaremos **MAX** e indica el número máximo de elementos que pueden estar almacenados en la pila.

Al utilizar un arreglo, la acción de insertar deberá verificar que la pila no esté llena; en caso de estar llena puede regresar una indicación de ello. Es una verificación similar a la que se debe hacer con la operación de sacar de la pila o de obtener el elemento superior de la pila, en estos casos, se debe verificar que la pila no esté vacía; en caso de estarlo, deberá regresar una indicación.

La indicación que regresa puede ser tan fácil como un booleano, si regresa verdadero es que sí se pudo realizar la acción, si regresa falso es que no se pudo. El programador deberá realizar ciertas cosas dependiendo del valor regresado por estas acciones. La verificación de pila llena o pila vacía se puede hacer por medio de dos procesos adicionales muy simples que sean parte del ADT a los que llamaremos **pilaLlena** y **pilaVacia**, ambos regresan verdadero cuando se cumple la condición y falso en caso contrario.

El procedimiento para insertar un elemento a una pila se puede describir como se indica en el siguiente recuadro.

**Entrada:** el dato que se desea insertar

**Salida:** el dato se coloca en la parte superior de la pila

**1.** Si la pila está llena, indicar que no se pudo insertar el dato (el dato se pierde)

**2.** Si la pila tiene espacio:

## 2.1. Incrementar el **top**

## 2.2. Colocar el dato en el lugar indicado por el **top**

## 2.3. Indicar que la operación se realizó correctamente

Veamos una implementación en código de lenguaje C++, para una pila que guarda datos enteros y tiene un tamaño máximo de 5 elementos, es decir MAX = 5. Podría quedar de la siguiente forma:

```
1. ///
2. // Implementación de la clase Pila usando un arreglo //
3. ///
4.
5. #include <iostream>
6. #define MAX 5
7.
8. using namespace std;
9.
10. class Pila{
11. private:
12. int top = -1;
13. int espacioPila[MAX];
14.
15. public:
16. bool push(int dato){
17. if (pilaLlena())
18. return false;
19. else {
20. top++;
21. espacioPila[top] = dato;
22. return true;
23. }
24. }
25.
```

```

26. bool pop(){
27. if (pilaVacia())
28. return false;
29. else {
30. top--;
31. return true;
32. }
33. }
34.
35. int getTop(){
36. return espacioPila[top];
37. }
38.
39. bool pilaLlena(){
40. if (top == MAX-1)
41. return true;
42. else return false;
43. }
44.
45. bool pilaVacia(){
46. if (top == -1)
47. return true;
48. else return false;
49. }
50. };

```

El tamaño máximo de la pila se define fuera de la clase con un **#define** (línea 6). El método **getTop()** (línea 35) no realiza ningún tipo de validación, esto implica que, cuando se utilice, el programador debe estar seguro que la pila tiene al menos un elemento, es decir, el programador debía llamarlo asegurando primero que el método **pilaVacia()** (línea 45) regresa falso.

Es importante notar que cuando se borra un elemento de la pila (línea 26), no se borra físicamente del arreglo, bastará con reducir el **top** en uno (línea 30) y eso deja disponible la casilla superior, a la cual se le podrá poner otro elemento con un **push**.

En esta implementación, tanto la acción de insertar (**push**) como la de borrar (**pop**) tienen un orden **O(1)**, al igual que la acción de obtener el elemento en el **top** de la pila.

### *8.2.2 La pila implementada con una lista ligada*

La pila también puede ser implementada con una lista ligada. El ADT **Pila** definido en la sección anterior se mantiene igual, pero sí tenemos que cambiar los algoritmos porque ahora la pila

no tiene un tamaño máximo (excepto por el tamaño de la memoria, lo cual no se verifica en esta implementación), siempre se podría insertar un elemento. La pila sí podría estar vacía, por lo que sí se tendrá que hacer esta verificación en algunos procesos, pero como los métodos de la lista ya tienen estas verificaciones, se pueden usar directamente y eso nos ahorra trabajo.

El procedimiento de inserción y el de borrado se hacen muy simples al usar las instrucciones ya definidas como métodos de la lista ligada, esto hace que los procesos en la pila se hagan sumamente simples y que el trabajo se le deje a lo que ya se implementó para la lista. Por ejemplo, el proceso de inserción (**push**) se codifica simplemente mandando llamar al procedimiento para insertar al inicio en una lista (**insertarInicio**). No se tiene que verificar nada porque todo ya se verificó en la implementación de la lista, así que **push** simplemente será un alias para **insertarInicio** en la lista, es decir, son los mismos métodos. Lo mismo sucede para borrar (**pop**), que es el mismo procedimiento para eliminarInicio en la lista.

Sin embargo, el método **getTop** de la pila, no tiene equivalente en la lista y lo tendríamos que agregar. Se trata de un simple **getter**, método que obtiene el valor de la variable, para la variable **info** del primer nodo de la lista, al que llamaremos **getPrimero**. El método se debe agregar a la clase Lista y en C++ será:

```
1. public:
2. int getPrimero(){
3. return primero->info;
4. }
```

Solo faltaría el método **pilaVacia** (**pilaLlena** no se requiere). La pila está vacía cuando el primero de la lista apunta a **nulo**, por lo que el método sería booleano y simplemente regresa **true** si la pila está vacía y **false** cuando esté llena, por lo que se tendría que agregar a la lista el método que nos diga si está vacía, al que lla-

maremos **listaVacia** y se codifica en C++ como sigue:

```

1. public:
2. bool listaVacia(){
3. return(primer == nullptr);
4. }

```

Desde luego que el mismo código se puede hacer con un **if**, pero la forma presentada es más compacta y funciona igual. En la línea 3 se compara primero con **nulo**, si son iguales, el resultado es verdadero y es lo que se regresa, si no son iguales, el resultado de la comparación es falso y es lo que se envía.

Finalmente, y antes de pasar al código completo, debemos recordar que para la implementación de la pila con una lista ligada vamos a requerir utilizar la clase **Lista definida** en la sección 8.1, con las modificaciones que ya se comentaron. El código completo es el siguiente:

```

1. //
2. // Implementación de la clase Pila usando una lista ligada //
3. //
4.
5. #include <iostream>
6. #include "Lista.h"
7.
8. using namespace std;
9.
10. class PilaLista{
11. private:
12. Lista espacioPila;
13.
14. public:
15. void push(int dato){
16. espacioPila.insertarInicio(dato);
17. }
18.
19. bool pop(){
20. return espacioPila.eliminarInicio();
21. }
22.
23. int getTop(){
24. return espacioPila.getPrimero();
25. }
26.
27. bool pilaVacia(){
28. return espacioPila.listaVacia();
29. }

```

En el código se puede observar que todos los métodos de la pila, cuando se usa una lista, son iguales a alguno de la lista, es decir, son envolturas.

### 8.3. Cola

**U**na cola o fila (**queue**, en inglés), es una estructura de datos lineal en la que sus elementos entran por un lado de la estructura y se eliminan o salen por el lado contrario. La cola puede contener datos de cualquier tipo, pero, por facilidad en la explicación, nos vamos a concentrar en una cola de enteros. La figura 8.12 muestra un ejemplo gráfico de una cola de enteros de cuatro elementos.

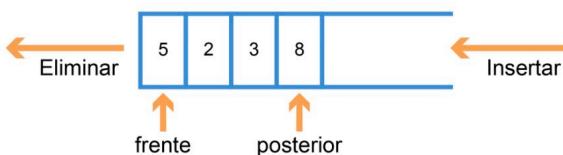


Figura 8.13. Ejemplo de una cola de cuatro elementos.

El último elemento que ha entrado a la cola se le conoce como **posterior (rear)** y el siguiente elemento que saldría de la cola se le conoce como **frente (front)**. Ambos están indicados en la misma figura 8.12.

Una cola es una estructura de tipo **FIFO** (**First-In, First-Out**), porque el primer elemento que entró en la cola es el primero en salir. Un ejemplo de este tipo de estructura es una fila para pagar en un supermercado, en donde los primeros clientes que llegan a la fila son los primeros en ser atendidos por el cajero para hacer el pago.

La cola tiene dos acciones principales:

- **enqueue(dato)**: inserta el dato recibido en la cola.
- **dequeue()**: saca un dato de la cola.

Se pueden agregar más acciones dependiendo de lo que sea necesario en el problema que se resolverá. Una de las más comunes

es:

- **getFrente()**: regresa el elemento que se encuentra en el frente de la cola ya que, comúnmente, se desea utilizarlo antes de eliminarlo.

Otra de las operaciones que nos ayudaría mucho es la que verifica si la cola está vacía o llena, cuando la cola es de tamaño fijo:

- **colaVacia()**: regresa una indicación cuando la cola está vacía. Generalmente, un booleano, verdadero si está vacía y falso en caso contrario.
- **colaLlena()**: regresa una indicación cuando la cola está llena. Generalmente, un booleano, verdadero si está vacía y falso en caso contrario.

En la figura 8.13 se muestra un ejemplo de varias acciones sobre una cola vacía. Las acciones están colocadas debajo de la cola y la cola muestra el resultado después de cada acción.

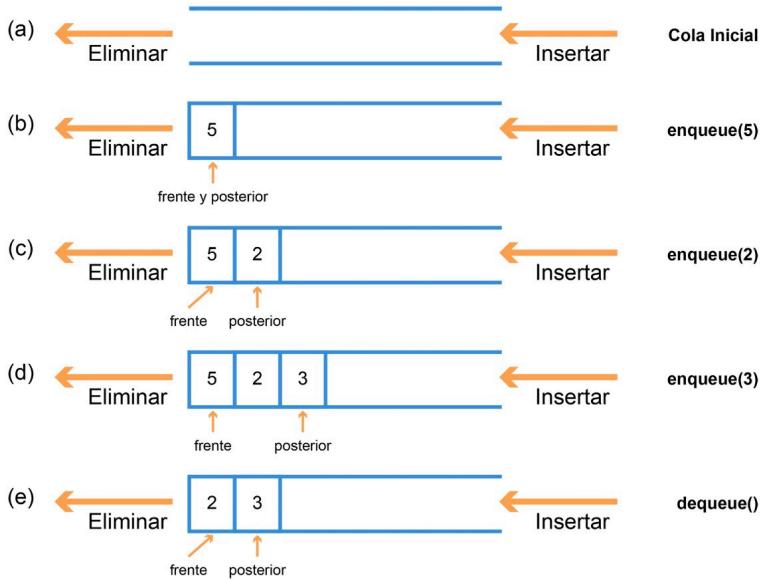


Figura 8.14. Ejemplo de una cola y algunas acciones realizadas sobre ella.

Si en la figura 8.13d, llamamos a la acción **getFrente()**, el resultado es un 5. En la figura 8.13e, el valor que se eliminó fue precisamente el 5 que se encontraba en el frente. Ahora se puede definir el ADT Cola como:

### Cola(T)

**frente:** \*NodoListaDoble

**posterior:** \*NodoListaDoble

**espacioCola:** Una estructura de datos que contenga los datos de tipo T de una cola. Esta puede ser un arreglo o una lista ligada.

**Constructor:** Hace frente y posterior iguales a nulo, para indicar que la lista está vacía.

**enqueue(data):** inserta el dato en la cola.

**dequeue():** saca un dato de la cola.

**getFrente()**: obtiene el dato de la cola que esté en **frente**.  
**colaVacia()**: regresa verdadero si la cola está vacía.  
Algunos otros métodos que se requieran.

### 8.3.1 Implementación de una cola utilizando un arreglo

Al igual que una pila, la cola puede ser implementada utilizando un arreglo o una lista ligada para guardar los datos de la estructura en **espacioCola**. Sin embargo, cuando se utiliza un arreglo, la implementación se complica un poco debido a que los dos indicadores, frente y posterior, cada vez que se realiza una operación de eliminar o insertar, se deben recorrer a la derecha de la cola, dejando que el lado izquierdo del arreglo se quede sin utilizar.

Además, como el tamaño de la cola es fijo (**MAX**) se debe verificar que la cola no esté llena. La figura 8.14 muestra un ejemplo de este problema.

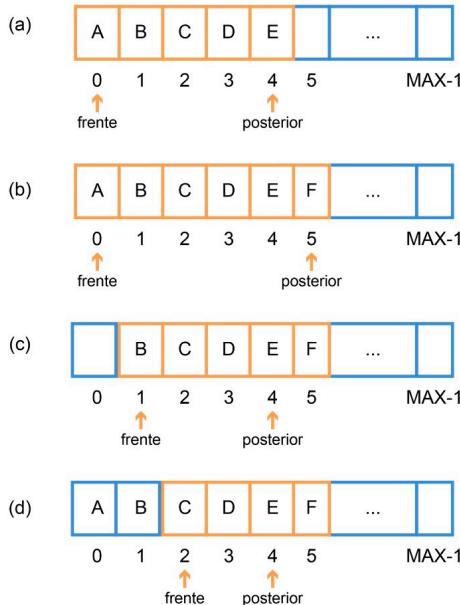


Figura 8.15. Problema al usar un arreglo para implementar una cola.

En la figura 8.14a, se tiene una cola inicial con algunos elementos. En la figura 8.14b, se observa que se insertó el elemento F, lo cual cambia el apuntador al **posterior**, recorriéndolo un lugar a la derecha del arreglo. En la figura 8.14c, se saca el elemento A de la cola, cambiando el apuntador a frente, recorriéndolo un lugar a la derecha de la cola. Lo mismo pasa en la figura 8.14d; ahora el apuntador frente se recorre un lugar más a la derecha. Si se continúa con estas operaciones llegará un momento en el apuntador a posterior que se encuentre en el final del arreglo, indicando que ya no hay más espacio cuando en realidad hay espacio a la izquierda del mismo.

Una forma de evitar este problema es usar el arreglo como si fuera circular, es decir, que el siguiente del lugar más a la derecha sea el lugar más a la izquierda, como se muestra en la figura 8.15.

Esto se logra fácilmente utilizando el módulo del índice sobre el tamaño máximo de la cola.

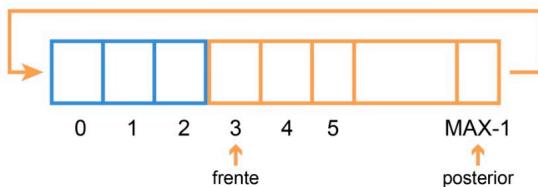


Figura 8.16. Una cola circular.

Sin embargo, se presentan todavía algunos otros problemas como son el determinar usando frente y posterior, cuando la cola está llena o vacía, o con qué valores los debemos inicializar en una cola circular. Todo esto se puede resolver agregando alguna información a la estructura, como el número de elementos que contiene y algunas otras cosas, sin embargo, la implementación utilizando listas ligadas parece ser una mejor opción.

### *8.3.2 Implementación de una cola usando una lista ligada*

Tenemos que utilizar la clase **Lista** para definir la clase **Cola**. De esta forma, todas las operaciones que se requieren para la cola ya están definidas para una lista por lo que solo se tiene que definir con otro nombre, es decir, hacer una envoltura (**wrap**) para ellas, de la siguiente forma:

- La operación de insertar un elemento en la cola es igual a la operación de insertar un nodo al final de la lista.
- La operación de sacar un elemento de la cola es igual a la operación de eliminar un nodo del inicio de la lista.
- La operación para obtener el valor del elemento a ser eliminado es igual a la operación para obtener el valor del primero

de la lista.

- La operación para saber si la cola está vacía es igual a la operación para saber si la lista está vacía.

El código completo de esta implementación en C++ es el siguiente:

```

1. ///
2. // Implementación de la clase Cola usando una lista ligada //
3. ///
4.
5. #include <iostream>
6. #include "Lista.h"
7.
8. using namespace std;
9.
10. class ColaLista{
11. private:
12. Lista espacioCola;
13.
14. public:
15. void enqueue(int dato){
16. espacioCola.insertarFinal(dato);
17. }
18.
19. bool dequeue(){
20. return espacioCola.eliminarInicio();
21. }
22.
23. int getFrente(){
24. return espacioCola.getPrimero();
25. }
26.
27. bool colaVacia(){
28. return espacioCola.listaVacia();
29. }
30. };

```

## 8.4. Deque

**U**na deque es una estructura de datos lineal que generaliza las filas, ya que se pueden insertar y eliminar elementos por cualquiera de sus dos extremos. Es una estructura muy versátil, con la que se pueden implementar fácilmente otros tipos de estructuras de datos como pilas y colas. Su nombre es un acrónimo de “cola doblemente terminada” en inglés (**double ended queue**).

La forma más sencilla para su implementación es con una lista

dblemente encadenada, pero con un inicio en el primer nodo de la lista y otro en el último nodo de la lista. No se tiene ninguna distinción entre cuál de los dos extremos es el inicio y el fin de la cola y solo lo vamos a establecer por conveniencia al momento de la implementación. A los dos inicios los llamaremos, como en el caso de las colas simples, frente (**front**) y posterior (**rear**). La figura 8.16 es una representación gráfica de un deque de 4 elementos.



Figura 8.17. Ejemplo de una lista doblemente encadenada de 4 elementos.

#### 8.4.1 Definición e implementación de un deque

Los procedimientos de estas operaciones son similares a los de las listas doblemente encadenadas (sección 8.1.5) debido a que los nodos para un **deque** son iguales a los de las listas doblemente encadenadas, es decir, cada nodo cuenta con dos apuntadores, uno a su nodo previo (**prev**) y otro a su nodo sucesor (**next**), entonces, al insertar o eliminar un elemento se deben actualizar los dos apuntadores de los nodos adyacentes. A continuación, se muestra su ADT tanto para su nodo como para el **deque**:

**NodoListaDoble(T)**

**info:** T

**prev:** \*NodoListaDoble(T)

**next:** \*NodoListaDoble(T)

**Constructor:** Recibe el dato a ser guardado y regresa un apuntador a un nodo con el dato guardado en su atributo **info**, y el apuntador a su sucesor igual a nulo.

Si es necesario, se pueden agregar otros métodos para acceso y modificación de los miembros de la clase

### Deque(T)

**frente:** \*NodoListaDoble(T)

**posterior:** \*NodoListaDoble(T)

**Constructor:** Cuando se invoca, regresa un apuntador a una lista nula, es decir, con sus variables **frente** y **posterior** apuntando a nulo.

Otros métodos de acceso y modificación de los miembros de la clase.

La implementación en una clase en código para el lenguaje C++ es:

```

1. //
2. // Implementación de la clase NodoListaDoble y Deque //
3. //
4.
5. #include <iostream>
6.
7. using namespace std;
8.
9. class NodoListaDoble{
10. public:
11. int info;
12. NodoListaDoble *next;
13. NodoListaDoble *prev;
14.
15. NodoListaDoble(int dato){
16. info = dato;
17. next = nullptr;
18. prev = nullptr;
19. }
20. };
21.
22. class Deque{
23. private:
24. NodoListaDoble *frente;
25. NodoListaDoble *posterior;
26. public:
27. Deque(){
28. frente = nullptr;
29. posterior = nullptr;
30. }
31. };

```

## 8.4.2 Operaciones básicas en un deque

Las operaciones básicas sobre estas colas son:

- **insertarFrente(dato)**: insertar un elemento al frente.
- **insertarPosterior(dato)**: insertar un elemento en la parte posterior.
- **eliminarFrente()**: eliminar un elemento del frente.
- **eliminarPosterior()**: eliminar un elemento de la parte posterior.

Algunas operaciones auxiliares que pueden ser de gran utilidad son:

- **getFrente()**: obtener el valor del elemento al frente.
- **getPosterior()**: obtener el valor del elemento en la parte posterior.
- **dequeVacia()**: saber si la lista está vacía.

Veámoslas una a una, con su procedimiento e implementación.

#### *Insertar un elemento al frente*

La operación de insertar un nodo al frente del deque **insertarFrente(dato)** es la misma que se usa en las listas doblemente encadenadas para insertar al inicio. Se deben tomar en cuenta dos casos:

- Cuando la **deque** está vacía: se genera el nodo nuevo con el dato a ser insertado, se hace que **frente** y **posterior** apunten a dicho nodo y listo.
- Cuando la **deque** no está vacía:
  - Se genera el nuevo nodo con el dato.
  - El **next** de ese nuevo nodo debe apuntar al frente.

- El **prev** del nodo al que apunta el frente debe apuntar al nuevo nodo.
- El **frente** debe apuntar al nuevo nodo.

En ambos casos, el primer paso es crear el nodo con el dato que se recibe como entrada, con la instrucción **new** y el constructor. La implementación completa en C++ es:

```

1. public:
2. void insertarFrente(int dato){
3. NodoListaDoble *nuevoNodo = new NodoListaDoble(dato);
4. if (frente == nullptr){ // deque vacía
5. frente = nuevoNodo;
6. posterior = nuevoNodo;
7. }
8. else { // deque no está vacía
9. nuevoNodo->next = frente;
10. frente->prev = nuevoNodo;
11. frente = nuevoNodo;
12. }
13. }

```

Esta operación es de orden constante **O(1)**, independientemente del tamaño de la **deque**.

### *Insertar un elemento en la parte posterior*

La operación de insertar un elemento en la parte posterior del deque **insertarPosterior()** es mucho más simple que la utilizada para las listas doblemente encadenadas, debido a que ahora tenemos un apuntador al final de la lista (**posterior**) y entonces no tenemos que recorrer toda la lista para colocarnos en el último nodo. De hecho, esta operación es prácticamente la misma que la de **insertarInicio()** pero ahora, todo lo que se hacía con frente se hace con posterior. Lo puede comprobar en el siguiente código en C++:

```

1. public:
2. void insertarPosterior(int dato){
3. NodoListaDoble *nuevoNodo = new NodoListaDoble(dato);
4. if (posterior == nullptr){ // deque vacía
5. frente = nuevoNodo;
6. posterior = nuevoNodo;
7. }
8. else { // deque no está vacía
9. nuevoNodo->prev = posterior;
10. posterior->next = nuevoNodo;
11. posterior = nuevoNodo;
12. }
13. }

```

El orden de esta operación es de orden constante **O(1)**, independientemente del tamaño de la deque, en contraste con la misma operación para listas doblemente encadenadas donde era **O(n)**.

### *Eliminar un elemento del frente*

El proceso de eliminar un elemento del frente de la deque **eliminarFrente()** es igual al de eliminar un elemento del frente de la lista doblemente encadenada. En todo proceso de eliminar un nodo de una lista, puede darse el caso de que la estructura esté vacía, en cuyo caso se debe indicar que el proceso falló. Por esta razón, se acostumbra que los procedimientos de eliminar regresen un booleano; si es falso significa que el proceso falló y si es verdadero, significa que el proceso tuvo éxito. Ahora se deben considerar tres casos:

- La **deque** está vacía: se regresa un indicador de que el proceso falló. Generalmente es un booleano falso.
- La **deque** solo tiene un elemento: tanto frente como posterior se hacen nulos y se regrese un indicador de que el proceso fue exitoso.
- La **deque** tiene más de un elemento:
  - Se retiene el nodo a eliminar haciendo **eliminar** igual al **frente**.
  - El **prev** del **frente** debe apuntar a **nulo** puesto que ya no

tiene nodo previo.

- Se hace que **frente** apunte al **next** del **frente**.
- Se regresa una indicación de que el proceso fue exitoso.

La implementación de este proceso en C++ es la siguiente:

```

1. public:
2. bool eliminarFrente(){
3. NodoListaDoble *elimina;
4. if (frente == nullptr) // deque vacía
5. return false; // la operación falló
6. if (frente == posterior){ // deque sólo tiene un elemento
7. elimina = frente;
8. frente = nullptr;
9. posterior = nullptr; // la deque queda vacía
10. delete(elimina);
11. return true; // la operación fue exitosa
12. }
13. // deque tiene más de un elemento
14. elimina = frente; // retener el nodo a eliminar
15. frente = frente->next;
16. frente->prev = nullptr;
17. delete(elimina);
18. return true; // la operación fue exitosa
19. }

```

El orden de este proceso es también constante **O(1)**, independientemente del tamaño del deque.

### *Eliminar un elemento en la parte posterior*

Este procedimiento es el mismo que para eliminar un elemento en el frente, solo que se cambia el frente por el posterior y el **next** por el **prev**, como se observa en el siguiente código en C++:

```

1. public:
2. bool eliminarPosterior(){
3. NodoListaDoble *elimina;
4. if (posterior == nullptr) // deque vacía
5. return false; // la operación falló
6. if (frente == posterior){ // deque sólo tiene un elemento
7. elimina = posterior;
8. frente = nullptr;
9. posterior = nullptr; // la deque queda vacía
10. delete(elimina);
11. return true; // la operación fue exitosa
12. }
13. // deque tiene más de un elemento
14. elimina = posterior; // retener el nodo a eliminar
15. posterior = posterior->prev;
16. posterior->next = nullptr;
17. delete(elimina);
18. return true; // la operación fue exitosa
19. }

```

De la misma forma, este proceso también tiene un orden cons-

tante **O(1)**, independientemente del tamaño del **deque**, gracias a que se tiene el apuntador al final posterior.

### *Operaciones adicionales*

Las operaciones adicionales se implementan fácilmente si no se tiene ningún tipo de validación para los **getters**. El usuario debe verificar que las condiciones del deque permitan utilizarlos. Su código completo en C++ se presenta a continuación:

```

1. public:
2. int getFrente(){
3. return frente->info;
4. }
5.
6. int getPosterior(){
7. return posterior->info;
8. }
9.
10. bool dequeVacia(){
11. return (frente == posterior);
12. }

```

### *8.4.3 Usando una deque para implementar otras estructuras de datos*

Con las acciones implementadas para que la **deque** se pueda utilizar, esta para implementar otra estructura de datos lineales como la **Pila** y la **Cola**. En ambos casos, sus procedimientos se convierten en envolturas para algunos de los procedimientos de la **Deque**.

Para el caso de la **Pila**, tomando como sitio de entrada y salida el frente del **deque**, tenemos que:

- **push(dato)** es igual a **insertarFrente(dato)**
- **pop()** es igual a **eliminarFrente()**
- **getTop()** es igual a **getFrente()**
- **pilaVacia()** es igual a **dequeVacia()**

También es posible que el sitio de entrada y salida de la es-

tructura como el posterior, en cuyo caso se usarían insertar y eliminar del lado posterior, es decir, **insertarPosterior(dato)** y **eliminarPosterior()**, respectivamente.

Para el caso de la **Cola**, considerando el sitio de entrada el posterior y el sitio de salida el frente, tenemos que:

- **enqueue(dato)** es igual a **insertaPosterior(dato)**
- **dequeue()** es igual a **eliminaFrente()**
- **getFrente()** es igual a **getFrente()**
- **colaVacia()** es igual a **dequeVacia()**

De la misma forma que para la **pila**, para la **cola** también se puede considerar el lado de entrada como el frente y el lado de salida como el posterior, lo que implicaría cambiar los métodos usados del **deque**, pero el resultado sería el mismo.

## Ejercicios



**1.** Escriba la pila que resulta en cada paso al hacer las siguientes operaciones en el orden establecido, partiendo de una pila de enteros vacía:

**push(9), push(2), pop(), push(4), push(7), pop(), pop().**

**2.** Para el ejercicio 1, escriba lo que regresaría el método getTop() si se invoca después del método:

**a) push(4)**

**b) pop()**

**3.** Escriba la lista ligada que resulta en cada paso al hacer las siguientes operaciones en el orden establecido, partiendo de una cola de enteros vacía:

**insertarInicio(6), insertarInicio(3), insertarFinal(4), insertarDespues(3,5), eliminar(6), eliminarFinal(), insertarAntes(6,1), eliminarInicio()**

**4.** Para el ejercicio 3, escriba lo que regresaría el método getPrimer() si se invoca después del método:

**a) insertarFinal(4)**

**b) eliminarInicio()**

**c) insertarDespues(3,5)**

5. Escriba la cola que resulta en cada paso al hacer las siguientes operaciones en el orden establecido, partiendo de una cola de enteros vacía:

**insertar(0), insertar(5), insertar(2), eliminar(),  
eliminar(), insertar(0), insertar(7)**

6. Para el ejercicio 5, escriba lo que regresaría los métodos getFrente() y getPosterior() si se invocan después del método:

a) El segundo insertar(0)

b) El primer eliminar()

c) insertar(7)

7. Las librerías de C++ ya tienen implementadas todas las estructuras de datos vistas en este capítulo, tal vez con otros nombres. Investigue en la documentación de C++:

a) ¿Cuáles son los nombres de todas las estructuras?

b) ¿Cuáles son los nombres de los métodos para una pila?

c) ¿Cómo se utilizan en un programa?

Ejercita ahora tu programación para implementar en código:

1. Una pila utilizando un deque.

2. Una cola utilizando un deque.

3. Una lista ligada utilizando un deque.

4. Una lista doblemente encadenada utilizando un deque.

5. Una pila con una lista ligada, pero sin utilizar la clase Lista.
6. Los dos métodos de inserción que faltaron para la clase Lista (sección 8.1.3).
7. Utilice la librería que C++ tiene para la pila para implementar un programa que pruebe el resultado que obtuvo en los ejercicios 1 y 2.
8. Utilice la librería que C++ tiene para la cola para implementar un programa que pruebe el resultado que obtuvo en los ejercicios 5 y 6.
9. Utilice la librería que C++ tiene para el deque, para implementar:
  - a) Una pila
  - b) Una cola



# Capítulo 9. Estructuras de datos jerárquicas

09

**R**ecordemos que las estructuras de datos se pueden clasificar en dos tipos: lineales y no lineales. Una estructura de datos lineal es aquella en donde los datos tienen un predecesor (excepto el primero) y un sucesor (excepto el último). Tal es el caso de los arreglos y las listas ligadas. Las estructuras de datos no lineales son aquellas en donde cada dato puede tener más de un predecesor y más de un sucesor.

Las estructuras de datos jerárquicas son un tipo especial de estructura de datos no lineal, en las que cada dato tiene exactamente un predecesor, excepto el primero, que no tiene predecesores. El número de sucesores de cada dato no está limitado, pudiendo ir desde 0 hasta un número n dado.

Su nombre viene del hecho de que, si se observa gráficamente cada uno de los datos, se pone a los predecesores hasta arriba, se inicia con el primero, y sus sucesores quedan hacia abajo; esto forma una especie de organigrama con todos los elementos colocados por niveles, lo cual da la idea de cierta jerarquía entre los datos.

A continuación, estudiaremos a fondo este tipo de estructuras, las cuales son muy útiles para guardar y operar cierto tipo de datos en forma eficiente.

## 9.1 Árboles

**U**n nombre más común para las estructuras de datos jerárquicas es el de árboles. Se llama árbol porque, si se observa gráficamente, además de parecer un organigrama, también da la idea de ser un árbol genealógico, o un árbol real colocado al revés; es decir, con la raíz en la parte de arriba y sus ramas se extienden hacia abajo. En la figura 9.1 se tiene un ejemplo gráfico de un árbol.

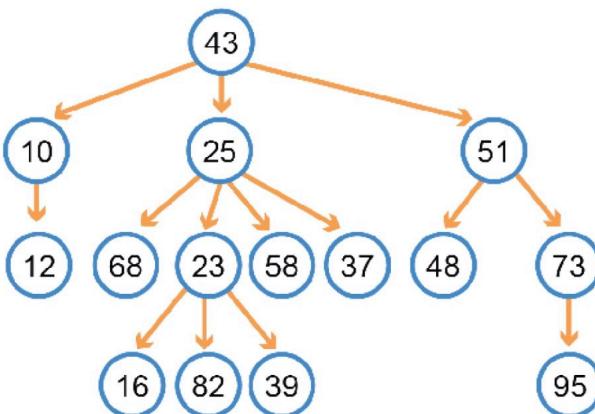


Figura 9.1 Ejemplo gráfico de un árbol.

Cuando hay una estructura que representa un árbol, se sabe de inmediato que todas las flechas en su representación gráfica, están apuntando de los elementos hacia los sucesores (normalmente, hacia abajo), por lo que cuando se dibuja un árbol no es necesario colocar las flechas. De esta forma, la representación gráfica del árbol de la figura 9.1 se hace sin flechas como se muestra en la figura 9.2.

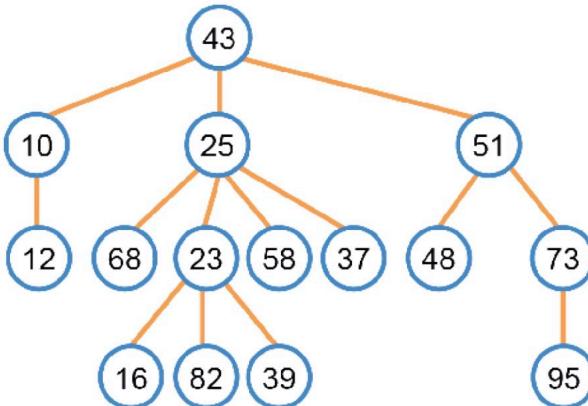


Figura 9.2. Representación gráfica de un árbol sin flechas.

### 9.1.1 Notación de árboles

Los árboles tienen una notación especial y a sus elementos se les suele llamar elementos de un árbol real. De esta forma tenemos las siguientes definiciones:

- **Nodo:** es cada uno de los elementos de un árbol. En su representación gráfica, se representan con un círculo (figura 9.2) o un óvalo (figura 9.3).
- **Arco:** también llamado arista, es la línea que une un nodo de un árbol con su sucesor, en realidad es una flecha, pero se dibuja como una línea.
- **Raíz:** es el primer nodo de un árbol, predecesor de todos los demás. Es el único nodo que no tiene predecesor. Normalmente, se coloca en la parte superior de su representación gráfica (el nodo 43 en la figura 9.2)
- **Hoja:** es un nodo que no tiene sucesores (por ejemplo, los

nodos 82 y 95 de la figura 9.2). También se le conoce como nodo terminal.

- **Nodo intermedio:** es un nodo que tiene al menos un hijo (por ejemplo, el nodo 51 en la figura 9.2).
- **Rama:** es un camino desde la raíz hasta una hoja (por ejemplo, la rama 43-25-23-82). También se conoce como camino.

También se pueden definir algunas relaciones entre sus miembros:

- **Hijo:** se dice que un nodo es hijo, o descendiente de otro, si este último apunta al primero (en la figura 9.2, el nodo 23 es hijo del nodo 25).
- **Padre:** se dice que un nodo es padre de otro, si este último es apuntado por el primero (en la figura 9.2, el nodo 25 es padre del nodo 37)
- **Hermano:** dos nodos son hermanos si son apuntados por el mismo nodo, es decir, si tienen el mismo parente (en la figura 9.2, el nodo 23 y el 37 son hermanos).

De acuerdo con estas nuevas definiciones, un árbol se puede ver como un nodo raíz y todos sus sucesores. Si nos concentramos en uno de los hijos de la raíz, observamos que, a partir de él hacia abajo, también se forma un árbol más pequeño. Por esta razón, se le conoce como un subárbol, lo cual nos lleva a tener una definición recursiva para un árbol.

### 9.1.2 Notación de nivel de un árbol

Si se observa la representación gráfica de un árbol, se puede ver que los nodos se forman por niveles, partiendo desde la raíz y llegando hasta las hojas. Por ejemplo, todos los nodos hermanos siempre están al mismo nivel.

Existe una notación especial para hacer referencia a estas relaciones y definir ciertos conceptos que serán útiles en el manejo de estructuras de árbol. Algunos de los conceptos se definen en el nivel y grado de cada nodo, y en la altura y el grado del árbol de la siguiente manera:

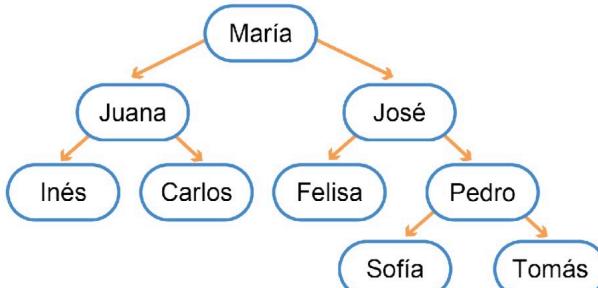
- **Nivel de un nodo:** es el número de arcos que deben ser recorridos, partiendo de la raíz para llegar hasta el nodo. La raíz tiene nivel 0 (en la figura 9.2, el nivel del nodo 58 es 2).
- **Altura de un nodo:** es el número de nodos en el camino más largo desde dicho nodo hasta una hoja. La altura de las hojas es 1 (en la figura 9.2, la altura del nodo 25 es 3).
- **Altura del árbol:** es la máxima de las alturas, considerando todos sus nodos (para el árbol de la figura 9.2, su altura es 4).
- **Grado de un nodo:** es el número de hijos que tiene un nodo (en la figura 9.2, el grado del nodo 25 es 4).
- **Grado del árbol:** es el máximo de los grados, considerando todos sus nodos (en la figura 9.2, el grado del árbol es 4).

## 9.2 Árbol binario

**H**ay un tipo especial de árbol en el que cada nodo puede tener más de dos sucesores, es decir, es un árbol de grado 2. A este tipo de árbol se le llama árbol binario (figura

9.3), tiene un gran número de aplicaciones dentro de las ciencias computacionales y otras ramas del conocimiento, por lo que lo estudiaremos más a fondo en el resto del capítulo.

No debemos perder de vista que un árbol es una estructura de datos; lo más importante para las ciencias computacionales es que se pueda implementar para guardar y manipular datos de una forma eficiente. Veamos una forma común de implementarlos.



*Figura 9.3 Ejemplo de un árbol binario que contiene nombres de personas.*

### 9.2.1 Implementación de un árbol binario

El ADT para un árbol binario es similar al ADT para una lista ligada, con la única diferencia que para un nodo de árbol se requieren varios sucesores, dos para el caso de un árbol binario.

Como un ADT es equivalente a una clase en la programación orientada a objetos, podemos aprovechar esta relación para definir el ADT utilizando la notación para definir clases, tal como lo hicimos en el caso de las listas ligadas.

Primero, definimos la clase Nodo. Cada nodo debe tener un espacio para guardar un dato. Un nodo puede guardar un dato de cualquier tipo T. Además, debe tener un par de apuntadores, que

apunten a sus dos hijos. En caso de no tener un hijo, este apuntador será nulo. La clase **Nodo** quedaría de la siguiente forma:

### NodoÁrbolBinario(T)

**info:** T

**hijoIzquierdo:** \*NodoÁrbolBinario(T)

**hijoDerecho:** \*NodoÁrbolBinario(T)

**Constructor:** Recibe el dato a ser guardado y regresa un apuntador a un nodo con el dato guardado en su atributo **info**, y sus dos hijos nulos.

Otros métodos para acceso y modificación de los miembros de la clase

Considerando que un árbol binario lo único que requiere es tener acceso a su raíz para poder llegar a cualquier nodo a partir de ella, y considerando que la raíz también es un nodo, podemos definir la clase **Árbol** de la siguiente forma:

### NodoÁrbolBinario(T)

**info:** T

**hijoIzquierdo:** \*NodoÁrbolBinario(T)

**hijoDerecho:** \*NodoÁrbolBinario(T)

**Constructor:** Recibe el dato a ser guardado y regresa un apuntador a un nodo con el dato guardado en su atributo **info**, y sus dos hijos nulos.

Otros métodos para acceso y modificación de los miembros de la clase

Escribamos las definiciones en C++, tomando en cuenta que, por el momento, las clases solo contarán con constructores como métodos y que el resto de los métodos que hagan falta, tanto

para la clase NodoÁrbolBinario como para la clase ÁrbolBinario, se podrán ir agregando conforme veamos otros conceptos. Para facilitar la implementación, el dato guardado en un nodo será un número entero no negativo, sin embargo, no se pierde generalidad porque la implementación para otros tipos es muy similar.

```

1. ///
2. // Implementación de la clase ArbolBinario //
3. ///
4.
5. #include <iostream>
6.
7. using namespace std;
8.
9. class NodoArbolBinario{
10. public:
11. int info;
12. NodoArbolBinario *hijoIzquierdo;
13. NodoArbolBinario *hijoDerecho;
14.
15. NodoArbolBinario(int dato){
16. info = dato;
17. hijoIzquierdo = nullptr;
18. hijoDerecho = nullptr;
19. }
20. };
21.
22. class ArbolBinario{
23. private:
24. NodoArbolBinario *raiz;
25. public:
26. ArbolBinario(){
27. raiz = nullptr;
28. }
29. };

```

Se puede observar que la codificación en C++ es prácticamente igual que la definición de la clase que se hizo anteriormente.

Veamos ahora algunas implementaciones particulares que utiliza un árbol binario para guardar datos con algún propósito específico.

### 9.2.2 Árbol binario de búsqueda

El Árbol binario de búsqueda (**BST**, por sus siglas en inglés, *Binary Search Tree*), es un árbol utilizado para hacer búsquedas de datos en forma muy eficiente.

Un BST debe cumplir las siguientes condiciones sobre los datos

que están guardados en él: la información de cada nodo es mayor que la información de cada uno de los nodos que están en su subárbol izquierdo y menor que la almacenada en los nodos que están en su subárbol derecho.

La figura 9.4 muestra un ejemplo de un BST, en el que se puede observar que todos sus nodos cumplen con las condiciones establecidas para un BST.

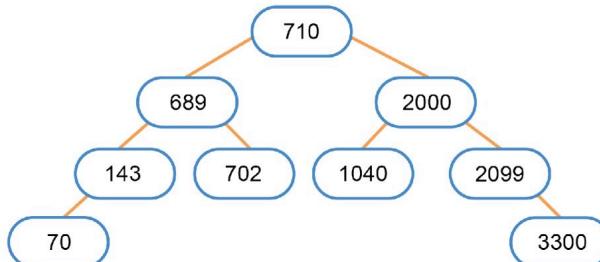


Figura 9.4 Ejemplo de un BST. Se observa que los datos guardados en él cumplen con las condiciones para formar un BST.

Si el árbol está bien balanceado, tendrá el mismo número de elementos en ambos lados de cada nodo. Para efectos prácticos, con que esté “casi” balanceado logra tener un buen desempeño en la búsqueda de datos. Una definición formal de balanceo se hará más adelante.

### 9.2.3 Representación

El ADT para un BST es igual al de cualquier árbol binario, la única diferencia es la forma en la que se manejan los datos en dicha estructura, la cual estará definida por las operaciones que se hacen sobre el mismo.

La inserción de un nuevo elemento debe mantener invariantes la propiedad de que, para cada nodo, su hijo izquierdo es menor que él, y su hijo derecho es mayor. Esto implica que un nuevo

elemento no se puede insertar en cualquier lugar sino en el lugar que mantenga el BST como un BST. De la misma forma, la eliminación de un elemento también debe mantener la propiedad principal de un BST.

El objetivo detrás de mantener el árbol con las propiedades establecidas para un BST es hacer una búsqueda óptima de datos. Esto se observará mejor al definir las operaciones sobre el árbol.

El código en C++ de un BST, solo tiene algunos cambios de nombre para poderlo distinguir de un árbol binario normal, pero en realidad es idéntico, como lo podrá comprobar si compara el código anterior con el siguiente:

```

1. ///
2. // Implementación de la clase BST //
3. ///
4.
5. #include <iostream>
6.
7. using namespace std;
8.
9. class NodoArbolBinario{
10. public:
11. int info;
12. NodoArbolBinario *hijoIzquierdo;
13. NodoArbolBinario *hijoDerecho;
14.
15. NodoArbolBinario(int dato){
16. info = dato;
17. hijoIzquierdo = nullptr;
18. hijoDerecho = nullptr;
19. }
20. };
21.
22. class BST{
23. private:
24. NodoArbolBinario *raiz;
25. public:
26. BST(){
27. raiz = nullptr;
28. }
29. };

```

## 9.2.4 Operaciones

Una de las propiedades más importantes de cualquier árbol binario es que su definición se puede realizar en forma recursiva, es decir, para cada nodo, lo que tiene en cada uno de sus hijos, es un árbol (subárbol), con sus hijos como raíz. Por ejemplo, en

la figura 9.4 se puede observar que, partiendo del nodo 689, hijo izquierdo del nodo 710, se forma un árbol más pequeño, y se le llama subárbol izquierdo del nodo raíz. De la misma forma, el árbol que tiene al nodo 2099 como raíz, es el subárbol derecho del nodo 2000.

Usando esta definición recursiva de árbol, podemos establecer las operaciones principales de un BST con algoritmos cortos que obtienen ventaja de la recursión computacional. Vemos las operaciones y sus algoritmos.

Considerando que el objetivo de un BST es realizar búsquedas, podemos definir algunas operaciones para su ADT, como son:

- Búsqueda de un elemento
- Inserción de un elemento
- Eliminación de un elemento
- Borrado (*delete*) del árbol completo

Desde luego que se pueden definir muchas operaciones más, pero por el momento, estas serán suficientes para este tipo de árbol.

Un árbol BST puede contener cualquier tipo de elementos ordenables. Para facilitar las explicaciones vamos a suponer que contiene enteros no negativos y que ninguno está repetido. Una vez definidas las operaciones se podrá observar que los elementos repetidos se pueden manejar fácilmente dentro de una BST.

### *Búsqueda de un elemento*

La búsqueda de un elemento se puede definir como: si el elemento a buscar es menor que la raíz, buscarlo en el subárbol

izquierdo, si es mayor, buscarlo en el subárbol derecho. Esta operación se repite hasta que se encuentra el elemento buscado (es decir, no es menor, ni mayor, sino igual), o la raíz del subárbol es un nodo nulo, en cuyo caso significa que el dato buscado no se encuentra en el árbol. El siguiente algoritmo define la búsqueda en un BST.

**1.** Preguntar si el nodo actual o nodo visitado (la primera vez es la raíz) contiene un valor (es decir, no es nulo)

**a.** Si la respuesta es verdadera, preguntar si el dato buscado es menor que el dato visitado

**i.** Si la respuesta es verdadera se procede a buscar el dato en el subárbol izquierdo

**ii.** Si la respuesta es falsa, preguntar si el dato buscado es mayor que el dato visitado

**1.** Si la respuesta es verdadera, buscar el dato en el subárbol derecho

**2.** Si la respuesta es falsa (es igual), la búsqueda termina exitosamente (el dato fue encontrado)

**b.** Si la respuesta (al punto 1) es falsa, la búsqueda termina con un fracaso, es decir, el elemento buscado no se encuentra en el BST.

Como ejemplo, sigamos el algoritmo para el árbol de la figura 9.4, busquemos el elemento 1040. En el paso 1, iniciamos preguntando si la raíz es nula; en este caso no es nula, ya que contiene el nodo con el elemento 710 y vamos al 1.a, preguntamos si el 1040 es menor que el 710; la respuesta es falsa y vamos

al 1.a.ii, preguntamos si  $1040 > 710$ ; la respuesta es verdadera por lo que seguimos la búsqueda en el subárbol derecho. Es aquí donde la definición recursiva nos ayuda.

Hacemos el nodo actual igual hijo derecho del nodo 710 y regresamos al punto 1 para continuar con la búsqueda, ahora en el subárbol derecho; ahora preguntamos si el nodo visitado contiene algún valor, la respuesta es verdadera porque contiene el valor 2000; preguntamos si el 1040 es menor que el 2000, lo cual es verdadero y hacemos el nodo actual igual al hijo izquierdo del nodo 2000, regresando al punto 1; enseguida preguntamos si el nodo contiene un valor, en este caso es verdadero porque contiene precisamente el 1040, sin embargo, el algoritmo no se ha dado cuenta que es el que buscamos por lo que pasamos al punto 1, preguntamos si el 1040 del nodo actual es menor que el 1040 buscado como la respuesta es vamos al 1.a.ii y preguntamos si el 1040 es mayor que el 1040 como esta respuesta final es falsa y vamos al 1.a.ii.1, la búsqueda fue exitosa.

El código de búsqueda se coloca en un método dentro de la clase BST. Para esta implementación se tomarán en cuenta dos cosas:

- El procedimiento para hacer la búsqueda para el usuario (público) debe recibir solo el dato a ser buscado (línea 2). Sin embargo, el procedimiento de búsqueda, al hacerlo recursivo, requiere recibir también el nodo raíz (línea 6) ya que este nodo raíz irá cambiando cuando se mueve uno al subárbol correspondiente. Por esta razón, se utiliza un método auxiliar recursivo y privado (líneas 6 a 16), el cual se manda a llamar desde el método que ve el usuario (líneas 2 a la 4).
- En algunos algoritmos, el método que hace la búsqueda re-

gresa el apuntador al nodo que contiene el elemento buscado (líneas 2 y 6). Esto es útil ya que, si se busca normalmente, es para hacer algo con la información que contiene el nodo.

```

1. public:
2. NodoArbolBinario* buscar(int dato){
3. return buscarRecurSivo(raiz, dato);
4. }
5. private:
6. NodoArbolBinario* buscarRecurSivo(NodoArbolBinario *p, int dato){
7. if (p == nullptr)
8. return nullptr;
9. else
10. if (dato < p->info)
11. return buscarRecurSivo(p->hijoIzquierdo, dato);
12. else if (dato > p->info)
13. return buscarRecurSivo(p->hijoDerecho, dato);
14. else
15. return p;
16. }

```

Para utilizar el método público **buscar**, debemos declarar un objeto de la clase **BST**, digamos **BST miBST**, y luego utilizar el operador punto para invocar al método **buscar**. Este método automáticamente invocará el método **buscarRecurSivo** con la raíz del árbol y el dato a ser buscado.

### *Inserción de un elemento*

La inserción de un elemento se puede definir como: si el elemento a insertar es menor que la raíz, insertarlo en su subárbol izquierdo; si es mayor, insertarlo en su subárbol derecho. Esta operación se repite hasta que en alguno de los subárboles que se debe insertar encuentren un espacio, es decir, un nodo nulo. En ese momento se inserta el elemento, garantizando que el árbol binario sigue cumpliendo las características de un BST. El algoritmo para inserción se describe enseguida:

**1.** Preguntar si el nodo actual o nodo visitado (la primera vez es la raíz) contiene un valor (es decir, no es nulo)

**a.** Si la respuesta es verdadera, preguntar si el dato a insertar es menor que el dato visitado

- i. Si la respuesta es verdadera se procede a insertar el dato en el subárbol izquierdo
- ii. Si la respuesta es falsa, preguntar si el dato a insertar es mayor que el dato visitado
  - 1. Si la respuesta es verdadera, insertar el dato en el subárbol derecho
  - 2. Si la respuesta es falsa (es igual), el dato a ser insertado ya se encuentra en el árbol y la inserción termina.
- b. Si la respuesta (al punto 1) es falsa, se procede a realizar la inserción en ese lugar del BST (crear nuevo nodo, almacenar la información y establecer las ligas entre el nuevo nodo y su parent).

Como ejemplo, insertemos el elemento 1500 en el BST de la figura 9.4. Iniciamos en el punto 1 con la raíz como el nodo actual; el nodo sí contiene un valor, que es el 710, por lo que pasamos al punto 1.a.; como 1500 no es menor que el 710 pasamos al punto 1.a.ii, como el 1500 es mayor que el 710, pasamos el punto 1.a.ii.1; eso significa que debemos continuar la inserción en el subárbol derecho de la raíz, es decir, en el subárbol que tiene como raíz el nodo 2000. Entonces regresamos al punto 1. El nodo contiene el valor 2000 por lo que pasamos al punto 1.a, el 1500 es menor que el 2000, por lo que pasamos al punto 1.a.i, lo que significa que debemos continuar la inserción en el subárbol izquierdo del 2000, es decir, en el nodo 1040. Lo hacemos el nodo actual y regresamos al punto 1. El nodo contiene el valor 1040 por lo que vamos al punto 1.a, el 1500 no es menor que el 1040 por lo que pasamos al punto 1.a.ii, el 1500 es mayor que el 1040

por lo que continuamos la inserción en el subárbol derecho del 1040, el cual es nulo.

Cuando regresamos al punto 1, pasamos de inmediato el punto 1.b y procedemos a insertar el nuevo valor en ese sitio. El árbol BST que resulta después de la inserción es el de la figura 9.5.

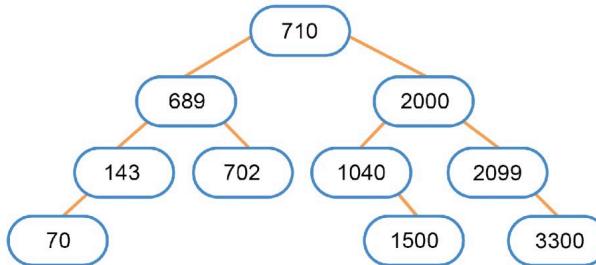


Figura 9.5 BST de la figura 9.4 después de la inserción del elemento 1500.

La implementación en código de C++ también se facilita si se hace en forma recursiva.

```

1. public:
2. void insertar(int dato){
3. raiz = insertarRecursivo(raiz, dato);
4. }
5. private:
6. NodoArbolBinario* insertarRecursivo(NodoArbolBinario *p, int dato){
7. if (p == nullptr)
8. p = new NodoArbolBinario(dato);
9. else if (dato < p->info)
10. p->hijoIzquierdo = insertarRecursivo(p->hijoIzquierdo, dato);
11. else if (dato > p->info)
12. p->hijoDerecho = insertarRecursivo(p->hijoDerecho, dato);
13. else
14. cout << "El elemento está repetido y no se pudo insertar" << endl;
15. return p;
16. }

```

El método que implementa exactamente el algoritmo 9.2, es el que se programa en forma recursiva y se llama **insertarRecursivo** (línea 6). Dicho método recibe como parámetros la raíz del árbol (o subárbol) y el dato que se desea **insertar**. Sin embargo, para el usuario, se tienen el método **insertar**, que solo recibe el dato a ser insertado (línea 2). Este método, que es público, llama al recursivo, que es privado, dando como nodo inicial a la raíz del

árbol. El método recursivo regresa el apuntador al nuevo nodo, lo cual hace que al final se actualice la raíz (línea 3).

Si el nodo recibido es nulo (línea 7), se crea un nuevo nodo (línea 8) con el dato recibido. Si no, si el dato al ser insertado es menor que la información del nodo (línea 9) se llama al procedimiento recursivo, pero ahora con el hijo izquierdo como raíz. Si el dato a insertar es mayor que el dato del nodo (línea 11), se llama al método recursivo, pero ahora con el hijo derecho como raíz. Si no es mayor ni menor (línea 13), significa que el dato a ser insertado ya está en el árbol y sólo se imprime un letrero (línea 14). Para llamar este método bastará usar el operador punto con un objeto de la clase BST.

### *Eliminación de un elemento*

Eliminar un elemento es la tarea más complicada que se realiza en este tipo de árbol. La razón es que la eliminación también debe garantizar que mantiene las propiedades del BST. Eliminar una hoja no es ningún problema, pero eliminar un nodo interior no es tan simple porque tiene dos hijos. Sin embargo, existe un algoritmo bien establecido para eliminar cualquier nodo y garantizar que el resultado sigue siendo un BST.

Otra cosa que se debe tomar en cuenta en la eliminación es que, la memoria que ocupa un nodo que se va a eliminar, debe liberarse (regresarla al **heap**) para que el sistema la pueda reutilizar cuando necesite crear otros nodos. Si la memoria no se libera y el sistema sigue operando, se puede dar el caso que cuando se requiera generar un nodo nuevo, el sistema nos diga que ya no hay memoria disponible, cuando en realidad el árbol sólo tiene unos cuantos elementos y debería haber suficiente memoria para crear un nodo. La razón es que al eliminar un nodo no se está liberando la memoria que ocupa, lo que provoca un **leak** de

memoria.

La tarea de liberar la memoria no es la misma siempre. Depende del lenguaje de programación. Algunos lenguajes de programación tienen lo que se conoce como “**Recolector de Basura**”, este consiste en un programa que se ejecuta periódicamente durante la ejecución de un programa. Al ejecutarse, revisa que todas las posiciones de memoria utilizadas estén referenciadas, es decir, que exista al menos una variable que tenga un apuntador a esa posición de memoria. Si encuentra alguna posición de memoria que no esté referenciada, automáticamente la libera. Java es un ejemplo de este tipo de lenguaje con **Recolector de Basura**.

Si un lenguaje cuenta con **Recolector de Basura**, bastará con hacer nula la variable (**apuntador**) que apunta a una determinada posición de memoria (en Java, si el apuntador es **p**, bastará con hacer **p = null**).

Existen algunos lenguajes que no tienen **Recolector de Basura**, como por ejemplo C++. En este caso, es responsabilidad total del programador liberar la memoria que se deje de usar. Para esto, debe invocar alguna función que libere dicha posición de memoria, para C++ es **delete**, si el apuntador es **p**, se invoca **delete(p)**.

La idea de la eliminación es simple y se divide en dos casos, de acuerdo con el tipo de nodo:

- Si el nodo a eliminar es una hoja (se sabe que es hoja porque sus dos hijos son nulos) bastará eliminar la memoria de ese nodo y hacer que su papá apunte a nulo (en el hijo que estaba apuntando al nodo eliminado).
- Si el nodo a eliminar es un nodo interno, se sustituye por

el nodo más a la derecha de su subárbol izquierdo (también se podría reemplazar por el nodo más a la izquierda de su subárbol derecho). Esto garantiza que el nuevo dato del nodo cumple con la condición del BST, lo que se sustituye es solo la información, para no tener que cambiar ligas. De esta forma, una vez sustituida la información y colocando la liga del hijo izquierdo o derecho del padre del nodo a eliminar, se procede a eliminar el nodo por el que se sustituyó.

El algoritmo recursivo es similar al de búsqueda y se muestra a continuación.

**1.** Preguntar si el nodo actual o nodo visitado (la primera vez es la raíz) contiene un valor (es decir, no es nulo)

**a.** Si la respuesta es verdadera, preguntar si el dato a eliminar es menor que el dato visitado

**i.** Si la respuesta es verdadera, invocar el proceso de eliminar con el subárbol izquierdo

**ii.** Si la respuesta es falsa, preguntar si el dato a eliminar es mayor que el dato visitado

**1.** Si la respuesta es verdadera, invocar el proceso de eliminar con el subárbol derecho

**2.** Si la respuesta es falsa (es igual), se elimina el nodo actual.

**a.** Si es hoja, la eliminación es directa

**b.** Si tiene un solo hijo, se reemplaza por éste

**c.** Si tiene dos hijos, se reemplaza por el nodo que esté

más a la derecha del subárbol izquierdo

**d.** En estos dos últimos casos (b y c), se libera el espacio de memoria del nodo por el que se reemplaza, mientras que en el primero (a) se libera el correspondiente al nodo en cuestión.

**b.** Si la respuesta (al punto 1) es falsa, el dato no está en el árbol (fracaso).

Como ejemplo, eliminemos el elemento 689 del árbol de la figura 9.6.

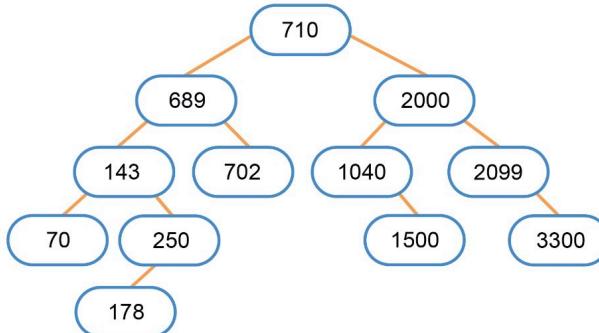


Figura 9.6 Árbol binario para el ejemplo de eliminación.

Iniciamos con la pregunta 1, en la raíz. Como la raíz sí contiene el valor 710 pasamos a 1.a; el valor a eliminar es menor que el actual, es decir,  $689 < 710$ , por lo que, pasamos al subárbol izquierdo, de acuerdo a 1.a.i, y reiniciamos la eliminación con la pregunta 1. La raíz actual contiene el valor 689, por lo que pasamos a la 1.a. como el valor a ser eliminado no es menor que el actual (es igual), nos lleva a 1.a.ii; aquí, el valor a ser eliminado tampoco es mayor que el actual, por lo que pasamos a 1.a.ii.2, y procedemos a eliminar el nodo actual. El nodo actual a ser eliminado tiene dos hijos (1.a.ii.2.c), entonces, tenemos que reem-

plazarlo por el nodo más a la derecha de su subárbol izquierdo, dicho nodo es el 250, por lo anterior reemplazamos el 689 por el 250, quedando solo por eliminar el nodo 250.

El nodo 250, como era de esperarse, solo tiene hijo izquierdo, de otra forma no sería el nodo de más a la derecha. Se coloca el hijo derecho de su papá, es decir, el nodo 143, a su hijo izquierdo, es decir el 178. Ahora el hijo derecho del 143 es el 178. Finalmente, se elimina el nodo que contiene el 250, quedando el árbol como el que se muestra en la figura 9.7. Se debe tener en cuenta que, en C++, la eliminación de un nodo se debe hacer explícitamente, esto es, mediante la instrucción **delete**. En un lenguaje con recolector de basura, como Java, bastará con dejar de apuntar al nodo que se desea borrar y el recolector hará el resto.

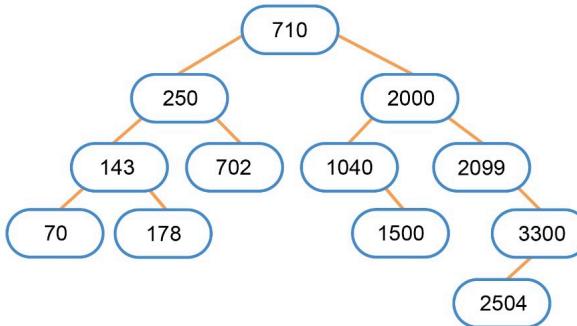


Figura 9.7 Árbol binario después de la eliminación del ejemplo.

La implementación del algoritmo 9.3 en C++ se muestra a continuación.

```

1. public:
2. void eliminar(int dato){
3. raiz = eliminarRecursivo(raiz, dato);
4. }
5. private:
6. NodoArbolBinario* eliminarRecursivo(NodoArbolBinario *p, int dato){
7. if (p == nullptr)
8. cout << "El dato no fue encontrado" << endl;
9. else if (dato < p->info){ // seguir buscando en su subárbol izquierdo
10. p->hijoIzquierdo = eliminarRecursivo(p->hijoIzquierdo, dato);
11. }
12. else if (dato > p->info){ // seguir buscando en su subárbol derecho
13. p->hijoDerecho = eliminarRecursivo(p->hijoDerecho, dato);
14. }
15. else{ // dato encontrado
16. NodoArbolBinario *q = p; // nodo a eliminar del árbol
17. if (q->hijoIzquierdo == nullptr)
18. p = q->hijoDerecho;
19. else if (q->hijoDerecho == nullptr)
20. p = q->hijoIzquierdo;
21. else // tiene los dos hijos
22. q = reemplazar(q);
23. delete(q);
24. }
25. return p;
26. }
27.
28. NodoArbolBinario* reemplazar(NodoArbolBinario *act){
29. NodoArbolBinario *a, *p;
30. p = act;
31. a = act->hijoIzquierdo; // subárbol de los menores que la raíz
32. while (a->hijoDerecho != nullptr){ // encontrar el más a la derecha
33. // del izquierdo
34. p = a;
35. a = a->hijoDerecho;
36. }
37. // copiar en act el valor del nodo apuntado por a
38. act->info = a->info;
39. if (p == act) // a es el hijo izquierdo de act
40. p->hijoIzquierdo = a->hijoIzquierdo; // enlaza el subárbol izquierdo
41. else
42. p->hijoDerecho = a->hijoIzquierdo; // enlaza subárbol derecho
43. return a;
44. }

```

En la implementación mostrada, se utiliza un método **reemplazar** (línea 28) para hacer el procedimiento de búsqueda del nodo más a la derecha del subárbol izquierdo. Este procedimiento regresa el nodo hoja por el que se va a reemplazar el actual.

El método que implementa el algoritmo 9.3 es el recursivo (línea 6) el cual recibe como parámetros la raíz del árbol donde se está buscando el dato a eliminar y el dato que se desea eliminar. Sin embargo, el método que se le proporciona al usuario solo recibe el dato a ser eliminado (línea 2). Este método es público y

manda a llamar al recursivo (línea 3), que es privado, con la raíz del árbol y el dato a ser borrado.

En la implementación mostrada, la pregunta inicial se hace al revés. Se pregunta si el nodo actual es nulo (en el algoritmo se pregunta si el nodo actual tiene un valor, es decir, si no es nulo), por lo que el **if** y el **else** están al revés con respecto al algoritmo, pero el resultado es el mismo.

El procedimiento es similar al de búsqueda, pero en la línea 16 inicia el procedimiento para borrar el nodo, una vez que se encontró. Se inicia haciendo una copia en **q** del apuntador **p** al nodo a ser eliminado (línea 16). Si el hijo izquierdo del nodo a ser eliminado es nulo (línea 17), se sustituye por el hijo derecho (línea 18). Este caso es muy interesante ya que, si el hijo derecho también es nulo, el nodo a eliminar simplemente se convierte en nulo y eso equivale al caso en el que el nodo a eliminar es una hoja; pero si el hijo derecho no es nulo, el nodo a eliminar se sustituye por éste, lo que equivale al caso en el que el nodo a eliminar tiene un solo hijo y es el hijo derecho. Si el hijo izquierdo no es nulo pero el hijo derecho sí es nulo (línea 19), se sustituye por el hijo izquierdo (línea 20), lo cual corresponde al caso donde el nodo a eliminar sólo tiene un hijo y es el hijo derecho. Si ninguno de los hijos es **nulo** (línea 21) se procede a reemplazarlo por el nodo más a la derecha de su subárbol izquierdo, lo cual se hace con el método **reemplazar** (línea 28).

En el método **reemplazar**, inicialmente se coloca **a** en el hijo izquierdo (línea 31) del original **act** que se desea eliminar, y se coloca a su papá (que es el original **act**) en **p** (línea 30). Luego, se va moviendo siempre al hijo derecho de **a** (línea 35) hasta que se encuentre un hijo derecho que sea **nulo** (línea 32, en la condición del **while**).

Conforme se mueve al hijo derecho que no sea nulo (línea 35), se debe ir guardando en **p** el papá del último nodo **a** que no es **nulo** (línea 34). Cuando se encuentra un hijo derecho de **a** que es **nulo**, se sabe que **a** contiene el nodo que reemplazará al que se desea eliminar. Se termina el ciclo (línea 36) y lo primero que se hace es copiar la información del nodo **a** (el más a la derecha del subárbol izquierdo del que se desea eliminar) al nodo original **act** que se desea eliminar (línea 38).

El papá de **a** está guardado en **p**; si **p** es igual al original **act** (línea 39), entonces **a** es el hijo izquierdo del original **act**, lo que significa que **a** no tiene hijo derecho. En ese caso, el hijo izquierdo del papá **p** se hace igual al hijo izquierdo de **a** (línea 40), y **a** es el nodo que se debe eliminar. Si **p** no es igual al original **act** (línea 41), significa que al menos una vez hubo un hijo derecho diferente de nulo y, como al menos una vez entró al ciclo y se salió, significa que el nodo **a** ya no tiene hijo derecho, pero posiblemente tenga hijo izquierdo (puede no tener, es decir, su hijo izquierdo apunta a nulo). En ese momento, el hijo derecho del papá **p** se hace igual al hijo izquierdo de **a** (línea 42). Si el hijo izquierdo de **a** es **nulo** significa que **a** era una hoja.

Como ya se restauraron las ligas del papá de **p**, el nodo **a** es el que se debe eliminar y por eso es regresado por el método **reemplazar** (línea 43). En el método recursivo de eliminar, si se invocó al método **reemplazar** entonces **q** recibe el nodo regresado por **reemplazar** (línea 22).

Por eso, al final del procedimiento recursivo de eliminar, es seguro que **q** contiene el nodo que se debe eliminar, en cualquiera de los casos, y se procede a liberar su memoria (línea 23).

### *Borrado (delete) del árbol completo*

La operación de borrado del árbol completo se hace sumamente simple si el lenguaje cuenta con **Recolector de Basura**, porque bastará hacer que la raíz del árbol apunte a **nulo**. Esto hará que el **Recolector de Basura** recoja toda la memoria a medida que se va dejando de referenciar, hasta liberar todo el árbol. Por ejemplo, en Java, si el árbol es **miArbol**, bastará con invocar la instrucción **miArbol.raiz = null**.

Sin embargo, si el lenguaje no tiene **Recolector de Basura**, el proceso se complica un poco debido a que tiene que irse liberando nodo por nodo pero partiendo de las hojas, ya que si se parte de la raíz se pierden las referencias a los hijos y se provocarían **leaks**, sin liberar la memoria.

El recorrido de un árbol nodo por nodo, es una operación más que se verá en la siguiente sección por lo que dejaremos el borrado del árbol completo para dicha sección.

### 9.2.5 Recorridos

El recorrido de un árbol es una operación que permite visitar todos los nodos de un árbol, uno por uno y una sola vez en cada recorrido. El recorrido es una operación que se puede definir para cualquier tipo de árbol, sin embargo, en esta sección nos limitaremos a definirlo para un árbol binario (su definición para árboles de mayor orden es muy similar y se puede extrapolar fácilmente). Por otro lado, el recorrido de un árbol binario funciona para cualquier árbol binario, incluyendo los árboles BST o los árboles AVL (ver sección 9.2).

En un recorrido, en cuanto un nodo se visita, se puede hacer algo con dicho nodo, por ejemplo, imprimir su información, eliminarlo (cuidando que no se produzcan leaks), etc.

Hay tres formas diferentes de recorrer los nodos de un árbol, las cuales definen el orden en el que se visitarán sus nodos, sus nombres son:

- recorrido en preorden,
- recorrido en inorden y
- recorrido en posorden.

Todos los recorridos aprovechan la naturaleza recursiva de un árbol, el tipo de recorrido indica en qué momento se visita la raíz del subárbol actual. Por ejemplo, el recorrido en preorden, nos dice que primero se visita la raíz y luego se continúan visitando sus hijos (subárboles izquierdo y derecho), también en preorden; el recorrido en inorden, nos dice que la visita de la raíz queda en medio del recorrido de sus hijos (subárboles izquierdo y derecho), y el recorrido en posorden indica que la raíz se visita después de haber recorrido sus hijos (subárboles izquierdo y derecho) también en posorden.

Los algoritmos de los recorridos son realmente simples si se establecen en forma recursiva. Se puede ver en los algoritmos 9.4, 9.5 y 9.6. Todos los recorridos reciben un nodo desde donde se inicia el recorrido, el cual se puede ver como el nodo actual. El nodo recibido es la raíz del subárbol que se esté recorriendo en ese momento, a partir de él se inicia todo el recorrido del resto de los nodos.

Al inicio de todo, el recorrido recibe la raíz del árbol; además, cuando se dice que un nodo es visitado significa que se hizo algo con él (por ejemplo, como ya se comentó, imprimir su información).

### Recorrido de un árbol binario en preorden

1. Visitar raíz
2. Recorrer el subárbol izquierdo en preorden
3. Recorrer el subárbol derecho en preorden

#### Recorrido de un árbol binario en inorden

4. Recorrer el subárbol izquierdo en inorden
5. Visitar raíz
6. Recorrer el subárbol derecho en inorden

#### Recorrido de un árbol binario en posorden

7. Recorrer el subárbol izquierdo en posorden
8. Recorrer el subárbol derecho en posorden
9. Visitar raíz

Como ejemplo de cómo llevar a cabo el recorrido, hagamos el recorrido en preorden del árbol binario de la figura 9.8.

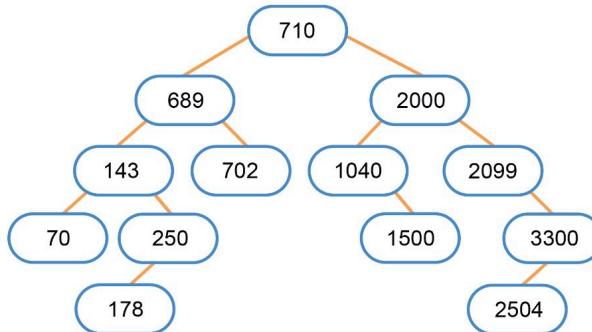


Figura 9.8 Árbol binario para ejemplos de recorridos.

Cada vez que se “visita la raíz” vamos a suponer que imprimimos el contenido del nodo. Debemos tomar en cuenta que el recorrido se hace en forma recursiva, por lo que, cuando se realiza

una acción, debemos regresar al punto en donde se llamó a esta acción para continuar con el resto de las acciones.

Para poder observar las llamadas recursivas vamos a ir enumerando el número de llamada, iniciando con la 1, seguida del número de paso. Por ejemplo, la primera vez que se visita la raíz del árbol se colocará el número 1.1, indicando que es la llamada 1 y que se está ejecutando el punto 1 del algoritmo. Iniciamos con la raíz del árbol, es decir, el nodo 710:

**1.1** Visitar la raíz: 710

**1.2** Recorrer el subárbol izquierdo en preorden (llamada 2), teniendo ahora como raíz el nodo 689.

**2.1** Visitar raíz: 710-689

**2.2** Recorrer el subárbol izquierdo del 689 en preorden (llamada 3), teniendo como raíz el 143.

**3.1** Visitar raíz: 710-689-143

**3.2** Recorrer el subárbol izquierdo del 143 en preorden (llamada 4), teniendo como raíz el 70.

**4.1** Visitar raíz: 710-689-143-70

**4.2** Recorrer el subárbol izquierdo del 70 en preorden, pero como no tiene subárbol izquierdo, no se hace la llamada.

**4.3** Recorrer el subárbol derecho del 70 en preorden, pero no tiene subárbol derecho, no se hace la llamada.

En este momento se termina la llamada 4 y regresamos a donde se hizo la llamada 4, que es el punto 3.2, para continuar con el

### 3.3.

**3.3** Recorrer el subárbol derecho del 143 en preorden (llamada 5), teniendo como raíz el 250.

**5.1** Visitar raíz: 710-689-143-70-250

**5.2** Recorrer el subárbol izquierdo del 250 en preorden (llamada 6), teniendo como raíz el 178.

**6.1** Visitar raíz: 710-689-143-70-250-178

**6.2** Recorrer el subárbol izquierdo del 178 en preorden, pero no tiene por lo que no se hace llamada.

**6.3** Recorrer el subárbol derecho del 178 en preorden, pero no tiene.

Terminamos la llamada 6 y regresamos a donde se hizo esta llamada que fue en la llamada 5, continuando con el 5.3.

**5.3** Recorrer el subárbol derecho del 250 en preorden, pero no tiene.

Terminamos la llamada 5 y regresamos a donde se hizo la llamada 5, que fue en la llamada 3. Pero ya se terminó la llamada 3, por lo que regresamos a donde se hizo la llamada 3, que fue en la el punto 2.2, por lo que regresamos al punto 2.3.

**2.3** Recorrer el subárbol derecho del 689 en preorden (llamada 7), teniendo como raíz el 702.

**7.1** Visitar raíz: 710-689-143-70-250-178-702

**7.2** Recorrer el subárbol izquierdo del 702 en preorden. No

tiene.

**7.3** Recorrer el subárbol derecho del 702 en preorden. No tiene.

Terminamos la llamada 7. Regresamos al 2.3, que fue donde se hizo la llamada 7, lo que indica que también se terminó la llamada 2 y regresamos a donde se hizo la llamada 2, que fue en el punto 1.2. Vamos ahora al 1.3.

**1.3** Recorrer el subárbol derecho del 710 en preorden (llamada 8), ahora con la raíz 2000.

**8.1 Visitar raíz: 710-689-143-70-250-178-702-2000**

**8.2** Recorrer el subárbol izquierdo del 2000 en preorden (llamada 9), ahora con la raíz 1040.

**9.1 Visitar raíz: 710-689-143-70-250-178-702-2000-1040**

**9.2** Recorrer el subárbol izquierdo del nodo 1040 en preorden.  
No tiene.

**9.3** Recorrer el subárbol derecho del nodo 1040 en preorden (llamada 10), ahora con la raíz 1500.

**10.1** Visitar raíz:  
710-689-143-70-250-178-702-2000-1040-1500

**10.2** Recorrer el subárbol izquierdo del nodo 1500 en preorden.  
No tiene.

**10.3** Recorrer el subárbol derecho del nodo 1500 en preorden.  
No tiene.

Regresamos al 8.2 y continuamos con el 8.3.

**8.3 Recorrer el subárbol derecho del 2000 en preorden (llamada 11), ahora con la raíz 2099.**

**11.1** Visitar raíz:  
710-689-143-70-250-178-702-2000-1040-1500-2099

**11.2** Recorrer el subárbol izquierdo del 2099 en preorden. No tiene.

**11.3** Recorrer el subárbol derecho del 2099 en preorden (llamada 12), ahora con la raíz 3300.

**12.1** Visitar raíz:  
710-689-143-70-250-178-702-2000-1040-1500-2099-3300

**12.2** Recorrer el subárbol izquierdo del 3300 en preorden (llamada 13), ahora con la raíz 2504.

**13.1** Visitar raíz:  
710-689-143-70-250-178-702-2000-1040-1500-2099-3300  
-2504

**13.2** Recorrer el subárbol izquierdo del 2504 en preorden. No tiene.

**13.3** Recorrer el subárbol derecho del 2504 en preorden. No tiene.

Terminamos la llamada 13 y regresamos al 12.2 para continuar con el 12.3.

**12.3** Recorrer el subárbol derecho del 3300 en preorden. No

tiene.

Terminamos la llamada 12 y regresamos al 11.3. Terminamos la llamada 11 y regresamos al 8.3. Terminamos la llamada 8 y regresamos al 1.3, con lo que terminamos la llamada 1 y terminamos el recorrido en preorden. La impresión final es:

**710-689-143-70-250-178-702-2000-1040-1500-2099-3300-  
2504**

Si recorremos el mismo árbol de la figura 9.8 en inorder (lo dejamos al lector como ejercicio), la secuencia obtenida sería:

**70-143-178-250-689-702-710-1040-1500-2000-2099-2504-  
3300**

Observe que al recorrer un BST en inorder, el resultado es una secuencia ordenada de menor a mayor, lo cual nos puede llevar a un método de ordenamiento diferente, en el que primero formamos el árbol BST y luego lo recorremos en inorder.

Si recorremos en posorden el mismo árbol de la figura 9.8 (lo dejamos como ejercicio al lector), la secuencia obtenida es:

**70-178-250-143-702-689-1500-1040-2504-3300-2099-2000-  
-710**

Los recorridos en preorden y posorden no proporcionan secuencias ordenadas, pero tiene otros usos sumamente valiosos. Por ejemplo, uno de los principales usos del recorrido en posorden es el borrado completo de un árbol BST, en lenguajes que no tiene Recolector de Basura.

La implementación en C++ de los tres algoritmos de recorrido, se pueden hacer como métodos para el ADT BST que ya se construyó.

```

1. public:
2. void preorden(){
3. preordenRecursivo(raiz);
4. }
5. private:
6. void preordenRecursivo(NodoArbolBinario *p){
7. if (p != nullptr){
8. cout << p->info << " ";
9. preordenRecursivo(p->hijoIzquierdo);
10. preordenRecursivo(p->hijoDerecho);
11. }
12. }
13. public:
14. void inorden(){
15. inordenRecursivo(raiz);
16. }
17. private:
18. void inordenRecursivo(NodoArbolBinario *p){
19. if (p != nullptr){
20. inordenRecursivo(p->hijoIzquierdo);
21. cout << p->info << " ";
22. inordenRecursivo(p->hijoDerecho);
23. }
24. }
25.
26. public:
27. void posorden(){
28. posordenRecursivo(raiz);
29. }
30. private:
31. void posordenRecursivo(NodoArbolBinario *p){
32. if (p != nullptr){
33. posordenRecursivo(p->hijoIzquierdo);
34. posordenRecursivo(p->hijoDerecho);
35. cout << p->info << " ";
36. }

```

En realidad, la implementación es sumamente simple si se hace en forma recursiva. El único cambio es que para hacerlo en forma recursiva, es necesario que el método reciba el nodo raíz como parámetro, entonces, para lograrlo sin afectar al usuario, se crea un método auxiliar público, el cual no recibe parámetros; ese método llama al método privado recursivo con la raíz del árbol como argumento.

Los métodos públicos se llaman: preorden, inorden y posorden, (líneas 2, 14 y 27, respectivamente), y los privados recursivos se llaman: preordenRecursivo, inordenRecursivo y posordenRecursivo (líneas 6, 18 y 31, respectivamente). En todos los métodos recursivos, lo primero que se verifica es que la raíz del subárbol que se recibe como parámetro no sea nula (líneas 7, 19 y 32).

Si es nula no se hace nada y si no es nula se procede al recorrido,

con llamadas recursivas con los subárboles izquierdo (líneas 9, 20 y 33) y derecho (líneas 10, 22 y 34) y una visita de la raíz, que en este caso consiste sólo en imprimir la información del nodo (líneas 8, 21 y 35), cuyo lugar depende del algoritmo que se esté implementado.

### *Borrado (delete) del árbol completo usando recorridos*

En un lenguaje de programación que no tenga **Recolector de Basura**, si se desea borrar un árbol completo es necesario hacerlo de abajo hacia arriba, es decir, iniciar borrando las hojas y terminar borrando la raíz principal. La razón es muy simple: nunca se debe borrar primero un nodo que tenga hijos, al borrarlo se perderían los apuntadores a sus hijos, que son las únicas referencias que se tienen para llegar a ellos. De esta forma, iniciamos borrando las hojas, que no tienen hijos, y luego sus papás, que ya no tendrían hijos, y así sucesivamente hasta llegar a la raíz, que sería la última en ser borrada.

La forma de borrar el árbol completo que se explicó implicaría borrar primero los hijos y al final la raíz. Esta es precisamente la forma en la que son visitados los nodos de un árbol cuando se hace el recorrido en posorden (algoritmo 9.6). Hacemos una pequeña modificación y en lugar de visitar la raíz, se borra. El algoritmo de borrado del árbol completo quedaría así:

#### Borrado de un árbol completo (delete)

1. Recorrer el subárbol izquierdo en posorden
2. Recorrer el subárbol derecho en posorden
3. Borrar la raíz

La implementación en C++ del borrado completo es muy similar al recorrido en posorden, tal y como se ha comentado. En ese

caso también se hace como un método del ADT BST:

```

1. public:
2. void borrar(){
3. borrarRecursivo(raiz);
4. raiz = nullptr;
5. }
6. private:
7. void borrarRecursivo(NodoArbolBinario *p){
8. if (p != nullptr){
9. borrarRecursivo(p->hijoIzquierdo);
10. borrarRecursivo(p->hijoDerecho);
11. cout << "Borrando: " << p->info << endl;
12. delete(p); // podría llamar a eliminar(p) pero no es necesario
13. }
14. }

```

De la misma forma que en los recorridos, se crea un método auxiliar público llamado **borrar** (línea 2), el cual no tiene parámetros; a su vez, este método llama a un método privado **borrarRecursivo** (línea 7), que sí recibe como parámetro la raíz del BST que se desea borrar. El método borrar, termina colocando la raíz del BST a nulo (línea 4).

Se puede observar que el método recursivo es exactamente igual que el recorrido en posorden, solo que al visitar la raíz, además de imprimir el contenido del nodo que se está borrando (línea 11) se libera la memoria del nodo (línea 12).

## 9.3 AVL

**E**l objetivo principal de un árbol binario de búsqueda es que sea muy eficiente al buscar un elemento guardado en su estructura de datos. La idea es que el árbol se comporte como la búsqueda binaria, en donde con una sola comparación se puede eliminar la mitad de los datos (la mitad izquierda o la derecha, porque solo continua por uno de los hijos).

Recordemos que la búsqueda binaria es de orden  **$O(\log n)$** , donde **n** es el número de datos que contiene el árbol. El número de comparaciones que realiza para llegar al final del árbol, en el peor de los casos, es el de su altura máxima, la cual es  **$\log(n)$** .

En realidad, la altura máxima de un árbol binario es  $\log(n)$  si y solo si está balanceado, es decir, que tenga la misma cantidad de datos en su subárbol izquierdo que en su subárbol derecho.

La forma que tiene el árbol depende completamente del orden en el que se inserten los elementos del mismo. Por ejemplo, al insertar los siguientes elementos en un BST: 6, 3, 8, 1, 5, 7, 9, el árbol resultante es el de la figura 9.9.

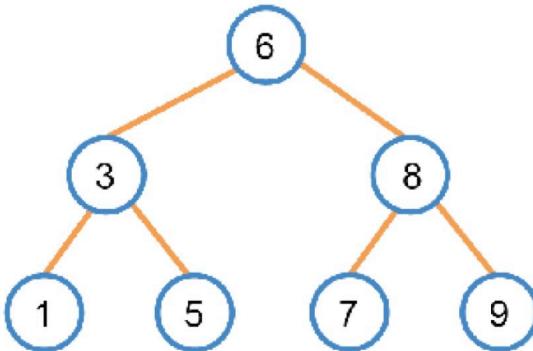


Figura 9.9 Ejemplo de un BST balanceado.

Se puede observar que el BST de la figura 9.9 está completamente balanceado. Sin embargo, si se forma un BST, con los mismos números, pero ahora entrando en el siguiente orden: 1, 3, 5, 6, 7, 8, 9, el árbol resultante es el de la figura 9.10, el cual está completamente desbalanceado.

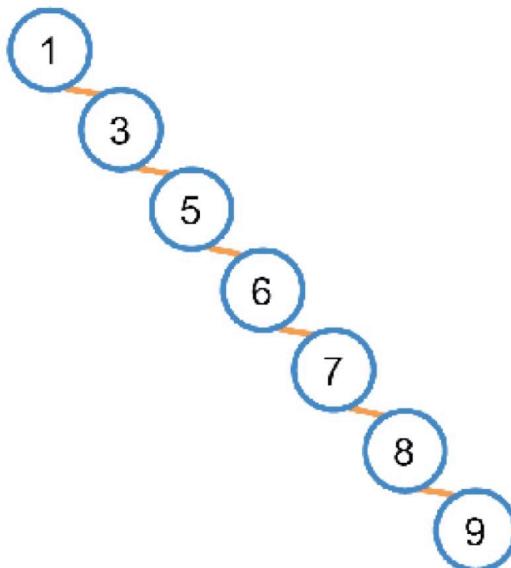


Figura 9.10 BST completamente desbalanceado.

En 1962, dos matemáticos rusos de apellidos Adelson-Velakii y Landis, idearon una forma de insertar y eliminar elementos en un árbol BST, de tal forma que se mantuviera el balance dentro de ciertos límites. A este nuevo tipo de árbol les llamaron AVL (por las siglas de sus apellidos).

Para definir un árbol AVL es necesario definir Factor de Equilibrio (FE) de la siguiente manera:

$$\text{FE} = \text{altura del hijo derecho} - \text{altura del hijo izquierdo}$$

Se dice que un BST está balanceado cuando el FE de todos sus nodos cumple la condición:

$$|\text{FE}| \leq 1$$

El BST de la figura 9.11 está balanceado. Se muestran sus FE de cada nodo y se puede observar que en todos los casos se cumple la condición anterior.

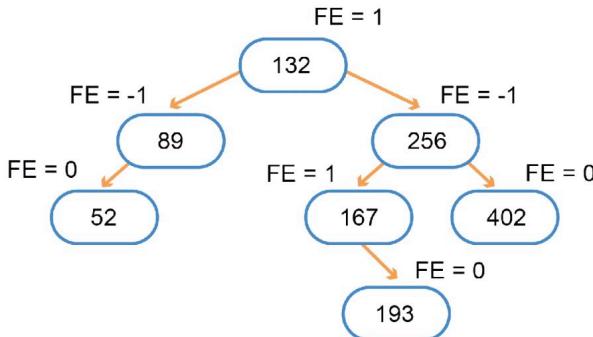


Figura 9.11 BST balanceado.

Para el árbol de la figura 9.11 podemos calcular los FE de todos sus nodos para comprobarlos. Por ejemplo, calculemos el del nodo 256. La altura de su hijo izquierdo es 2, la de su hijo derecho es 1, por lo que su FE =  $1 - 2 = -1$ . El del nodo 193, que es una hoja, es  $FE = 0 - 0 = 0$ . El del nodo 132, que es la raíz, es  $FE = 3 - 2 = 1$ , y así sucesivamente. Una posible implementación de un AVL en C++ es la siguiente:

```

1. ///
2. // Implementación de la clase AVL //
3. ///
4.
5. #include <iostream>
6.
7. using namespace std;
8.
9. class NodoArbolAVL{
10. public:
11. int info;
12. int fe;
13. NodoArbolAVL *hijoIzquierdo;
14. NodoArbolAVL *hijoDerecho;
15.
16. NodoArbolAVL(int dato){
17. info = dato;
18. fe = 0;
19. hijoIzquierdo = nullptr;
20. hijoDerecho = nullptr;
21. }
22. };
23.
24. class AVL{
25. private:
26. NodoArbolAVL *raiz;
27. public:
28. AVL(){
29. raiz = nullptr;
30. }
31. ...
32. };

```

Se puede observar que el ADT de un árbol AVL es muy similar al del BST, solo que ahora se requiere que cada nodo mantenga el valor de su factor de equilibrio (línea 12). Lo único que cambia es la clase que define al nodo del AVL, al cual llamaremos ahora **NodoArbolAVL** (línea 9).

Para poder mantener el BST balanceado después de eliminar o insertar un elemento, sus creadores idearon unas operaciones básicas nuevas llamadas “rotaciones”, las cuales ayudan a restaurar el balance del BST, si es que lo ha perdido.

### 9.3.1 Rotaciones básicas

Si al insertar o eliminar un elemento de un BST se ha perdido el balance, se utilizan las rotaciones, las cuales son movimientos sobre los nodos y sus apuntadores, basados en la raíz del árbol o subárbol.

Existen dos tipos de rotaciones básicas:

- **Rotación derecha:** se realiza sobre la raíz, quedando como nueva raíz el hijo izquierdo de la raíz original, además de reacomodar los nodos.
- **Rotación izquierda:** se realiza sobre la raíz, quedando como nueva raíz el hijo derecho de la raíz original, además de reacomodar los nodos.

Rotación derecha

1. Recordar el valor del hijo izquierdo de la raíz: **temp = raíz.izquierdo**
2. Colocar el valor de **raíz.izquierdo** al valor de **temp.derecho**
3. Colocar **temp.derecho** a la raíz

#### 4. Colocar la raíz a **temp**

Como ejemplo realizamos rotación derecha sobre el árbol de la figura 9.12(a).

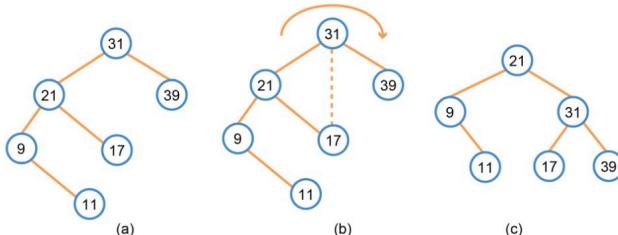


Figura 9.12 (a) árbol original, (b) indicación de movimientos para hacer la rotación derecha, (c) árbol después de la rotación derecha.

La rotación derecha se realiza sobre el nodo 31, que es la raíz. La nueva raíz será el hijo izquierdo del nodo 31, que es el nodo 21. El hijo derecho del nodo 21, si es que tiene, pasa a ser el hijo izquierdo del nodo 31; el nodo 31 pasa a ser el hijo derecho del nodo 21.

Estos movimientos se indican en la figura 9.12(b) y el árbol final, después la rotación derecha, se muestra en la figura 9.12(c).

#### Rotación izquierda

1. Recordar el valor del hijo izquierdo de la raíz: **temp = raíz.derecho**
2. Colocar el valor de **raíz.derecho** al valor de **temp.izquierdo**
3. Colocar **temp.Izquierdo** a la raíz
4. Colocar la raíz a **temp**

Como ejemplo, hagamos una rotación derecha sobre el árbol de la figura 9.13(a).

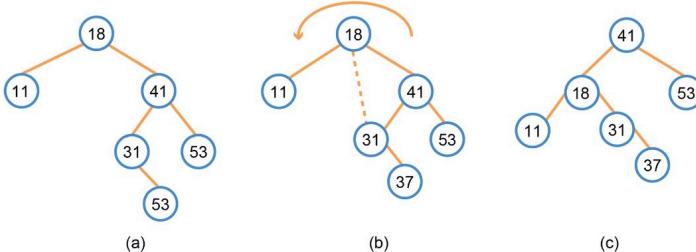


Figura 9.13 (a) árbol original, (b) indicación de movimientos para hacer la rotación izquierda, (c) árbol después de la rotación izquierda.

La rotación izquierda la realizamos sobre el nodo 18. El hijo de recho del nodo 18 debe ser ahora el hijo derecho, si es que tiene, del nodo 41. El nodo 41 pasa a ser la nueva raíz, y el nodo 18 pasa a ser el nuevo hijo izquierdo del nodo 18.

Todos estos movimientos son indicados en la figura 9.12(b) y el árbol final, después de hacer la rotación izquierda, se muestra en la figura 9.13(c).

### 9.3.2 Rotaciones en árboles AVL

Las inserciones y las eliminaciones en un AVL se hacen de la misma forma que en un BST. La única situación es que, si después de hacer la operación el AVL queda desbalanceado, se deben hacer operaciones adicionales para recuperar el balance. Estas operaciones especiales son rotaciones y están basadas en las rotaciones básicas anteriormente descritas.

Si después de hacer una inserción o una eliminación en el árbol AVL, para algún nodo se cumple la condición de que  $|FE| > 1$ , es decir, el nuevo árbol está balanceado. Se debe realizar una de las dos siguientes operaciones de rotación:

- Rotación simple

- Rotación compuesta

Ambas rotaciones están basadas en las rotaciones básicas explicadas anteriormente.

### *Rotación simple*

La rotación simple presenta dos casos: el primero en donde están involucrados dos hijos derechos (D-D) y el segundo, en donde están involucrados dos hijos izquierdos (I-I). Los dos se detectan analizando los FE de los nodos, a partir del que resulta desbalanceado.

En el caso D-D, uno de los nodos tiene un FE = 2, lo cual indica que se ha perdido el equilibrio en su rama derecha. Su hijo derecho tiene un FE=1, lo cual nos dice que la rotación debe involucrar a su rama derecha. Después de efectuar la rotación, el nodo tiene como hijo a su padre. El caso I-I tiene las mismas operaciones, pero con diferentes ligas.

Con un ejemplo quedará más claro. Veamos el caso D-D para el árbol de la figura 9.14(a).

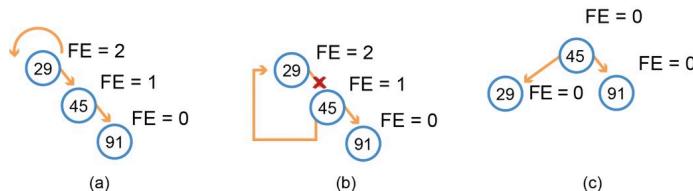


Figura 9.14 (a) árbol original desbalanceado, (b) indicaciones de las rotaciones que se deben realizar, (c) árbol final balanceado, después de hacer las operaciones.

El nodo 29 tiene un FE=2 lo que indica un desbalance. Su hijo derecho tiene un FE=1; eso indica que estamos en el caso D-D. Es claro que este caso se resuelve con solo una rotación izquierda, como las básicas que se vieron anteriormente.

La figura 9.14(b) indica las operaciones que se deben hacer con los apuntadores para hacer la rotación, las cuales son equivalentes a las explicadas en la sección anterior, dando como resultado el árbol final de la figura 9.14(c), el cual se puede ver que ya está balanceado.

Ahora un ejemplo de caso I-I, con el árbol que se muestra en la figura 9.15(a).

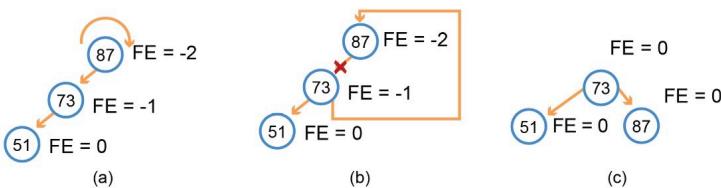


Figura 9.15 (a) Árbol original desbalanceado, (b) indicaciones de las rotaciones que se deben realizar, (c) árbol final balanceado, después de hacer las operaciones.

El caso se reconoce porque un nodo tiene  $FE=-2$ , lo que implica que tiene problemas con su rama izquierda, y su hijo izquierdo tiene  $FE=-1$ . Este caso se resuelve con una rotación simple sobre el nodo raíz, que es el que tiene el desbalance.

Las operaciones que se realizan con los apuntadores están indicadas en la figura 9.15(b), dando como resultado el árbol final balanceado de la figura 9.15(c).

### *Rotación compuesta*

Es una operación que está compuesta por dos rotaciones seguidas. Con estas operaciones se afectan las ligas de tres nodos.

También tiene dos casos: el primero se presenta cuando están involucrados un hijo derecho y un hijo izquierdo (D-I). El segundo, cuando están involucrados un hijo izquierdo y uno derecho (I-D).

En el caso D-I, uno de los nodos tiene un FE = 2, lo que implica que se ha perdido el equilibrio en su rama derecha. Su hijo tiene un FE= -1, que significa que la rotación debe involucrar a su rama izquierda. Después de la rotación el nodo final tiene como hijos a los otros dos.

En otras palabras, este caso se resuelve con una rotación derecha y luego con una rotación izquierda, de ahí su nombre. El caso I-D, requiere las mismas operaciones, pero con diferentes nodos y ligas. Este caso se resuelve con una rotación izquierda y una derecha. Veamos un ejemplo del caso D-I, con el árbol de la figura 9.15(a).

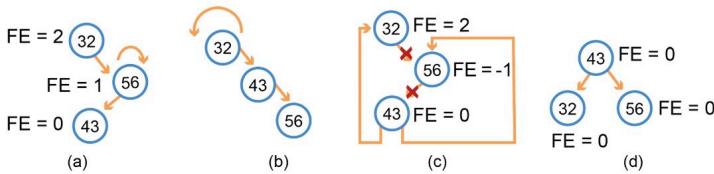


Figura 9.15 Caso D-I (a) árbol original desbalanceado, (b) árbol después de la primera rotación (derecha), (c) movimientos de las dos rotaciones en un solo proceso, (d) árbol final balanceado, después de la segunda rotación (izquierda).

El caso se detecta porque el nodo raíz tiene un FE=2, lo que indica que está desbalanceado y que su hijo derecho tiene FE=-1. Se hace primero una rotación izquierda sobre el nodo 56, como la que se indica en la figura 9.15(a), quedando el árbol todavía desbalanceado, de la figura 9.15(b).

Se hace una rotación izquierda sobre el nodo 32, como la que se indica en la figura 9.15(b), quedando el árbol final balanceado de la figura 9.15(d). La figura 9.15(c) muestra las dos rotaciones en un mismo proceso. Un ejemplo del caso I-D, se puede ver con el árbol de la figura 9.16(a).

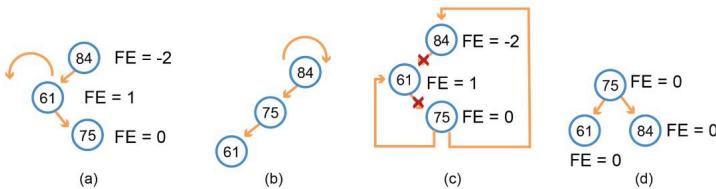


Figura 9.16 Caso I-D (a) árbol original desbalanceado, (b) árbol después de la primera rotación (izquierda), (c) movimientos de las dos rotaciones en un solo proceso, (d) árbol final balanceado, después de la segunda rotación (derecha).

El caso I-D se detecta porque el nodo raíz tiene un FE=-2, lo que indica que está desbalanceado y que su hijo izquierdo tiene FE=1. Se hace primero una rotación izquierda sobre el nodo 61, como la que se indica en la figura 9.16(a), por lo que el árbol queda todavía desbalanceado, de la figura 9.16(b).

Se hace una rotación derecha sobre el nodo 84, como la que se indica en la figura 9.15(b), por lo que el árbol final queda balanceado de la figura 9.16(d). La figura 9.16(c) muestra las dos rotaciones en un mismo proceso.

En resumen, hay cuatro tipos de árboles no balanceados críticos, que nos dan 4 casos, los cuales se reconocen y resuelven de la siguiente manera:

- Caso I-I (papá -2, hijo izquierdo -1): rotar a la derecha alrededor del padre.
- Caso I-D (papá -2, izquierdo +1): rotar a la izquierda alrededor del hijo y luego rotar a la derecha alrededor del padre.
- Caso D-D (papá +2, derecho +1): rotar a la izquierda alrededor del padre.

- Caso D-I (papá +2, derecho -1): rotar a la derecha alrededor del hijo y luego a la izquierda alrededor del padre.

Existen dos casos más que solo aparecen en la operación de eliminación, pero son sumamente raros. Estos se resuelven de forma similar a los ya descritos y son:

- D-D (papá 2, hijo 0)

- I-I (papá -2, hijo 0)

Es necesario comentar que en la inserción cuando el árbol queda desbalanceado después de hacer una rotación, el árbol queda completamente balanceado. Sin embargo, en el caso de la eliminación, algunas veces, para balancear el árbol, pueden ser necesarias más de una rotación, desde donde se presenta el primer desbalance hacia arriba, hasta llegar a la raíz.

Todos estos casos se pueden aplicar de la misma forma con árboles más grandes. Un ejemplo de un caso I-I, con un árbol mayor, se presenta en la figura 9.17.

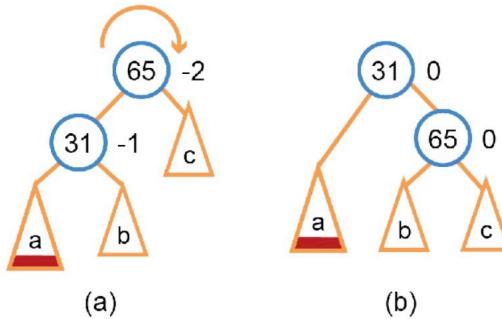


Figura 9.17 (a) Árbol original desbalanceado en caso I-I. (b) Árbol final balanceado después de efectuara la rotación derecha. La parte roja es la que ocasiona el desbalance.

En la figura 9.17(a) se puede ver que el papá tiene  $FE=-2$ , y su

hijo izquierdo  $FE=-1$ , lo que nos lleva al caso I-I. Este caso se resuelve con una rotación simple a la izquierda, quedando el árbol balanceado de la figura 9.17(b). La parte roja es la que ocasionó el desbalance y se puede observar que después de la rotación, quedó al mismo nivel que el resto del árbol.

Un ejemplo de una rotación compuesta se puede observar en la figura 9.18.

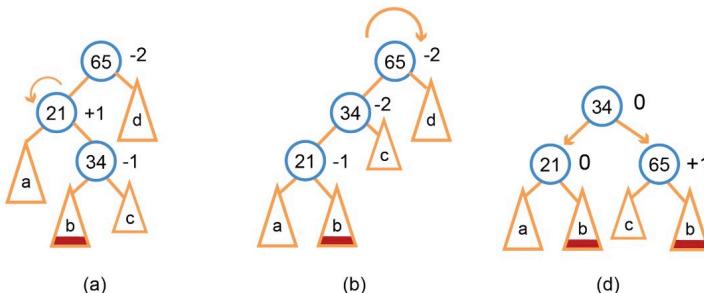


Figura 9.18 (a) Árbol original desbalanceado de caso I-D. (b) Árbol después de la primera rotación (izquierda). (c) Árbol final balanceado.

En la figura 9.18(a) se observa un árbol desbalanceado. Se reconoce que es del caso I-D porque el papá tiene un  $FE=-2$  y su hijo izquierdo un  $FE=1$ . Este caso se resuelve con una rotación izquierda sobre el nodo 21, la cual está indicada en la figura 9.18(a), lo que da como resultado el árbol, todavía desbalanceado, mostrado en la figura 9.18(b).

Como se indica en la misma figura 9.18(b), se debe realizar una rotación derecha sobre el nodo 65, dando como resultado el árbol balanceado de la figura 9.18(c). Se puede observar que la parte roja es la que produce el desbalance en la figura 9.18(a), después de las rotaciones queda básicamente al mismo nivel que el resto del árbol, como se observa en la figura 9.18(c).

La implementación de estos árboles AVL debe seguir los algo-

ritmos explicados, llamándolos de acuerdo con el caso de que se trate. Además, debe detectar cada uno de los casos explicados. Este balanceo de árbol se debe llamar cada vez que se detecta que se produjo un desbalanceo provocado por la inserción o eliminación de algunos elementos.

Ciertamente, la implementación completa se complica un poco y requiere de un código algo extenso, por lo que queda fuera del alcance de este libro.

## 9.4 Árboles HEAP

**E**n el nivel lógico o abstracto, un **HEAP** es un árbol binario que cumple con las siguientes reglas:

- Todo nodo padre del árbol contiene un elemento que tiene un valor de mayor prioridad que los valores de sus nodos hijos. La prioridad está determinada por la aplicación, pudiendo ser valor mayor o valor menor. En la figura 9.19 se aprecian dos ejemplos, uno que es correcto y otro que no.

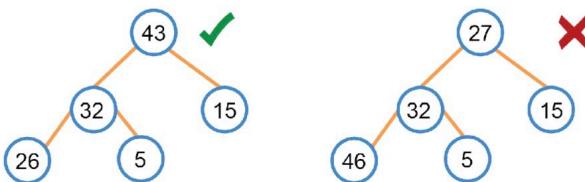


Figura 9.19 Ejemplo de árbol HEAP con la regla de mayor prioridad del padre.

- El árbol debe de estar completamente balanceado. Todos los niveles del árbol tienen el máximo de nodos posible, excepto el nivel inferior que puede estar incompleto
- Los nodos hojas del nivel inferior están lo más a la izquierda

possible. La figura 9.20 muestra dos ejemplos, uno correcto y uno incorrecto sobre estas dos últimas reglas.

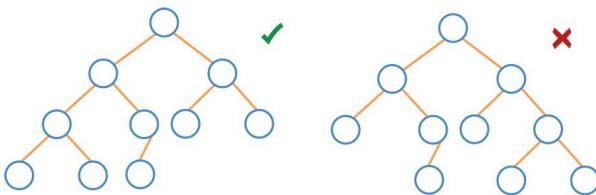


Figura 9.20 Ejemplo de árbol HEAP con la regla de hojas de último nivel a la izquierda.

Aunque lógicamente un **HEAP** es un árbol binario, dadas sus características y las aplicaciones en que se utiliza, la representación de **HEAP** puede realizarse en un arreglo unidimensional (sin ocupar la celda 0). La figura 9.21 muestra un ejemplo de la representación de un árbol HEAP en un arreglo unidimensional.

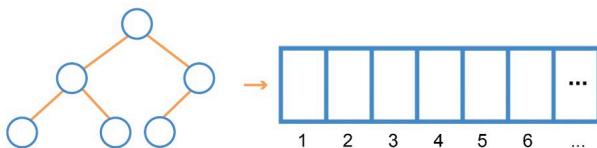


Figura 9.21 Representación de un HEAP en un arreglo unidimensional.

Dado que un **HEAP** es un árbol binario completo en el arreglo, el elemento en la posición “k” representa un nodo en el árbol cuyos nodos hijos estarán en las posiciones  **$2*k$  y  $2*k+1$**  del arreglo. La figura 9.22 muestra como quedaría un **HEAP** almacenado en un arreglo y se puede apreciar que la raíz esta en la posición 1 y sus hijos en la 2 y 3 ( $2*1$  y  $2*1+1$ ), por ejemplo el 21 esta en la posición 2 y sus hijos en la posición 4 y 5 ( $2*2$  y  $2*2+1$ ).

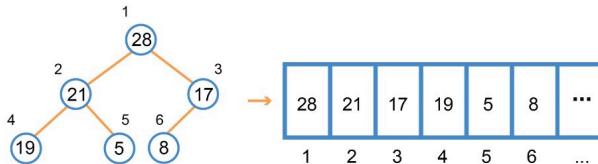


Figura 9.22 Representación de un HEAP en un arreglo unidimensional y su correspondencia.

### 9.4.1 Lista vs HEAP

Toda lista de datos ordenada es por si misma un árbol HEAP. Si la lista no esta ordenada, se puede convertir en un HEAP y de ahí obtener las aplicaciones correspondientes.

Inicialmente se puede considerar que todos los nodos hoja del árbol HEAP cumplen con la regla del HEAP por default, ya que estos no tienen hijos donde se localizan los nodos hojas, si la cantidad de nodos es par, son los  $n/2$  posiciones de la derecha del arreglo, si la cantidad de nodos es impar son los  $n/2 + 1$  nodos de la derecha del arreglo. La figura 9.23 muestra un ejemplo de la localización de los nodos hoja.

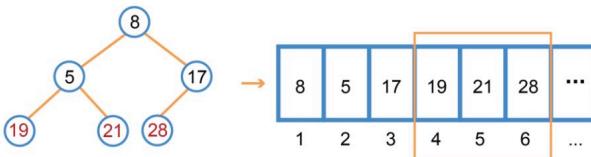


Figura 9.23 Localización de los nodos hoja dentro del arreglo.

Al continuar analizando los nodos padre del nivel inferior hacia arriba, y de derecha a izquierda, se reacomodan hacia abajo los valores de los nodos descendientes para cumplir la regla del **HEAP**. Tomar la primera mitad del arreglo del subíndice más grande hacia el más pequeño.

La figura 9.24 muestra como debe de irse dando esta transi-

ción.

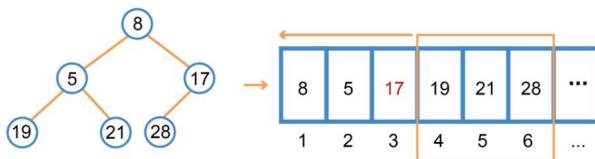


Figura 9.24 Transición para la generación de un HEAP dado una lista.

El algoritmo de reacomodo hacia abajo es el siguiente para cada nodo padre:

- Comparar el valor del nodo con los nodos hijos.
- Si el valor del padre es de mayor prioridad, el proceso de reacomodo para ese elemento aquí termina.
- Si el valor de algún hijo es de mayor prioridad, se intercambia este con el padre, y se repite el proceso de reacomodo hacia abajo con los hijos del elemento intercambiado.

Ejemplo:

Se desea, dada la lista de elementos: 8, 5, 17, 19, 21 y 28, generar un HEAP con prioridad de valor mayor. La figura 9.25 muestra paso a paso el procedimiento para realizarse.

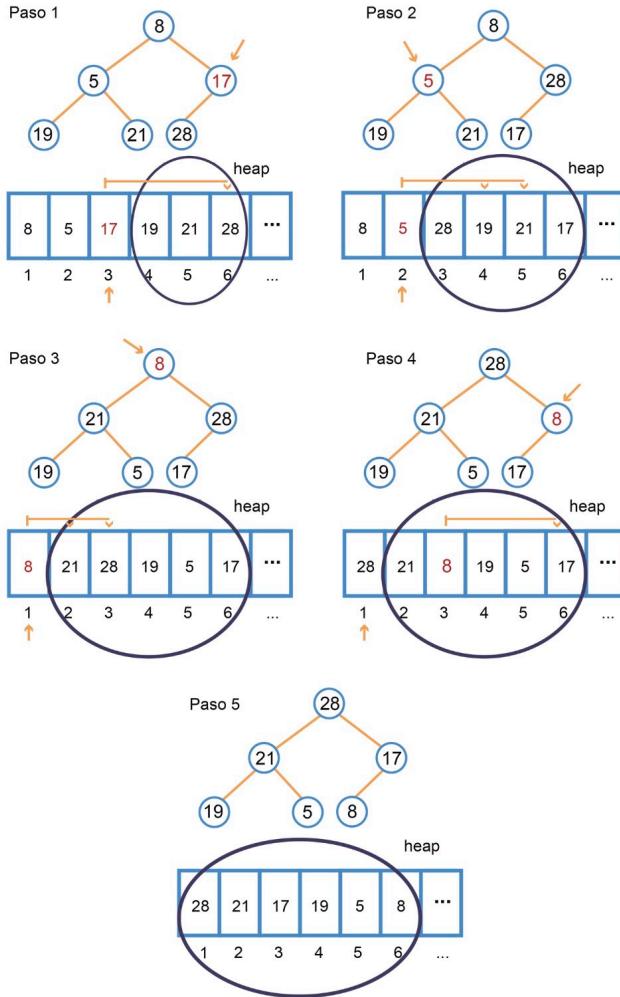


Figura 9.25 Ejemplo paso a paso de la generación de un HEAP dado una lista.

### 9.4.2 Fila priorizada en un árbol HEAP

Una cola priorizada puede ser representada en un árbol HEAP, ya que la prioridad de la fila será exactamente la misma que la

del HEAP. La fila priorizada tiene dos principales operaciones: agregar (**push**) y sacar (**pop**), las cuales analizaremos a continuación.

### *Sacar un elemento de la fila priorizada (pop)*

Las colas priorizadas requieren dar de baja el elemento de mayor prioridad cuando realizan la operación de **pop**, esto quiere decir que se requiere dar de baja la raíz del HEAP (el primer elemento del arreglo). Esto se realiza de la siguiente forma:

- Se toma el valor de la última hoja del HEAP, es decir el último elemento del arreglo.
- Se intercambia este valor con el que está en la raíz.
- Físicamente, se elimina al nodo que representa la última hoja.
- Por último, se reacomoda hacia abajo al elemento que quedó como raíz del HEAP. Esto equivale a ejecutar el último paso de la construcción del HEAP.

La figura 9.26 muestra un ejemplo de eliminación de la raíz de un HEAP. Representando la operación de pop de una fila priorizada.

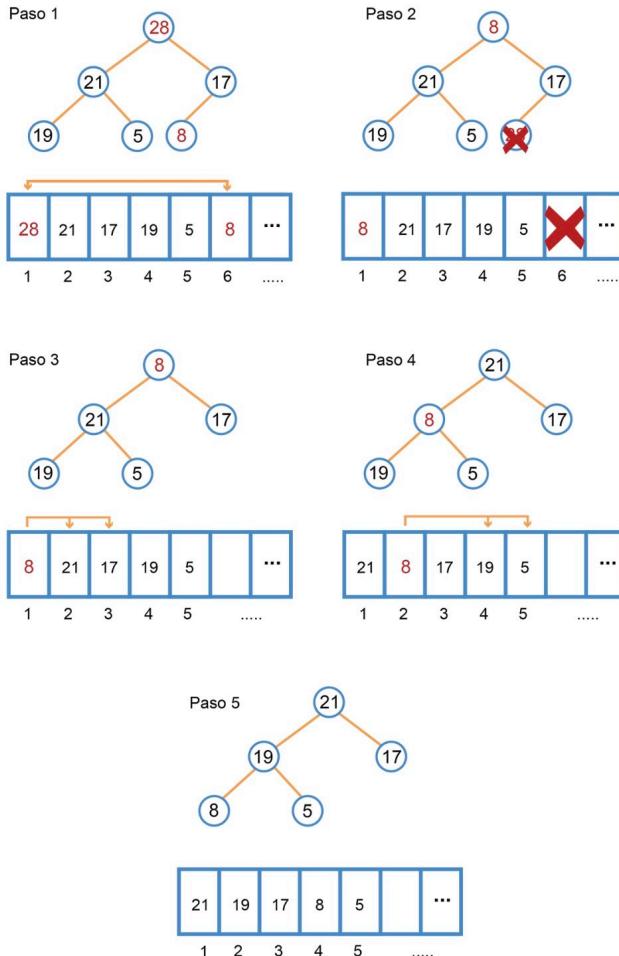


Figura 9.26 Ejemplo paso a paso la operación de pop de una fila priorizada.

### Agregar un elemento a la fila priorizada (push)

Las colas priorizadas requieren dar un elemento y este tiene que ser acomodado según la prioridad que tenga la fila, mediante la operación de **push**. Esto se realiza insertando el nuevo elemento como la última hoja del **HEAP**, esto es, como el último

elemento del arreglo. Sin embargo, se tiene que reacomodar este nuevo elemento con respecto al valor de los ancestros, esto se realiza de la siguiente forma:

- Se inserta el nuevo elemento como el último elemento del arreglo.
- Compara este nuevo elemento con respecto a su padre, si el elemento fue insertado en la posición  $k$ , el padre estará en la posición  $k/2$ .
- Si el valor del padre tiene menor prioridad, se intercambia y se sigue comparando hacia arriba.
- Si el valor del padre tiene mayor prioridad, el proceso ahí termina.

La figura 9.27 muestra un ejemplo de la inserción del elemento 20 en un HEAP, representando la operación de push de una fila priorizada.

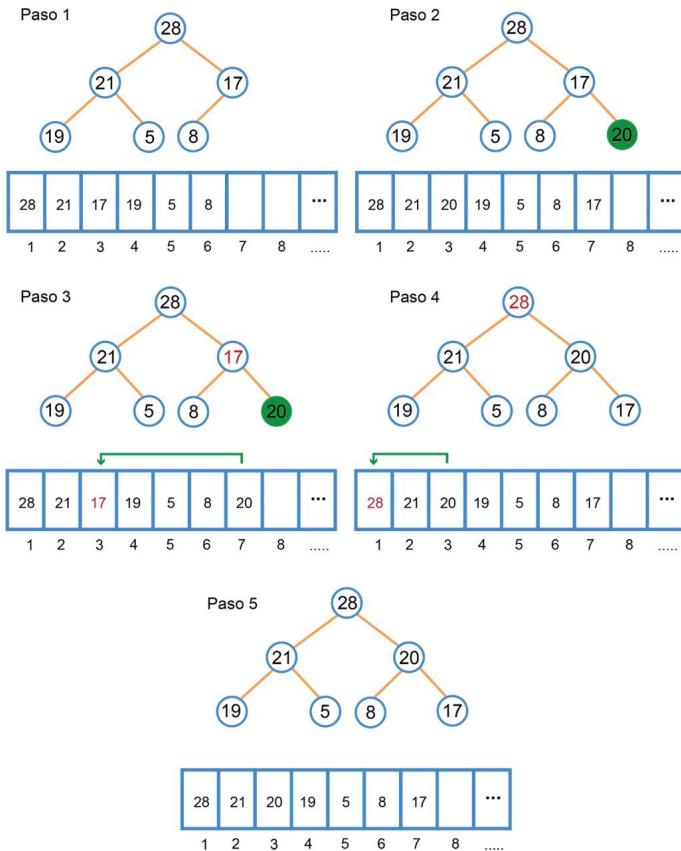


Figura 9.27 Ejemplo paso a paso la operación de push (20) de una fila priorizada.

## 9.5 Trie

**E**l trie es una estructura de datos muy útil para resolver problemas relacionados con **strings**, es también conocido como árbol de prefijos. El nombre proviene de la palabra **RETRIEVAL**, que significa recuperación. Internamente funciona como un árbol ordenado, en donde cada nodo representa un carácter de una palabra insertada en el **trie**, y los nodos hijos

de este son caracteres que aparecen después de él en la palabra.

La raíz del árbol es el carácter vacío, ya que se puede considerar que aparece al principio de cualquier palabra. En la figura 9.28 se puede apreciar la estructura de datos trie generada con las palabras: loco, loca, lapiz, poca y pozo.

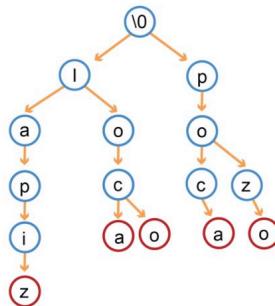


Figura 9.28 Ejemplo de un trie.

Al construir un trie se requiere almacenar en cada nodo lo siguiente:

- letra que representa,
- si es o no la última letra de una palabra,
- nodo padre y
- nodos hijos.

Además, se requieren dos métodos más:

- getChild: para saber si un nodo es hijo de ese nodo
- addChild: para agregar un nuevo hijo al nodo.

El constructor del trie quedaría de la siguiente forma:

```

1. class trie
2. {
3. public:
4. root = new nodeTrie('\0', null);
5. void insert(string word);
6. bool search(string word);
7.
8. private:
9. nodeTrie root;
10. };

```

El nodo del trie quedaría de la siguiente forma:

```

1. class nodeTrie
2. {
3. public:
4. nodeTrie(char c, nodeTrie p){ parent = p; letter = c; }
5. nodeTrie getChild(char c) { return child[c-'a']; }
6. nodeTrie addChild(nodeTrie node) { child[node.content - 'a'] = node; }
7.
8. private:
9. char letter;
10. bool end;
11. nodeTrie parent;
12. nodeTrie[] child = new nodeTrie[26];
13. };

```

La función de insert del trie quedaría de la siguiente forma:

```

1. bool trie::search(string word){
2. nodeTrie current = root;
3. for(int i=0; i < word.length(); i++){
4. char c = word[i];
5. nodeTrie sub = current.getChild(c);
6. if (sub == nullptr){
7. return false;
8. }
9. current = sub;
10. }
11. }

```

## 9.6 Splay Tree

**S**play Tree es una estructura de datos jerárquica que emplea rotaciones para mover cualquier nodo o clave accesada en la búsqueda o inserción a la raíz. Dejando a los nodos recién insertados cerca de la raíz para hacer más eficiente la búsqueda de ellos que se realicen posteriormente. Así, la forma del árbol va cambiando dependiendo de los accesos de las claves.

En el caso de los árboles splay se lleva el elemento buscado o insertado a la posición de la raíz. En la búsqueda, o inserción **bottom-up**, se realiza un recorrido desde la raíz hasta encontrar este elemento, o bien, hasta encontrar una hoja en caso de la

inserción; después se realiza una operación **splay** para mover el elemento a la posición de la raíz.

La operación **splay**, consiste de una secuencia de dobles rotaciones, hasta que el nodo quede un nivel debajo de la raíz; en este caso basta una rotación simple para completar la operación.

En cada operación **splay** se asciende el nodo en uno o dos niveles, dependiendo de su orientación relativa respecto de su nodo abuelo. Hay tres casos:

- Zig: el nodo es un hijo izquierdo o derecho (Zag) de la raíz y sin abuelo. La figura 9.29 muestra un ejemplo de la rotación de Zig.

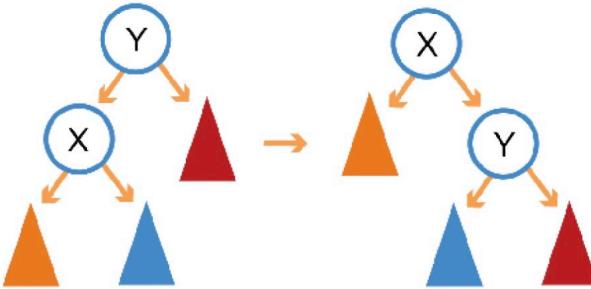


Figura 9.29 Rotación Zig.

- Zig-Zag: El nodo es un hijo izquierdo de un hijo derecho; o un hijo derecho de un hijo izquierdo (Zag-Zig). La figura 9.30 muestra un ejemplo de la rotación Zig-Zag.

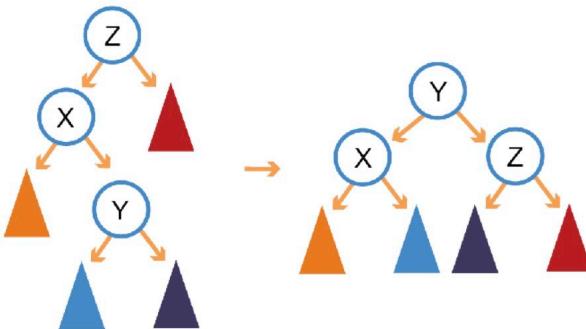


Figura 9.30 Rotación Zig-Zag.

- **Zig-Zig:** El nodo es un hijo izquierdo de un hijo izquierdo; o un hijo derecho de un hijo derecho (Zag-Zag). La figura 9.31 muestra un ejemplo de la rotación Zig-Zig.

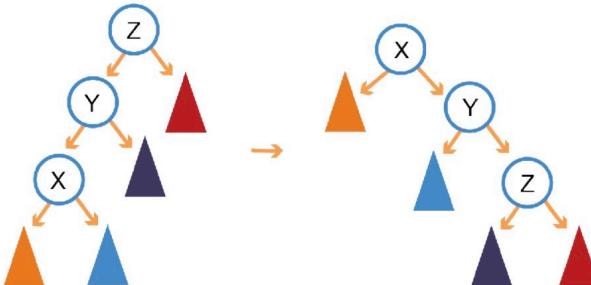


Figura 9.31 Rotación Zig-Zig.

Debe notarse que mover un nodo hacia la raíz, siguiendo la forma inversa de la trayectoria de búsqueda desde la raíz hasta el nodo, no es equivalente a las dobles rotaciones propuestas en árboles slay.

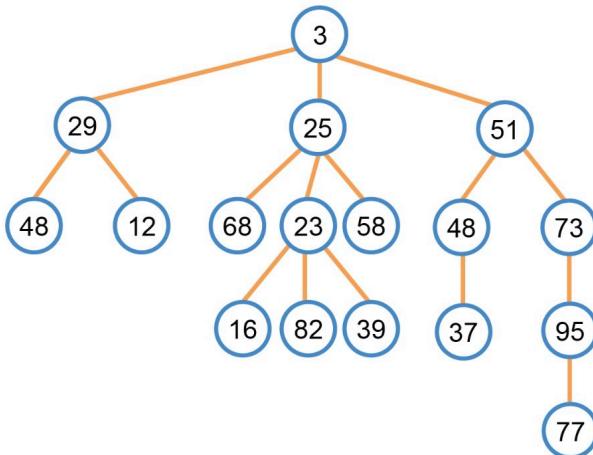
Las operaciones **Zig**, **Zag**, **Zig-Zag** y **Zag-Zig**, son equivalentes a mover hacia la raíz; la diferencia está en las operaciones **Zig-Zag** y **Zag-Zig**. En el caso de mover hacia la raíz, se rota el padre del nodo a la derecha y luego el nuevo padre del nodo a la derecha.

Medir el efecto de estos dos tipos de rotaciones requiere un análisis de costos denominado “Análisis Amortizado”. Se puede verificar el costo amortizado de  $n$  operaciones splay sobre un árbol con  $n$  nodos es de:  $O((m+n) * \log_2 (n+m))$ . Es con este fundamento que se eligen las dobles rotaciones en este tipo de árboles, y tienden a acortar la altura del árbol.

## Ejercicios



1. Para el siguiente árbol:



- a. Diga cuál es la raíz.
- b. De dos ejemplos de nodos hoja.
- c. De dos ejemplos de nodos intermedios.
- d. Escriba los nodos que pertenecen a la rama que llega al nodo 37.
- e. Escriba los hijos del nodo 25.
- f. Escriba los nodos hermanos del nodo 48.
- g. Escriba el nodo padre del nodo 82.
- h. ¿Cuál es el grado del árbol?
- i. ¿Cuál es el grado del nodo 51?

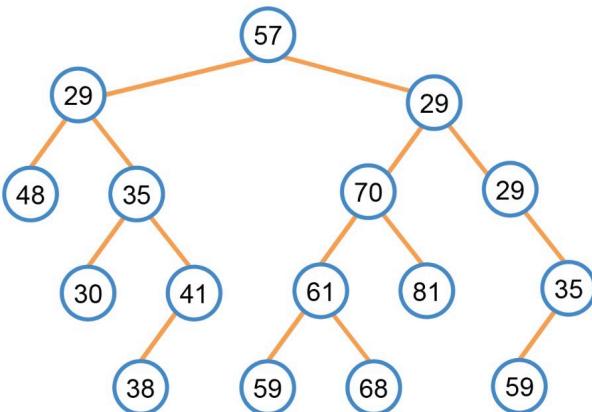
j. ¿Cuál es la altura del nodo 16?

k. ¿Cuál es la altura del árbol?

l. ¿Cuál es el nivel del nodo 3?

m. ¿Cuál es el nivel del nodo 58?

2. Para el siguiente árbol:



a. Diga si es un BST y explique la razón.

b. Dibuje el árbol que resulta después de insertar los números: 10 y 90, en ese orden.

c. Dibuje el árbol que resulta después de eliminar el número 84.

d. Dibuje el árbol que resulta después de eliminar los números: 38, 70 y 57, en ese orden.

e. Haga el recorrido de los nodos en preorden.

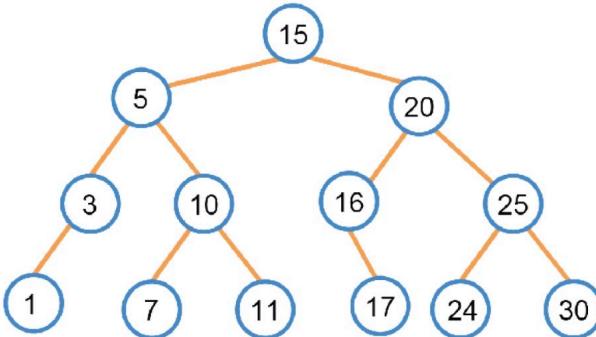
f. Haga el recorrido de los nodos en inorder.

g. Haga el recorrido de los nodos en posorden.

h. Calcule los FE de cada nodo y diga si es un AVL.

**3.** Dibuje el árbol AVL que resulta al insertar los siguientes números, en el orden indicado: 3, 5, 7, 9, 11, 10 y 12.

**4.** Para el siguiente árbol AVL:



**a.** Dibuje el árbol que resulta al eliminar en forma sucesiva, los siguientes números: 3, 1, 15.

**b.** Dibuje el árbol que resulta al agregar en forma sucesiva, los siguientes números: 23, 22, 21.

**5.** El siguiente arreglo representa un Árbol Heap, dibuja cómo se vería el árbol.



**6.** ¿Cómo quedaría el árbol y su representación en el arreglo después de borrar el 100, dado los siguientes datos?



**7.** Dadas las siguientes listas de datos, ¿cómo quedaría el Árbol Heap y su representación en el arreglo?

**a.** 14, 10, 8, 25, 73, 5, 43, 17, 43, 23

**b.** 1, 5, 2, 8, 7, 9, 10, 3, 6, 4



# Capítulo 10. Estructuras de datos de red (Grafos)

10

Las estructuras de tipo red, mantienen los datos interconectados en una relación de muchos a muchos. En muchas aplicaciones y problemas, se requiere tener los datos interrelacionados con múltiples enlaces que nos permitan su acceso por diferentes vías y conexiones, formando estructuras de este tipo.

## 10.1 Terminología de grafos

En las jerarquías de las organizaciones de estructuras de datos, los grafos son el caso más general que existe. Es una estructura tipo red donde se mantiene una relación de muchos a muchos (N:M) entre sus elementos.

Una red de carreteras entre las diferentes ciudades del país es una es un ejemplo de un grafo. La figura 10.1 nos muestra un ejemplo gráfico de un grafo.

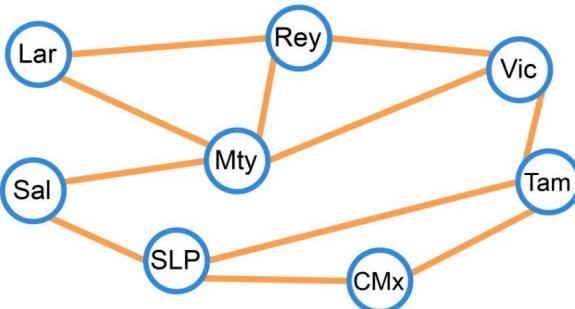


Figura 10.1 Ejemplo gráfico de un grafo

Un grafo se puede describir además como un conjunto de nodos y arcos.

- **Nodo:** los nodos, también llamados vértices (**vertex** en inglés), son el elemento básico de información de un grafo.
- **Arco:** los arcos, también llamados aristas (**edges** en inglés), son la conexión entre dos nodos, estableciendo la relación entre dos elementos del grafo.

Dentro de la figura 10.1 el conjunto de nodos es:

$$V = \{Lar, Rey, Mty, Sal, Vic, SLP, Tam, CMx\}$$

Y el conjunto de arcos es:

$$E = \left\{ (Lar, Rey), (Lar, Mty), (Rey, Mty), (Mty, Sal), (Mty, Vic), (Vic, Tam), (Sal, SLP), (Tam, SLP), (Tam, CMx), (SLP, CMx) \right\}$$

- **Subgrafo:** un subgrafo de un grafo **G**, es un grafo cuyo conjunto de nodos es un subconjunto de los nodos del grafo **G** y el conjunto de arcos es un subconjunto de los arcos del grafo **G**.

Dentro de la figura 10.1 un subgrafo pudiera ser los conjuntos **V'** y **E'**:

$$V' = \{Lar, Rey, Mty\}$$

$$E' = \{(Lar, Rey), (Lar, Mty), (Rey, Mty)\}$$

La figura 10.2 muestra gráficamente los conceptos de nodo, arco y subgrafo.

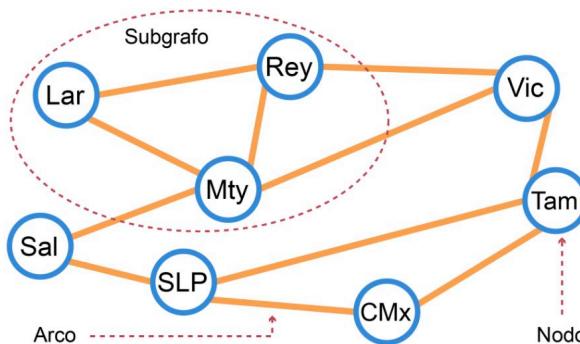


Figura 10.2 Ejemplo gráfico de nodo, arco y subgrafo

- **Nodos adyacentes:** cuando se tiene una conexión entre dos nodos **A** y **B**, se dice que estos (**A** y **B**) son adyacentes.
- **Vecinos de un nodo:** al conjunto de nodos que son adyacentes a un nodo **A** se dice que estos son vecinos del nodo **A**.

En la figura 10.2 se puede apreciar por ejemplo que **SLP** y **CMx** son nodos adyacentes, además que los vecinos de **Vic** son: {**Rey**, **Mty**, **Tam**}.

- **Camino:** un camino o trayectoria es una secuencia de nodos para ir de un nodo **A** a un nodo **B**, donde para cada par de nodos en secuencia, estos deben de ser adyacentes.
- **Trayectoria simple:** una trayectoria simple es aquel camino para ir del nodo **A** al nodo **B**, en donde todos los nodos contenidos en el camino son distintos.

En la figura 10.2 un camino simple para ir de **Rey** a **SLP** es: **Rey** → **Vic** → **Tam** → **CMx** → **SLP**.

- **Grafo no-dirigido:** un grafo no-dirigido es aquel donde los

arcos no tienen una dirección, esto es que si se tiene un arco (**A,B**) este puede ir tanto de **A** a **B** como de **B** a **A**. La figura 10.3 muestra un ejemplo de grafo no-dirigido.

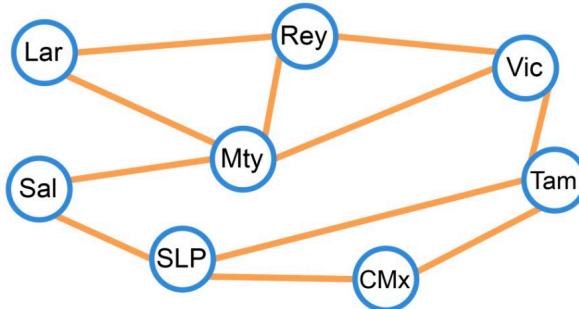


Figura 10.3 Grafo no-dirigido

- **Grafo dirigido:** un grafo dirigido es aquel donde los arcos tienen una dirección, esto significa que, si se tiene un arco (**A,B**) este solo puede ir de **A** a **B** y no puede ir de **B** a **A**. La figura 10.4 muestra un ejemplo de grafo dirigido.

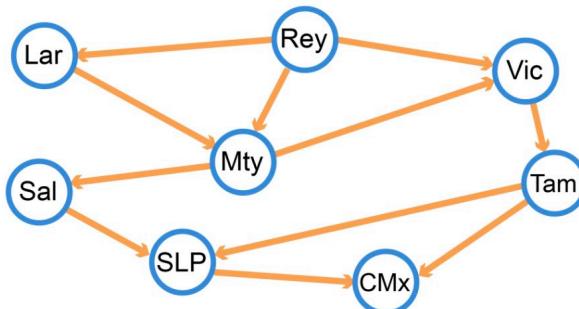


Figura 10.4 Grafo dirigido

- **Grafo ponderado:** cuando los arcos tienen un valor asociado, se dice que el grafo es un grafo ponderado. Típicamente este valor representa el valor del problema que se esté solucio-

nando: costo, distancia, kilómetros, gasolina, entre otros. La figura 10.5 muestra un grafo no dirigido y ponderado.

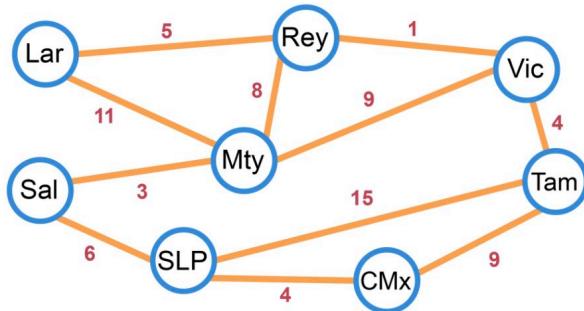


Figura 10.5 Grafo no-dirigido y ponderado

- **Ciclo:** un ciclo es una trayectoria donde el nodo de inicio y el de terminación es el mismo nodo. En la figura 10.5 Un ciclo puede ser: Rey → Vic → Mty → Rey.

## 10.2 Representación de un grafo

**U**n ADT Grafo debe contener al menos las siguientes operaciones:

- insertar un nodo,
- insertar un arco,
- borrar un nodo,
- borrar un arco,
- buscar un nodo y
- recorrer el grafo.

Para poder soportar estas operaciones se tiene que pensar en

cómo poder representar un grafo en el lenguaje de programación. Existen muchas formas de representar un grafo, las más comunes son:

- matriz de adyacencia,
- lista de adyacencia y
- lista de arcos.

### 10.2.1 Matriz de adyacencia

La representación de un grafo en una matriz de adyacencia sería tener una matriz de  $N \times N$ , en donde  $N$  es la cantidad de Nodos y cada celda de la matriz representaría si hay o no adyacencia entre el nodo del renglón con el nodo de la columna.

En la figura 10.6 se puede observar un Grafo No-Dirigido y No-Ponderado de 5 nodos y su representación en una matriz de adyacencia.

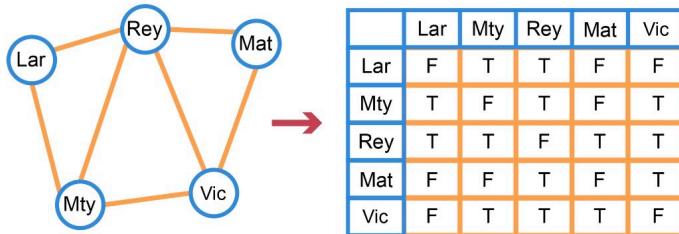


Figura 10.6 Grafo No-Dirigido y No-Ponderado en Matriz de Adyacencias

En caso de que el grafo fuera ponderado, en lugar de una matriz de booleanos sería una matriz de enteros, en donde se pondría la ponderación de ir saliendo del nodo del renglón al nodo de la columna. Cuando no exista conexión tradicionalmente se pone infinito y en la diagonal principal se pone 0.

Cuando el grafo es No-Dirigido, el triángulo superior de la matriz es un espejo del triángulo. En la figura 10.7 se puede observar un Grafo No-Dirigido y Ponderado de 5 nodos, su representación en una matriz de adyacencia.

La ventaja de representar un grafo en una matriz de adyacencia es que las operaciones de manipulación de arcos es muy fácil, pero se requiere saber el número exacto de nodos en un inicio y este no podrá ser modificado tan fácilmente.

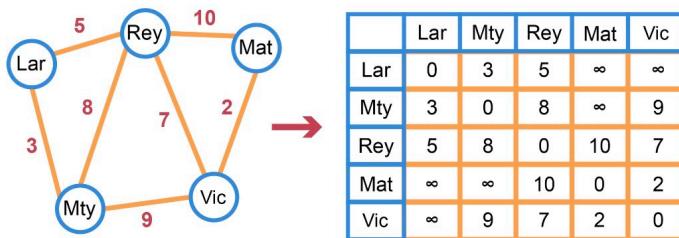


Figura 10.7 Grafo no-dirigido y ponderado en matriz de adyacencias.

### 10.2.2 Lista de adyacencias

La representación de un grafo en una lista de adyacencia sería tener por lado una lista de **N** nodos y por cada nodo una lista de los nodos con los que tiene adyacencia, y su ponderación en caso de que sea un grafo ponderado. En la figura 10.8 se pude observar un Grafo no-dirigido y ponderado de 5 nodos y su representación en una lista de adyacencia.

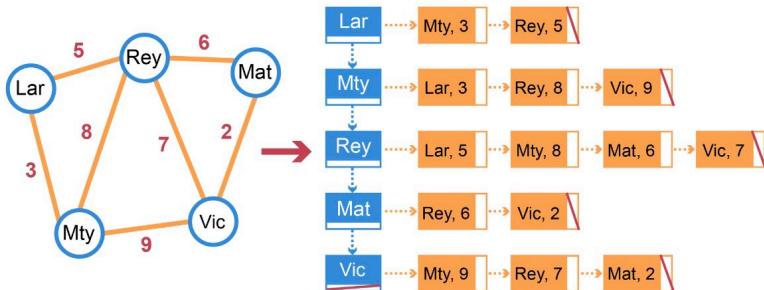


Figura 10.8. Grafo no-dirigido y ponderado en lista de adyacencias

La ventaja de representar un grafo en una lista de adyacencia es que no se requiere conocer con anterioridad la cantidad de nodos y arcos que conforman el Grafo, pero requiere más espacio de memoria debido al manejo de apuntadores.

### 10.2.3 Lista de arcos

La representación de un grafo en una lista de arcos es la más compleja de implementar. Por un lado, se tiene una lista de los nodos; por cada nodo hay 2 apuntadores, uno a una lista de arcos, en donde ese nodo es el origen del arco y otro apuntador a una lista arcos, en donde ese nodo es el destino del arco. En figura 10.9 se muestra un Grafo No-Dirigido y No-Ponderado de 5 nodos y 7 arcos en una representación de lista de arcos.

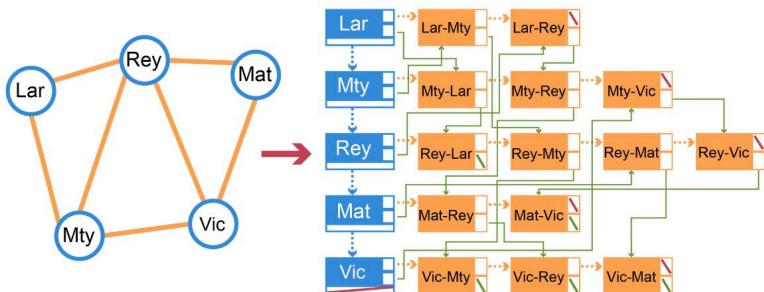


Figura 10.9 Grafo No-Dirigido y Ponderado en Lista de Arcos

La ventaja de representar un grafo en una lista de arcos es una representación bastante eficiente, pero requiere más espacio de memoria debido al manejo de apuntadores.

## 10.3 Recorridos de un grafo

**U**n recorrido sobre una estructura de datos es visitar a cada uno de los nodos una sola vez, en el caso de los **grafos** no existe un nodo inicial, por lo que se tiene que indicar el nodo sobre el cual iniciará el recorrido. Los recorridos más utilizados en los grafos son:

- BFS: Breadth First Search (primero en anchura).
- DFS: Depth First Search (primero en profundidad).

Los dos recorridos son métodos sistemáticos que sirven para visitar todos los nodos y arcos de un grafo exactamente una vez. Podríamos decir que estos algoritmos nos permiten realizar recorridos controlados sobre el grafo.

Una de las operaciones más sencillas y elementales de cualquier estructura de datos es la búsqueda, por lo que se ha estandarizado el uso de estos dos recorridos; por eso se les conoce como algoritmos de búsqueda. Son utilizados para muchas otras operaciones sobre los grafos que no necesariamente realizan la búsqueda de un elemento dentro éste.

### 10.3.1 BFS – Breadth First Search (primero en anchura).

La idea fundamental de este recorrido es que, a partir de visitar el nodo de inicio, se visitan a todos sus vecinos, después a los vecinos de estos vecinos, simulando un recorrido nivel por nivel. Si habláramos de un árbol binario de búsqueda, el orden en que

se vistan a los vecinos es normalmente el orden en que fueron almacenados los datos en la representación que se esté utilizando.

Este algoritmo utiliza una fila (**queue**) auxiliar; para evitar ciclos infinitos se requiere un almacenamiento de estatus para cada nodo, los cuales arrancan en “Espera” y una vez que son procesados se cambia a “Procesado”.

El algoritmo general para BFS es:

Iniciarizar el status de todos los Nodos a “En Espera”.

Para cada Nodo del Grafo:

Si el status del Nodo es “En Espera”, entonces:

- Insertar el Nodo en la FILA cambiándole su estado a “Procesado”
- Mientras la FILA no esté vacía:
  - Sacar Nodo del frente la FILA y procesarlo.
  - Meter a la FILA todos los Vecinos del Nodo que tengan status “En Espera”, cambiándolo a “Procesado”.

La implementación del BFS utilizando una representación del Grafo en Matriz de Adyacencias sería:

```

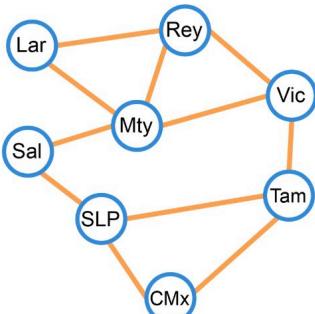
1. /* matAdj = a una matriz de bool que
2. representa la matriz de adyacencia
3. N = cantidad de Nodos.
4. */
5. void BFS(bool matAdj[N][N]){
6. /* status es un vector de bool
7. false = "En Espera"
8. true = "Procesado"
9. */
10. vector<bool> status(N, false)
11. queue<int> proceso;
12. int data;
13. for (int i=0; i<N; i++){
14. if (!status[i]){
15. fila.push(i);

```

```

16. status[i] = true;
17. while (!fila.empty()){
18. data = fila.front();
19. fila.pop();
20. cout << (data) << " ";
21. for (int j=0; j<v; j++){
22. if (matAdj[data][j] && !status[j]){
23. fila.push(j);
24. status[j] = true;
25. }
26. }
27. }
28. }
29. cout << endl;
30.
31. }
```

La figura 10.10 muestra el recorrido BFS que tendría un Grafo No-Dirigido y No-Ponderado de 8 nodos:



Asumiendo un almacenamiento en orden alfabético el recorrido BFS sería:  
**CMx-SLP-Tam-Sal-Vic-Mty-Rey-Lar**

Figura 10.10 – Recorrido BFS en un grafo no-dirigido y no-ponderado

### 10.3.2 DFS – Depth First Search (primero en profundidad).

Este recorrido se basa primordialmente en que, a partir de visitar el nodo de inicio, se visita al primer vecino de él, luego al primer vecino del primer vecino, y así recursivamente a todos sus vecinos NO procesados. El orden en el que se vistan a los vecinos, es normalmente el orden en que fueron almacenados los datos en la representación que se está utilizando.

Este algoritmo utiliza una pila (**stack**) auxiliar para simular la recursividad. Para evitar ciclos infinitos se requiere un almacenamiento de estatus para cada nodo, los cuales arrancan en “Espera” y una vez que son procesados se cambia a “Procesado”. El

algoritmo general para DFS es:

Iniciarizar el status de todos los Nodos a “*En Espera*”.

Para cada Nodo del Grafo:

Si el status del Nodo es “*En Espera*”, entonces:

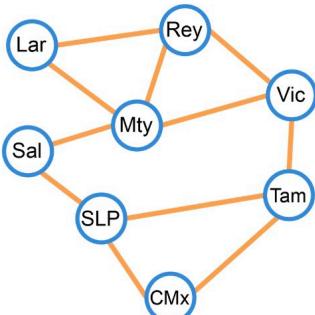
- Insertar el Nodo en la PILA.
- Mientras la PILA no esté vacía:
  - Sacar Nodo del tope la PILA, en caso de que su estado sea “*En Espera*” entonces procesarlo y cambiarle su estado a “*Procesado*”
  - Meter a la PILA todos los Vecinos del Nodo que tengan status “*En Espera*”, cambiándolo a “*Procesado*”.

La implementación del DFS y utilizando una representación del grafo en lista de adyacencias sería:

```

1. /* listAdj = a un vector de vectores de enteros que
2. representa la lista de adyacencia
3. */
4. void DFS(vector<vector<int> > &listAdj){
5. int v = listAdj.size();
6. stack<int> pila;
7. int data;
8. /* status es un vector de bool
9. false = "En Espera"
10. true = "Procesado"
11. */
12. vector<bool> status(v, false);
13. for (int i=0; i<v; i++){
14. if (!status[i]){
15. pila.push(i);
16. while (!pila.empty()){
17. data = pila.top();
18. pila.pop();
19. if (!status[data]){
20. cout << (data)<< " ";
21. status[data] = true;
22. for (int i=listAdj[data].size()-1; i>=0; i--){
23. if (!status[listAdj[data][i]]){
24. pila.push(listAdj[data][i]);
25. }
26. }
27. }
28. }
29. }
30. }
31. cout << endl;
32. }
```

La figura 10.11 muestra el recorrido DFS que tendría un grafo no-dirigido y no-ponderado de 8 nodos:



Asumiendo un almacenamiento en orden alfabético el recorrido DFS sería:

**CMx-SLP-Sal-Mty-Lar-Rey-Vic-Tam**

Figura 10.11 Recorrido DFS en un Grafo No-Dirigido y No-Ponderado

## 10.4 Conceptos complementarios de un grafo.

### 10.4.1 Grafo sin ciclos (árbol)

**U**n Grafo que no contenga ciclos es considerado un árbol y tendrá las siguientes características:

- El grafo debe ser conexo, esto es que para cualquier par de nodos existe una trayectoria simple que los conecta, si se quitará un arco dejaría de ser conexo.
- El grafo no debe contener ciclos. Si tiene N nodos contiene N-1 arcos, y no contiene ciclos; si se añadiera un arco se formaría un ciclo.

La figura 10.12 muestra un ejemplo de un grafo conexo y sin ciclos (árbol).

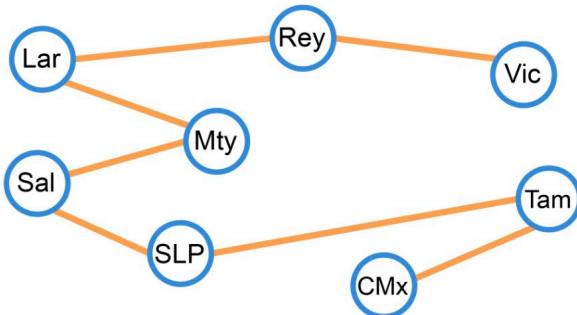
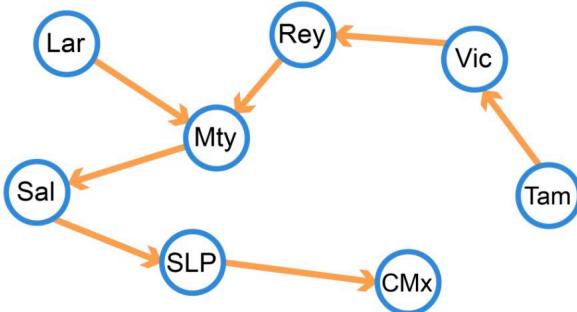


Figura 10.12 Grafo conexo y sin ciclos (árbol)

#### 10.4.2 DAG (*Directed Acyclic Graph*).

Un DAG o grafo dirigido y sin ciclos, es un grafo que los arcos tienen con una sola dirección y en donde no hay ciclos; además de ser imposible atravesar el grafo saliendo de cualquier punto.

La figura 10.13 muestra un ejemplo de un DAG (*Directed Acyclic Graph*).

Figura 10.13 DAG (*Directed Acyclic Graph*)

#### 10.4.3 Ordenamiento topológico (*Topological Sort*).

Una ordenación topológica se puede llevar a cabo en un DAG. Lo que realiza es encontrar el orden en el que se pueden visitar los Nodos (realizar tareas) de forma tal que respete la proceden-

cia de un nodo para ser visitado.

Uno de los algoritmos utilizados para el ordenamiento topológico es una modificación DFS, pero en lugar de ir imprimiendo, se utiliza una pila donde se almacenan cuando ya no se tienen nodos a visitar, para posteriormente imprimirla. Se puede llegar a tener diferentes ordenamientos topológicos válidos. En la figura 10.14 se puede apreciar un ejemplo de ordenamiento topológico.

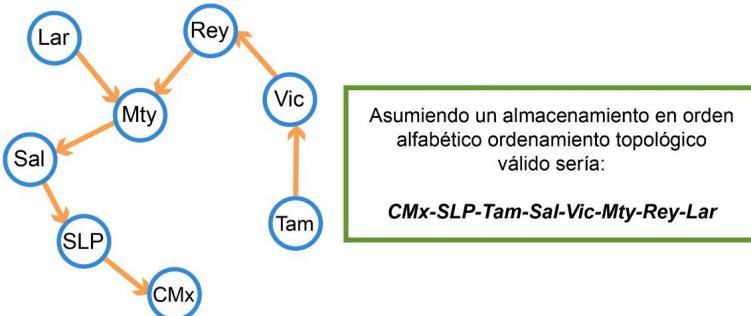


Figura 10.14. Ordenamiento topológico

#### 10.4.4 Grafo bipartito

Un grafo bipartito es un grafo cuyos nodos se pueden separar en dos conjuntos disjuntos **U** y **V**, donde:

$$U \cup V = N$$

$$U \cap V = \emptyset$$

De forma que los arcos solo pueden conectar nodos de conjuntos diferentes. La figura 10.15 muestra un ejemplo de un grafo que es considerado grafo bipartito.

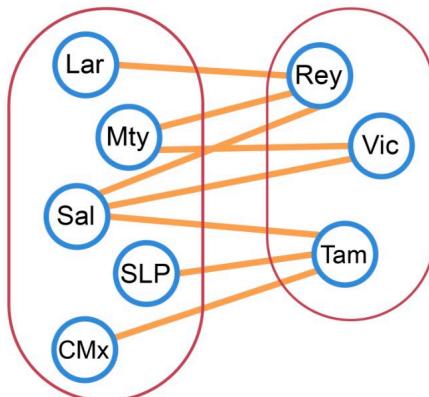
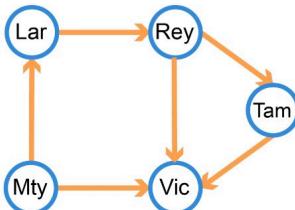


Figura 10.15 Grafo bipartite

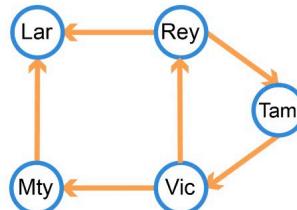
#### 10.4.5 Componentes fuertemente conectados

Un grafo dirigido es fuertemente conexo o conectado cuando existe una trayectoria o camino entre cada par de nodos que componen lo componen; en caso de que no exista para todo par de nodos, el grafo será un grafo débilmente conexo o conectado.

En la figura 10.16 se puede ver un grafo fuertemente conectado (**grafo A**), como se ve desde cualquier nodo, existe alguna trayectoria para visitar a cada uno de los nodos restantes. Por otro lado, también se puede ver un grafo débilmente conectado (**grafo B**) ya que de **Lar** no puedes ir a ningún otro nodo.



Grafo A: Fuertemente conectado



Grafo B: Débilmente conectado

Figura 10.16 Grafo fuertemente y débilmente Conectado

Dado un grafo débilmente conectado, se pueden llegar a identificar componentes fuertemente conectados. En la figura 10.17 se puede visualizar que el grafo tiene 2 componentes fuertemente conectados:

- {Lar, Rey, Tam, Vic, Mty}
- {Sal, SLP, CMx}

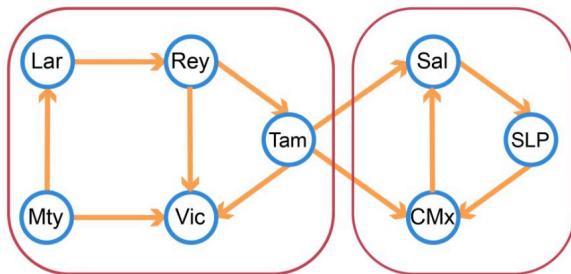
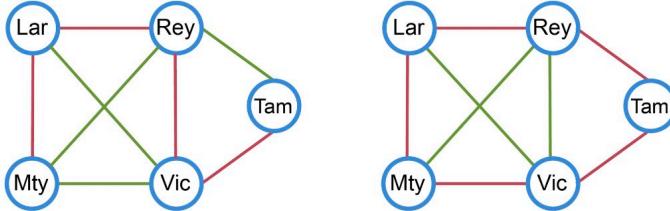


Figura 10.17 Componentes fuertemente conectados

#### 10.4.6 Camino y ciclo hamiltoniano

Dentro de un grafo, un camino hamiltoniano es un camino de un nodo a otro que visita a todos los nodos una sola vez y sin repetir. Si el último nodo visitado es un adyacente al inicial, entonces se puede cerrar el ciclo teniendo así un ciclo hamiltoniano.

En la figura 10.18 se puede apreciar por un lado un posible camino hamiltoniano en el grafo A y por el otro un posible ciclo hamiltoniano en el grafo B.



Grafo A: Camino Hamiltoniano

 $Mty \rightarrow Lar \rightarrow Rey \rightarrow Vic \rightarrow Tam$ 

Grafo B: Ciclo Hamiltoniano

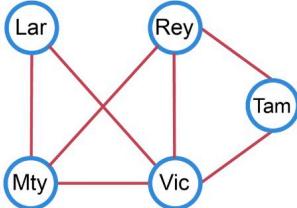
 $Mty \rightarrow Lar \rightarrow Rey \rightarrow Tam \rightarrow Vic \rightarrow Mty$ 

Figura 10.18 Camino y ciclo hamiltoniano

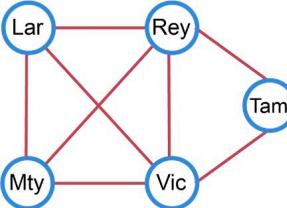
#### 10.4.7 Camino y ciclo euleriano

Dentro de un grafo, un camino euleriano es un camino de un nodo a otro visitando a todos los arcos una sola vez y sin repetir. Si el último nodo visitado es un adyacente al inicial, entonces se puede cerrar el ciclo teniendo así un ciclo euleriano.

En la figura 10.19 se puede apreciar por un lado un posible camino hamiltoniano en el grafo A y por el otro un posible ciclo euleriano en el grafo B.



Grafo A: Camino Euleriano

 $Mty \rightarrow Lar \rightarrow Vic \rightarrow Mty \rightarrow Rey \rightarrow Vic \rightarrow Tam \rightarrow Rey$ 

Grafo B: Ciclo Euleriano

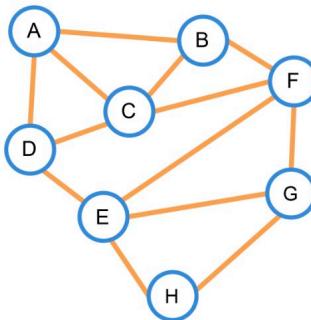
 $Mty \rightarrow Lar \rightarrow Vic \rightarrow Mty \rightarrow Rey \rightarrow Vic \rightarrow Tam \rightarrow Rey \rightarrow Lar$ 

Figura 10.19 Camino y Ciclo Euleriano.

## Ejercicios



1. Dado el grafo mostrado a continuación, contesta:



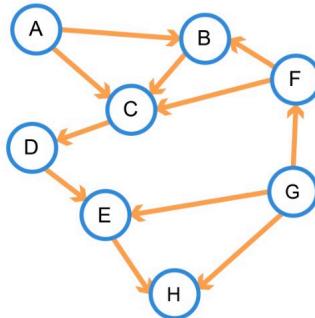
- a. ¿Cuántos nodos tiene el grafo?
- b. ¿Cuántos arcos tiene el grafo?
- c. ¿Quiénes son los nodos vecinos del nodo F?
- d. Menciona al menos una trayectoria simple para ir de H a C.
- e. ¿Es un grafo dirigido o no-dirigido?
- f. ¿Es un grafo ponderado o no-ponderado?
- g. Menciona un posible ciclo saliendo de E.
- h. ¿Cómo queda la representación de ese grafo en una matriz de adyacencias?
- i. ¿Cómo queda la representación de ese grafo en una lista de adyacencias?
- j. ¿Cómo imprimirá la visita de primero en anchura (BFS)?

**k.** ¿Cómo imprimirá la visita de primero en profundidad (DFS)?

**l.** Menciona un posible ciclo hamiltoniano saliendo de A.

**m.** Menciona un posible ciclo euleriano saliendo de A.

**2.** Dado el grafo mostrado a continuación, contesta:



**a.** ¿Cuáles son los nodos tiene el grafo?

**b.** ¿Cuáles son los arcos tiene el grafo?

**c.** ¿Es un grafo dirigido o no-dirigido?

**d.** ¿Es un grafo ponderado o no-ponderado?

**e.** Menciona al menos un posible orden topológico.



# Capítulo 11. Estructuras de datos sin relación (conjuntos)

11

**U**n conjunto es un grupo de datos sin relación entre sí. Al trabajar con un conjunto, lo importante es si un elemento pertenece o no a ese conjunto de datos.

Los elementos de la estructura de datos conjunto no pueden ser de diferentes tipos, no se aceptan datos repetidos. Según la aplicación se puede almacenar:

- Elementos simples
- Elementos compuestos: objetos, pero un atributo tendría que ser la llave.

Un conjunto de elementos simples tiene aplicaciones relacionadas con las operaciones típicas de conjuntos matemáticos: pertenencia, intersección, unión, diferencia, etc. Entre otras cosas, es útil para hacer validaciones.

Un conjunto de elementos estructurados, gracias a la operación de pertenencia, puede funcionar como una estructura de búsqueda.

Típicamente se utiliza un arreglo para representar la estructura de datos conjunto de elementos, la forma en como se almacenan los elementos en el arreglo esta determinado por la técnica de Hashing.

## 11.1 Técnica de hashing.

**L**a idea de la técnica de hashing es dispersar los datos sobre el arreglo, realizando una función matemática llamada función de hashing a la llave del elemento para localizar la posición que le correspondería. Lo ideal es que esa posición siempre esté disponible para él.

La figura 11.1 muestra un ejemplo de la técnica de hashing.

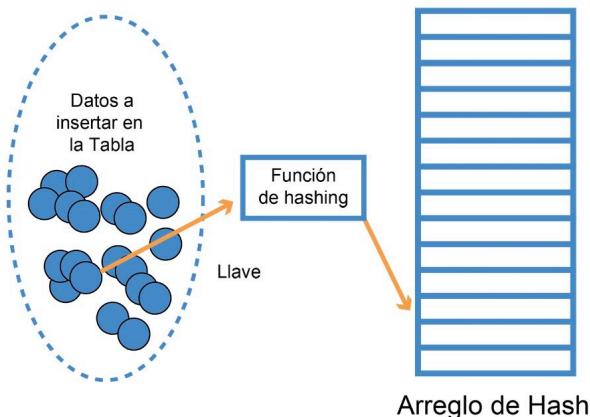


Figura 11.1 Ejemplo gráfico de la técnica de hashing.

La técnica del hashing debe preocuparse por:

- Diseñar una buena función de hashing para distribuir los elementos lo más aleatoriamente posible en el arreglo y que se puedan ejecutar de forma rápida.
- Dado que no es posible asegurar una función de hashing que garantice un valor distinto para cada elemento, también se debe preocupar por una estrategia para el manejo de colisiones.

Algunos métodos más comunes empleados para el diseño de funciones de hashing son:

- **Selección de dígitos:** esta técnica consiste en seleccionar algunos de los dígitos que conforman la llave y con ellos generar el índice para la tabla.
- **Residual:** consiste en utilizar como índice el residuo de dividir la llave entre el tamaño de la tabla, se recomienda normalmente que el tamaño sea un número primo.
- **Cuadrática:** consiste en elevar al cuadrado el valor de la llave y del resultado tomar los dígitos centrales, ya que son los más involucrados en el proceso.
- **Plegamiento:** consiste en agrupar algunos de los dígitos de la llave y aplicarles alguna(s) operación matemática para generar un índice.

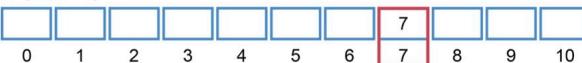
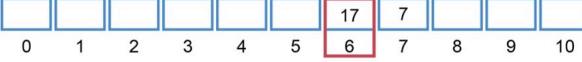
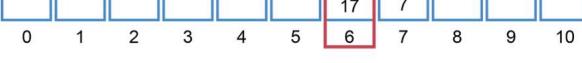
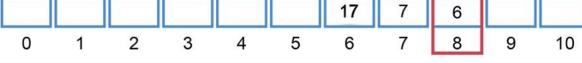
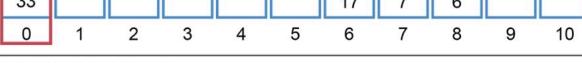
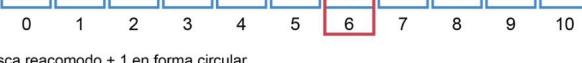
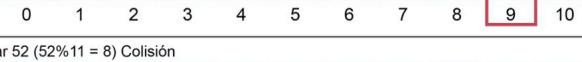
Las estrategias más comunes para el manejo de colisiones se dividen en dos grupos, dependiendo dónde y cómo se almacenan los elementos colisionados.

Estos grupos son:

- **Método de dirección abierta/ método de hashing cerrado/ rehashing:** su principal característica es que el elemento colisionado es “reacomodado” dentro del espacio de la tabla. La tabla se considera un espacio **circular** y su funcionamiento se basa en el índice original generado por la función de **hashing** (dirección base).

Ejemplo de **hashing** cerrado en una tabla de 11 posiciones con una función de **hashing** de residuo 11 y con un manejo de colisiones de base + 1, insertando los elementos 7, 17, 6, 33, 28, 52 y

21.

|                                          |                                                                                     |
|------------------------------------------|-------------------------------------------------------------------------------------|
| Insertar 7 ( $7 \% 11 = 7$ )             |    |
| Insertar 17 ( $17 \% 11 = 6$ )           |    |
| Insertar 6 ( $6 \% 11 = 6$ ) Colisión    |    |
| Se busca reacomodo + 1 en forma circular |    |
| Insertar 33 ( $33 \% 11 = 0$ )           |    |
| Insertar 28 ( $28 \% 11 = 6$ ) Colisión  |    |
| Se busca reacomodo + 1 en forma circular |    |
| Insertar 52 ( $52 \% 11 = 8$ ) Colisión  |   |
| Se busca reacomodo + 1 en forma circular |  |
| Insertar 21 ( $21 \% 11 = 10$ ) Colisión |  |
| Se busca reacomodo + 1 en forma circular |  |

- **Métodos de encadenamiento/hashing abierto:** este encadenamiento puede ser:

- **Encadenamiento externo:** es donde se tiene un apuntador en cada casilla a una lista encadenada de colisiones para ese dato. La figura 11.2 muestra un ejemplo de encadenamiento externo.

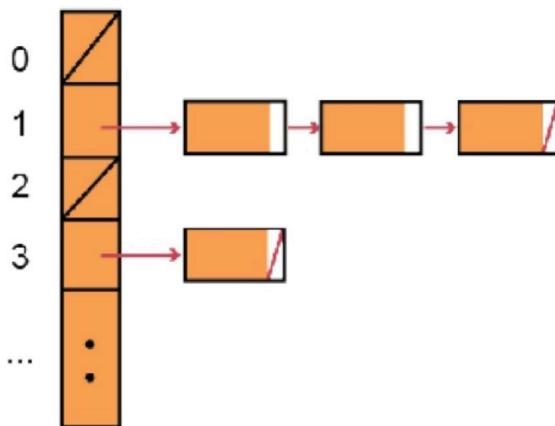


Figura 11.2 Ejemplo gráfico del manejo de colisiones de encadenamiento externo

- **Encadenamiento de colisiones:** en la misma tabla existe un espacio para almacenar las colisiones. La figura 11.3 muestra un ejemplo de encadenamiento de colisiones.

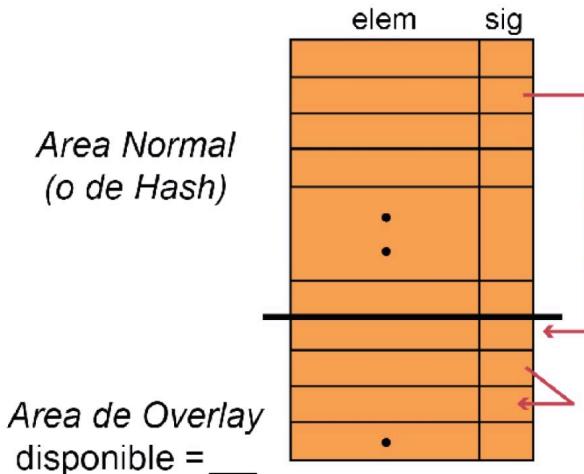


Figura 11.3 Ejemplo gráfico del manejo de colisiones de encadenamiento de colisiones

## 11.2 Tipo de dato conjunto disjunto (disjoint set)

**E**s una estructura de datos que mantiene un conjunto de elementos particionados en un número de conjuntos disjuntos (no se solapan los conjuntos). Un algoritmo unión-buscar es un algoritmo que realiza dos importantes operaciones en esta estructura de datos:

- **Buscar:** determina a cuál subconjunto pertenece un elemento. Esta operación puede usarse para verificar si dos elementos están en el mismo conjunto.
- **Union:** une dos subconjuntos en uno solo.

La figura 11.4 muestra el ejemplo de una estructura de conjuntos disjuntos en donde tiene 4 particiones.

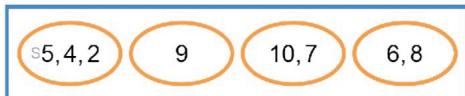


Figura 11.4 Ejemplo gráfico de estructura de datos de conjunto disjunto.

## Ejercicios



- 1.** Dados los siguientes datos a insertar en una tabla de 13 posiciones con función de hashing residuo 13 y manejo de colisiones de hashing cerrado con función de rehashing de base +2. Mostrar cómo quedaría la tabla:

**5, 100, 43, 26, 29, 52, 29, 44, 3, 10, 113, 126.**

- 2.** Dados los siguientes datos a insertar en una tabla de 13 posiciones con función de hashing residuo 13 y manejo de colisiones de hashing externo en una lista encadenada. Mostrar cómo quedaría la tabla:

**5, 100, 43, 26, 29, 52, 29, 44, 3, 10, 113, 126.**



## Aviso legal



**Nombres:** Cueva Hernández, Víctor M. de la, autor. | González Guerra, Luis Humberto, autor. | Salinas Gurrión, Edgar Gerardo, autor.

**Título:** Estructuras de datos y algoritmos fundamentales / Víctor Manuel de la Cueva Hernández, Luis Humberto González Guerra, Edgar Gerardo Salinas

**Descripción:** Primera edición | México : Editorial Digital del Tecnológico de Monterrey, [2020]

**Identificadores:** ISBN

**Temas:** LCSH: Data structures (Computer science) | Computer algorithms. | Electronic books. | Local: Estructuras de datos (Computación) | Algoritmos computacionales. | Libros electrónicos.

**Clasificación:** LCC QA76.9.D35 | DDC 005.73

---

### Editorial Digital del Tecnológico de Monterrey

Gerardo Isaac Campos Flores. Director de Efectividad Institucional del Tecnológico de Monterrey

Alejandra González Barranco. Líder de Editorial Digital.

Elizabeth López Corolla. Coordinadora editorial.

María Fernanda Vergara Bernal. Correctora de estilo.

---

### Innovación y diseño para la enseñanza y el aprendizaje.

Noemí Villarreal Rodríguez. Coordinación de proyectos institucionales y empresariales.

Jesús Alejandro Rocha Gámez. Administración de proyecto.

María Isabel Zendejas Morales. Diseño Editorial.

Gustavo Arteaga Mondragón. Diseño Editorial.

---

eBook editado, diseñado, publicado y distribuido por el Instituto Tecnológico y de Estudios Superiores de Monterrey. Se prohíbe la reproducción total o parcial de esta obra por cualquier medio sin previo y expreso consentimiento por escrito del Instituto Tecnológico y de Estudios Superiores de Monterrey.

Editorial: Instituto Tecnológico y de Estudios Superiores de Monterrey

Ave. Eugenio Garza Sada 2501 Sur Col. Tecnológico C.P. 64849 | Monterrey, Nuevo León | México.

Estructuras de datos y algoritmos fundamentales.

ISBN Obra Independiente: 978-607-501-605-4

Primera edición: julio 2020.

