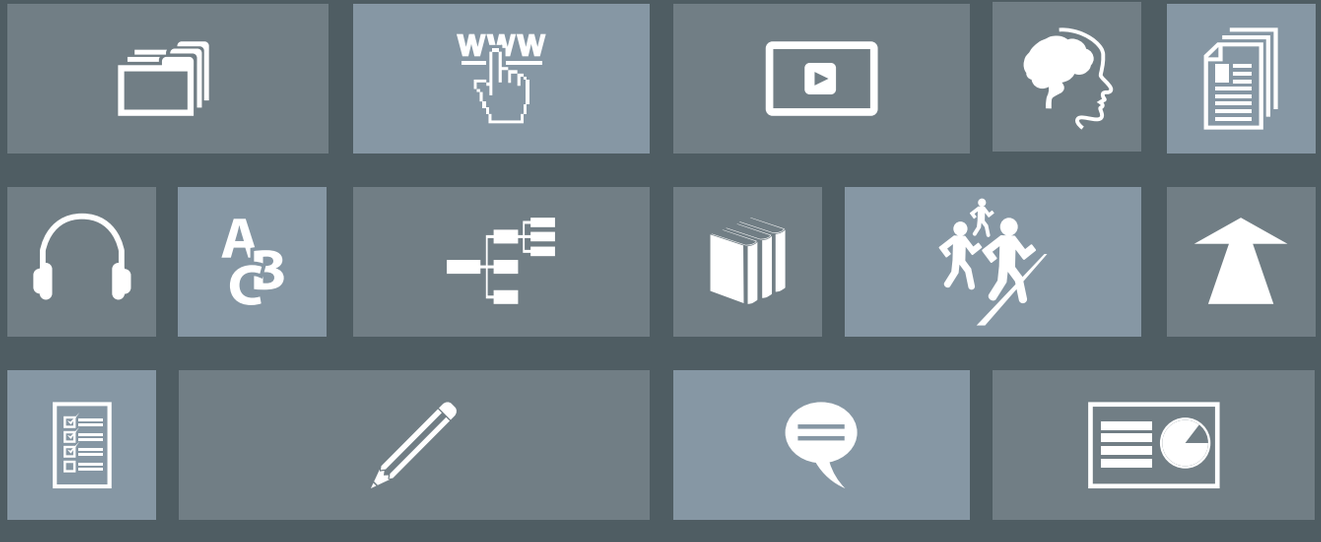
Universidad Continental

Estructura de Datos

Manual Autoformativo Interactivo

Walter F. Jesús Videla



Datos de catalogación bibliográfica

Estructura de Datos. Manual Autoformativo Interactivo
Walter F. Jesús Videla
Primera edición

Huancayo, abril de 2017

Av. San Carlos 1980, Huancayo-Perú
Teléfono: (51 64) 481-430 anexo 7361
Correo electrónico: recursosucvirtual@continental.edu.pe
http://www.continental.edu.pe/

Versión e-book
Disponible en http://repositorio.continental.edu.pe/
ISBN electrónico N.° 978-612-4196-

Dirección: Emma Barrios Ipenza
Edición: Eliana Gallardo Echenique
Asistente de edición: Andrid Poma Acevedo
Asesoría didáctica: Karine Bernal
Corrección de textos: Eliana Gallardo Echenique
Diseño y diagramación: Francisco Rosales Guerra

ÍNDICE

INTRODUCCIÓN 7

ORGANIZACIÓN DE LA ASIGNATURA 8

RESULTADO DE APRENDIZAJE: 8

UNIDADES DIDÁCTICAS 8

TIEMPO MÍNIMO DE ESTUDIO 8

UNIDAD I REPRESENTACIÓN DE DATOS Y ESTRUCTURAS DE CONTROL 9

DIAGRAMA DE ORGANIZACIÓN DE LA UNIDAD I 9

ORGANIZACIÓN DE LOS APRENDIZAJES 10

Tema N.º 1: Representación de datos 11

Introducción al tema 11

1. ¿En qué consiste la estructura de datos? 11
2. ¿Por qué se denomina estructura de datos? 12
3. Representación de datos 12

Tema n.º 2: Estructuras de control 22

Introducción al tema 22

1. ¿Por qué usar estructuras de control? 22
2. La estructura de control condicionada simple 23
3. La estructura de control condicional doble 25
4. La estructura de control múltiple 26

Tema n.º 3: Estructuras de control repetitivas 29

Introducción al tema 29

1. Estructuras de control repetitivas 29
2. La instrucción FOR 31

Lectura seleccionada n.º 1: 33

Actividad N.º 1 33

Glosario de la Unidad I 34

Bibliografía de la Unidad I 35

Autoevaluación n.º 1 36

UNIDAD II ARREGLOS Y MATRICES 39

DIAGRAMA DE ORGANIZACIÓN DE LA UNIDAD II 39

ORGANIZACIÓN DE LOS APRENDIZAJES 40

Tema n.º 1: Arrays unidimensionales, bidimensionales y multidimensionales. Operaciones con ellos 41

Introducción 41

1. Definición 41
2. Creación de arreglos 42
3. Operaciones con arreglos 42
4. Conversión de arreglos multidimensionales a unidimensionales 47
5. Arreglo de arreglos 50

Tema n.º 2: Matrices 52

1. Definición 52
2. Implementación 53
3. Operaciones con matrices 55
4. Matrices especiales 59

Lectura seleccionada n.º 02: Gestión dinámica de memoria 59

Actividad N.° 02 60

Glosario de la Unidad II 61

Bibliografía de la Unidad II 62

Autoevaluación de la Unidad II 62

UNIDAD III ESTRUCTURAS LINEALES Y NO LINEALES DE DATOS 65

DIAGRAMA DE ORGANIZACIÓN 65

ORGANIZACIÓN DE LOS APRENDIZAJES 66

Tema n.º 1: Pilas, colas y listas 67

1. Pilas 67
2. Colas 69
3. Listas 73

Tema N.º 2: Grafos 78

1. Definición 78

2. Representación de grafos 79
3. Operaciones con grafos 81

Tema n.º 3: Árboles 89
1. Definición 89
2. Implementación de árboles 90
3. Árbol binario 90
Lectura seleccionada n.º 3 97
Actividad N.º 3 97
Actividad N.º 4 97

Glosario de la Unidad III 98

Bibliografía de la Unidad III 98

Autoevaluación N.º3 100

UNIDAD IV ORGANIZACIÓN DE DATOS Y ARCHIVOS 103

DIAGRAMA DE ORGANIZACIÓN 103

ORGANIZACIÓN DE LOS APRENDIZAJES 104

Tema N.º 1: Tablas Hash 105
1. Definición 105
2. Función de dispersión 105
3. Resolución de colisiones 107

Tema N.º 2: Modelo de datos relacional 112
1. Base de datos 112
2. El modelo entidad-relación 112
3. Fundamentos del modelo entidad-relación 114
4. Estructura de una base de datos relacional 116
5. Interactuación Aplicación–Base de datos 120

Tema N.º 3: Organización de archivos 124
1. Archivo 124
2. Organización de archivos 125
Lectura seleccionada n.º 04 137
Actividad N.º 4 138
Actividad N.º 5 138

Glosario de la Unidad IV 139

Bibliografía de la Unidad IV 140

Autoevaluación N.º 4 141

ANEXO: Solucionario de las autoevaluaciones 144

INTRODUCCIÓN

El presente manual autoformativo corresponde a la asignatura de Estructura de Datos de la modalidad virtual de la Universidad Continental y constituye el material didáctico más importante para su estudio.

Al finalizar la asignatura de estructura de datos usted estará en capacidad de elaborar y programar algoritmos más eficientes en términos de tiempo de procesamiento y uso de memoria, garantizándole el desarrollo de *software* de calidad. Los temas a tratar son los siguientes:

- Unidad I: Representación de datos y estructuras de control
- Unidad II: Arreglos y matrices
- Unidad III: Estructuras lineales y no lineales
- Unidad IV: Organización de datos y archivos

Durante su desarrollo se describirán conceptualmente y con ejemplos las principales estructuras de datos como arreglos, matrices, colas, listas, grafos y árboles de búsqueda.

Se sugiere seguir esta secuencia de pasos:

a) Realizar el estudio de los contenidos

b) Estudiar las lecturas seleccionadas

c) Desarrollar la autoevaluación

d) Desarrollar las actividades programadas

El autor

ORGANIZACIÓN DE LA ASIGNATURA

RESULTADO DE APRENDIZAJE:

Al finalizar la asignatura, el estudiante será capaz de seleccionar las estructuras de datos adecuadas para los algoritmos, de acuerdo a la problemática planteada.

UNIDADES DIDÁCTICAS

UNIDAD I	UNIDAD II	UNIDAD III	UNIDAD IV
Representación de datos y estructuras de control	Arreglos y matrices	Estructuras lineales y no lineales de datos	Organización de datos y archivos
Resultado de aprendizaje Al finalizar la unidad, el estudiante será capaz de seleccionar las estructuras de control adecuadas, según la problemática propuesta.	**Resultado de aprendizaje** Al finalizar la unidad, el estudiante será capaz de seleccionar las estructuras de arreglos y matrices, según la problemática propuesta.	**Resultado de aprendizaje** Al finalizar la unidad, el estudiante será capaz de seleccionar las estructuras lineales y no lineales, en la solución de diversos problemas.	**Resultado de aprendizaje** Al finalizar la unidad, el estudiante será capaz de reconocer la organización de datos y archivos a través del estudio de casos.

TIEMPO MÍNIMO DE ESTUDIO

UNIDAD I	UNIDAD II	UNIDAD III	UNIDAD IV
1.ª y 2.ª semanas 12 horas	3.ª y 4.ª semanas 20 horas	5.ª y 6.ª semanas 20 horas	7.ª y 8.ª semanas 12 horas

UNIDAD I

REPRESENTACIÓN DE DATOS Y ESTRUCTURAS DE CONTROL

DIAGRAMA DE ORGANIZACIÓN DE LA UNIDAD I

ORGANIZACIÓN DE LOS APRENDIZAJES

Resultado de aprendizaje de la Unidad I:

Al finalizar la unidad, el estudiante será capaz de seleccionar las estructuras de control adecuadas, según la problemática propuesta.

CONOCIMIENTOS	HABILIDADES	ACTITUDES
Tema n.º 1: Representación de datos 1. ¿En qué consiste la estructura de datos? 2. ¿Por qué se denomina estructura de datos? 3. Representación de datos **Tema n.º 2: Estructuras de control** 1. ¿Por qué se deben usar estructuras de control? 2. Estructura de control condicionada simple 3. Estructura de control condicionada doble 4. Estructura de control condicionada múltiple **Tema n.º 3: Estructura de control repetitivas** 1. Estructura de control repetitiva 2. La instrucción FOR **Lectura seleccionada n.º 1:** El concepto de datos estructurados **Autoevaluación de la Unidad I**	1. Identifica los conceptos de representación de datos. 2. Describe las estructuras de control simple, doble y múltiple. 3. Identifica y compara las estructuras de control repetitivas.	1. Demuestra perseverancia y esfuerzo durante el desarrollo de los ejercicios. 2. Toma conciencia de la importancia de la asignatura en su formación profesional. 3. Valora las relaciones entre sus compañeros.

TEMA N.º 1: REPRESENTACIÓN DE DATOS

Introducción al tema

El objetivo de esta primera unidad es brindar los conceptos fundamentales sobre los que se elaboran los demás contenidos de la asignatura. Por ello, en este primer tema se describe en qué consisten las estructuras de datos y qué relación guardan con los algoritmos. A continuación, se detallan las formas de representar los datos en el computador, diferenciando entre datos simples y datos estructurados. Asimismo, es primordial, para la implementación de tales estructuras la comprensión del tema de las denominadas estructuras de control, las cuales se abordan como último acápite. Se debe aclarar que, aunque tengan un nombre relacionado al título del curso, más están vinculadas al diseño de algoritmos.

1. ¿En qué consiste la estructura de datos?

Posiblemente a usted le resulten familiares algunas de estas frases: "se cayó el sistema", "el programa está lento", "se colgó el *software*, reinicia la máquina", entre otras. Descartando la existencia de problemas técnicos con el computador y la red de datos, el siguiente factor que puede afectar el rendimiento de un *software* es la calidad de los algoritmos que emplea para procesar la información. Precisamente, la estructura de datos se refiere al conjunto de técnicas para desarrollar *software*, o exactamente algoritmos, que utilicen de una manera eficiente los recursos de la computadora. Tal eficiencia es medida, principalmente, en términos de tiempo de procesamiento y uso de memoria.

Weiss (2000) considera que muchos algoritmos requieren una representación apropiada de los datos para lograr ser eficientes. Esta representación junto con las operaciones permitidas se llama estructura de datos.

RECUERDA:

Un algoritmo es "un conjunto de instrucciones claramente especificadas que el ordenador debe seguir para resolver un problema" (Weiss, 2000, p. 103).

La figura 1 ilustra esta descripción.

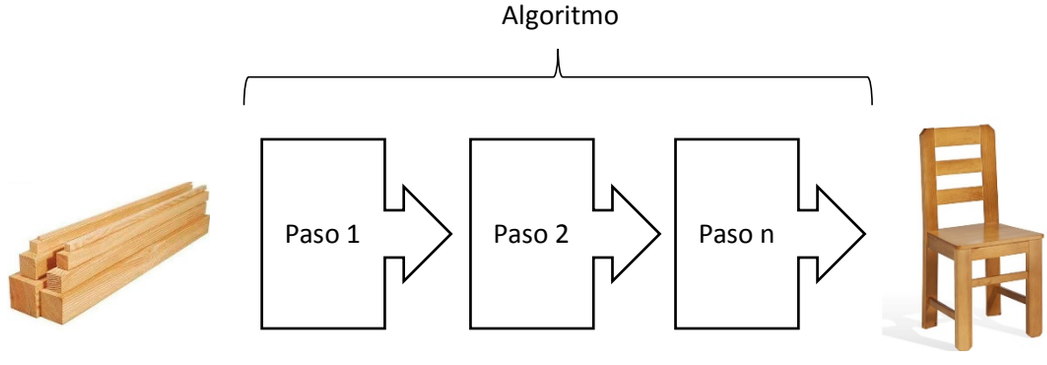


Figura 1 Representación de un algoritmo
Fuente: Elaboración propia

2. ¿Por qué se denomina estructura de datos?

Usted recordará que por definición todo computador transforma datos en información útil para el usuario; por ejemplo, si se ingresa la operación (5 + 2) se espera que el computador lo interprete, lo procese y devuelva el resultado 7. En lenguaje C++ podría trabajarse un programa como el siguiente:

```
void main()
{
      int a, b;
      cin>> a;
      cin>> b;
      cout<< "La suma es: "<< a+b << endl;
}
```

Programa 1.1

Una tarea muy sencilla ¿verdad?

¿Pero qué sucede cuando se debe procesar mayor cantidad de datos, por ejemplo, calcular la cantidad de notas aprobatorias de una lista de 50 estudiantes?

1ª opción: Se elabora un programa que solicite cada nota (una por una), mientras un contador va incrementándose en la unidad cada vez que identifica una nota mayor que 10.

¿Cuánto tiempo tomaría la digitación? ¿Y si se ingresa mal un dato?

2ª opción: Se puede solicitar cada nota y guardarla en una variable independiente; al final se compran por separado para determinar si son aprobatorias.

¿Cuántas líneas de código tendría este programa?

Entonces, frente a estos inconvenientes se propone disponer los datos en estructuras (de ahí "estructura de datos") como, por ejemplo, arreglos —o *arrays* por su denominación en inglés—, a fin de agilizar las tareas de lectura y escritura de los mismos, mientras se minimiza el esfuerzo de procesamiento, la cantidad de líneas de código, el tiempo de programación, etc.

3. Representación de datos

3.1. Tipos de datos

Para que un programa funcione debe estar en la capacidad de recibir, procesar y almacenar diferentes tipos de datos. Aunque cada lenguaje de programación maneja sus propios tipos, en términos generales su pueden considerar los siguientes:

- Numérico: 1, -5, 3.67, etc.
- Texto o cadena: "ABC", "Palabra", "wjesus@continental.edu.pe", etc.
- Fecha / hora: 07/09/15, 12/12/16 15:23, etc.
- Lógico: Verdadero, falso.

Sahni (2005) denomina a este universo de posibles valores como "objetos de datos" (o *Data Objects*), y lo define como un conjunto de instancias o valores.

Para que un programa pueda guardar un determinado valor hace uso de las llamadas "variables", las cuales físicamente se traducen en espacios reservados de memoria: imagine a la memoria de la computadora como una gran tabla en la que en cada celda se puede almacenar un dato; al crearse una variable y asignársele un nombre, el *software* reserva una de estas celdas. Observe la siguiente figura:

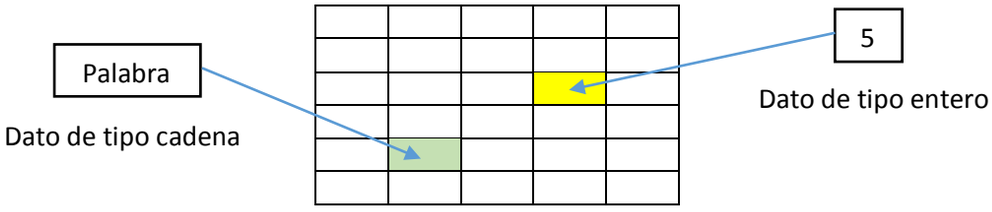


Figura 2. Representación de la memoria de un computador
Fuente: Elaboración propia

Para recuperar al valor correspondiente bastará con referenciar (llamar) al nombre de la variable creada.

El programa 1.2 muestra un ejemplo de cómo crear una variable, asignarle un valor y luego recuperarlo.

```
void main()

{
      int a;// Declarar la variable "a" de tipo entero
      a=5;        // La variable recibe el valor de 5
      cout<<"El valor ingresado es: "<< a << endl;    // Recuperar el
                                          // valor de a
}
```

Programa 1.2

3.2. Datos simples y datos estructurados

Según manifiesta Cruz (2011, p. 8), "la principal característica de los datos simples es que ocupan solo una casilla de la memoria, por lo tanto, hacen referencia a un único valor a la vez", mientras que "los datos estructurados se caracterizan por el hecho de que con un solo nombre (o también llamado identificador de variables) se hace referencia a un grupo de casillas de memoria"; a su vez cada uno de los elementos o componentes de un dato estructurado puede ser un dato simple u otro estructurado. Comparando las figuras 2 y 3, notará esta diferencia.

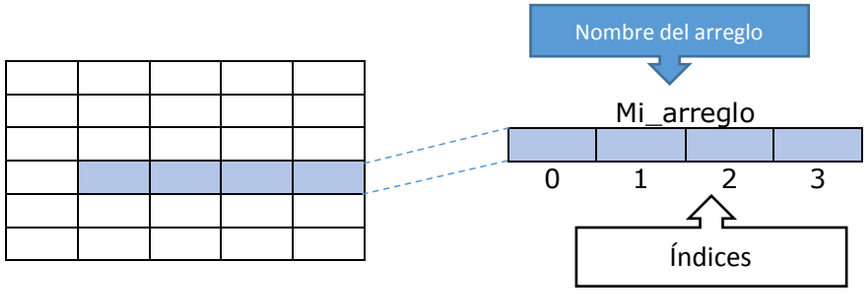


Figura 1.3. Representación de un dato estructurado en la memoria de un computador

Cabe precisar que la figura 3 es sólo una representación, ya que las casillas de un dato estructurado, si bien operan en conjunto, no necesariamente son adyacentes. Es precisamente esta característica la que permite realizar una primera clasificación de las estructuras de datos:

- Basadas en arreglos (o *Arrays*), también denominadas "contiguas".

- Basadas en punteros (o *Linked*), también denominadas "enlazadas".

La segunda forma de clasificarlas es según la "figura" que describen al graficarlas, pudiendo ser lineales y no lineales; a su vez estas se subclasifican de la siguiente manera:

- Estructuras lineales
  - Arreglos
  - Pilas
  - Colas
  - Listas
- Estructuras no lineales
  - Árboles
  - Grafos

Todos estos tipos de estructuras serán descritos al detalle a lo largo del presente manual.

3.3. Estructuras de datos basadas en arreglos (estructura contigua)

Cada instancia de una estructura de datos tipo lista lineal es una colección ordenada de elementos. Cada una de estas instancias es de la forma $(e_0, e_1, e_2, \ldots, e_{n-1})$ donde:

- n es un número natural finito, que representa el ancho o tamaño de la lista.
- e_i representa cada elemento de la lista e *i* es su índice.

Aunque resulte obvio mencionar que precede a , a y así sucesivamente, es necesario recalcar que esta relación de precedencia solo se da en listas lineales.

Algunos ejemplos de listas lineales son los siguientes:

- La lista de estudiantes de esta clase (ordenadas por el nombre)
- La lista de puntajes de un examen ordenados por mayor a menor
- La lista de arribos de vuelos aéreos, ordenados del más reciente al más antiguo

3.3.1. Operaciones con una lista lineal

a) Crear la lista
b) Destruir la lista
c) Determinar si la lista está vacía
d) Determinar el tamaño de la lista
e) Encontrar un elemento a partir de su índice
f) Encontrar el índice de un elemento dado
g) Borrar un elemento dado a partir de su índice

h) Insertar un nuevo elemento en un índice determinado

i) Listar los elementos in orden (ascendente o descendentemente)

3.3.2. Abstracción de una lista lineal:

Es posible representar una lista lineal al margen de algún lenguaje de programación (de ahí el término "abstracto") de la siguiente forma:

```
Lista Lineal
{
    Instancias
    Colección finita ordenada de cero o más elementos.
    Operaciones
    Empty()          :   Retorna verdadero si la lista está vacía; de lo contrario, falso.
    Size()           :   Retorna el número de elemento de la lista (tamaño)
    Get(índice)      :   Retorna el elemento de la posición (índice) especificada.
    IndexOf(x)       :   Retorna el índice del elemento x (primera ocurrencia). Si el elemento no existe,
                         devuelve -1.
    Erase(índice)    :   Borra el elemento cuyo índice es especificado.
    Insert(índice, x) :  Inserta el elemento x en la posición indicada como índice.
    Output()         :   Muestra (imprime) la lista de elementos de izquierda a derecha.
}
```

Tomada de *Data Structures, Algorithms, and Applications in C++*, por Sahni (2005), p. 141.

Por ejemplo, al declarar la siguiente lista lineal L = {Apple, HP, Toshiba, Vaio}, los resultados de las operaciones indicadas serían (considerar L original para cada caso):

```
L.Empty()              :   Falso
L.Size()               :   4
L.Get(2)               :   "HP"
L.IndexOf("Toshiba")   :   2 (el primer elemento tiene índice 0)
L.Erase(1)             :   {Apple, Toshiba, Vaio}
L.Insert(3, "Assus")   :   {Apple, HP, Toshiba, Assus, Vaio}
L.Output()             :   {Apple, HP, Toshiba, Vaio}
```

Ahora intente usted, con la siguiente lista: M = {a, b, c, d}

```
M.Empty()      :
M.Size()       :
M.Get(0)       :
M.Get(2)       :
M.Get(-3)      :
M.IndexOf("c") :
M.IndexOf("q") :
M.Erase(1)     :
M.Insert(0, "e") :
M.Insert(2, "f") :
M.Output()     :
```

3.3.3. Representación de arreglos

Cada elemento de un arreglo puede ser asignado o localizado empleando una fórmula matemática; la más sencilla, y usada de forma natural, es la siguiente:

Ubicación (i) = i	*formula (a)*

Esta fórmula significa que el i-ésimo elemento de la lista se encuentra en la posición i. Observe el siguiente caso:

Se decide guardar la lista [5, 2, 4, 8, 1] en un arreglo de tamaño 10. Haciendo uso de la fórmula los elementos, se ubicarían como sigue:

5	2	4	8	1					
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Figura 4. Ubicación de elementos en un arreglo con la fórmula (a)

Esto debido a que, si se reemplazan los valores de i, se tiene:

- Ubicación(0) = 0
- Ubicación(1) = 1
- Ubicación(2) = 2
- Ubicación(3) = 3
- Ubicación(4) = 4

¿Le pareció demasiada obvia la primera fórmula? Cuando se trata de estructurar datos, no es la única que existe.

Si se emplea la fórmula:

Ubicación(i) = Tamaño del arreglo – i – 1	*formula(b)*

Al asignar los valores se ubicarían de derecha a izquierda, así:

					1	8	4	2	5
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Figura 5. Ubicación de elementos en un arreglo con la fórmula (b)

Debido a que:

- Ubicación(0) = 10 – 0 – 1 = 9
- Ubicación(1) = 10 – 1 – 1 = 8
- Ubicación(2) = 10 – 2 – 1 = 7
- Ubicación(3) = 10 – 3 – 1 = 6
- Ubicación(4) = 10 – 4 – 1 = 5

La fórmula que se emplee para asignar/localizar un elemento específico en el arreglo también afectará la forma cómo se insertan y eliminan nuevos elementos.

Revisemos el caso para la fórmula (a): al insertarse un nuevo valor (7) en la posición 2 del arreglo, los elementos [4, 8, 1] deberían correr una posición hacia la derecha.

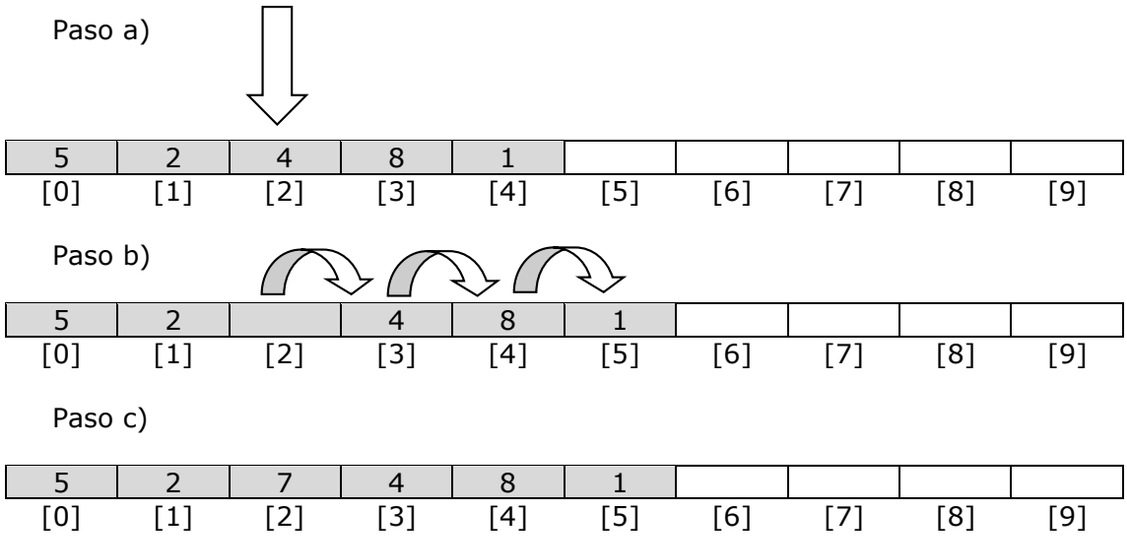


Figura 6. Proceso de inserción de un nuevo elemento en un arreglo que emplea la fórmula (a)

Por el contrario, al eliminar un elemento, todos los que se ubiquen a la derecha de él deberán correr una posición hacia la izquierda.

El programa 1.3 muestra la implementación de la operación de inserción de un nuevo elemento a partir de la fórmula (a).

```
#include <iostream>
using namespace std;

const int N = 10; // Tamaño del arreglo
int miArreglo[N]; // Creación del arreglo con N elementos


// Procedimiento para inicializar la matriz
void init()
{
      miArreglo[0] = 5; //Insertar en la posición 0, el elemento 5
      miArreglo[1] = 2; //Insertar en la posición 1, el elemento 2
      miArreglo[2] = 4; //Insertar en la posición 2, el elemento 4
      miArreglo[3] = 8; //Insertar en la posición 3, el elemento 8
      miArreglo[4] = 1; //Insertar en la posición 4, el elemento 1
};

// Función que determina si la matriz está vacía
bool empty()
{
      bool vacío = true;
      int i = 0;
      while (vacío==true && i<=N)
      {
            if (miArreglo[i] == 0)
                  vacío = false;
            i++;
      }
```

```
      return vacío;
};

// Procedimiento para insertar un elemento en el arreglo
void Insert(int indice, int elemento)
{
      // Si la lista está vacía insertar el elemento en la posición [0]
      if (empty() == true)
            miArreglo[0] = elemento;
      else
      // Correr una posición a la derecha a partir de la posición del nuevo ele-
mento
      {
            int i = 0;
            while ((N-1)-i >= indice)
            {
                  miArreglo[(N-1)-i] = miArreglo[(N-1)-(i+1)];
                  i++;
            }
      }
      // Insertar el nuevo valor
      miArreglo[indice] = elemento;
};

// Procedimiento para imprimir los elementos del arreglo
void Output()
{
      cout<<endl;
      cout<<"Elementos del arreglo"<<endl;
      for (int i=0; i<=N; i++)
            cout<<"Elemento["<<i<<"]: "<<miArreglo[i]<<endl;
};

// PROGRAMA PRINCIPAL
void main()
{
      init();           // Insertar valores iniciales
      Output();         // Mostrar en pantalla
      Insert(2,7);      // Insertar, en la posición 2, el elemento 7
      Output();         // Mostrar en pantalla
      Insert(5,9);      // Insertar, en la posición 5, el elemento9
      Output();         // Mostrar en pantalla
      system("Pause");
};
```

Programa 1.3

3.4. Estructuras de datos basadas en punteros (estructura enlazada)

En este tipo de estructuras, los elementos pueden almacenarse en cualquier ubicación de la memoria. Por consiguiente, para crear una lista cada elemento tiene un enlace o puntero (o dirección) al siguiente elemento de la misma.

Como se vio, en el caso anterior la dirección es determinada por una fórmula matemática; en este tipo de representación las direcciones están distribuidas a lo largo de la lista.

La figura 7 ilustrala la lista L = {e_0, e_1, e_2, e_n}

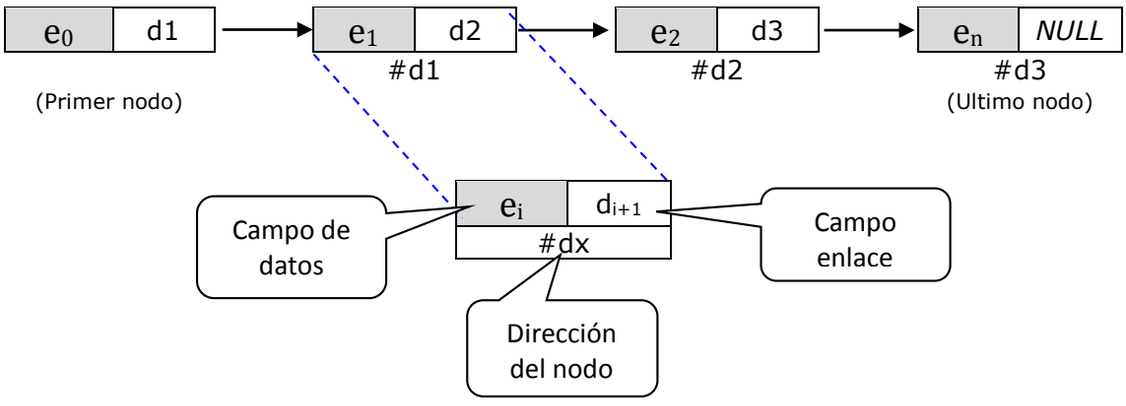


Figura 7. Representación de una lista basada en punteros

Como puede apreciar, a diferencia de una lista de tipo arreglo, en esta representación cada elemento (e_i) ocupa su propio nodo. A su vez, cada nodo tiene un "campo enlace" donde se almacena la dirección (en memoria) del siguiente elemento (d1, d2, …).

Asimismo, para ubicar un determinado elemento de la lista es necesario ubicarse en el primer nodo y "saltar" hasta el nodo del elemento deseado a través de los campos enlace.

La lista mostrada en la figura 7 es llamada "Lista enlazada simple" debido a que cada nodo tiene solo un enlace; también es denominada "estructura tipo cadena" debido a que los nodos están dispuestos de izquierda a derecha y el valor de enlace del último nodo es *NULL*.

3.4.1. **Eliminar elementos de una lista enlazada**

Observe el siguiente procedimiento para eliminar, por ejemplo, el elemento 1 de una lista enlazada:

- Ubicar segundo nodo (elemento 1).
- Capturar el valor de su campo enlace.
- Vincular el nodo 1 con el nodo 3.

Analice cuidadosamente la figura 8.

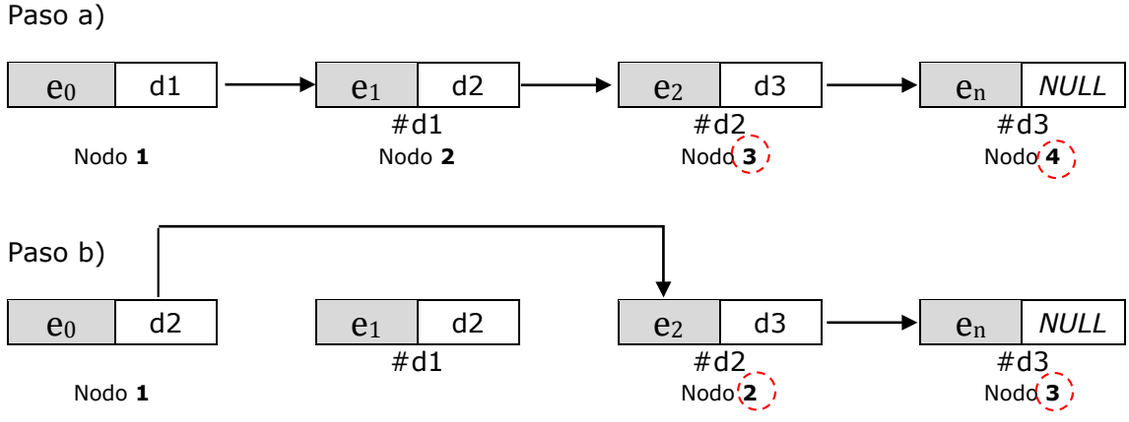


Figura 8. Proceso para eliminar un elemento de una lista enlazada

Note que, al eliminar un nodo, automáticamente el índice de todos los nodos siguientes decrece en 1 (círculos entrecortados en la figura). A diferencia de un arreglo, este índice es solo referencial.

3.4.2. **Insertar elementos en una lista enlazada**

Para insertar en elemento en la posición i, es necesario ubicarse en el nodo anterior (i-1) e insertar el nodo justo después de él.

El programa 1.4 permite implementar una lista enlazada e insertarle nuevos elementos.

```
#include <iostream>
#include <stdlib.h>
using namespace std;

struct nodo{
int nro;        // en este caso es un número entero
struct nodo *sgte;
};

typedef struct nodo *Tlista;

void insertarInicio(Tlista &lista, int valor)
{
    Tlista q;
    q = new(struct nodo);
q->nro = valor;
    q->sgte = lista;
    lista  = q;
}

void insertarFinal(Tlista &lista, int valor)
{
    Tlista t, q = new(struct nodo);

q->nro  = valor;
    q->sgte = NULL;

    if(lista==NULL)
    {
        lista = q;
}
else
    {
        t = lista;
        while(t->sgte!=NULL)
        {
            t = t->sgte;
        }
        t->sgte = q;
    }
}

void reportarLista(Tlista lista)
{
     int i = 0;

     while(lista != NULL)
     {
          cout <<' '<< i+1 <<") " << lista->nro << endl;
lista = lista->sgte;
          i++;
     }
}
```

```
void menu1()
{
    cout<<"\n\t\tLISTA ENLAZADA SIMPLE\n\n";
    cout<<" 1. INSERTAR AL INICIO                "<<endl;
    cout<<" 2. INSERTAR AL FINAL                 "<<endl;
    cout<<" 3. REPORTAR LISTA                    "<<endl;
    cout<<" 4. SALIR                             "<<endl;

    cout<<"\n INGRESE OPCIÓN: ";
}
/*                        Función Principal
------------------------------------------------------------------------*/

int main()
{
    Tlista lista = NULL;
    int op;     // opción del menu
    int _dato;  // elemenento a ingresar
int pos;    // posición a insertar


    system("color 0b");

    do
    {
        menu1();  cin>> op;

        switch(op)
{
            case 1:

cout<< "\n NÚMERO A INSERTAR: "; cin>> _dato;
                insertarInicio(lista, _dato);
            break;

            case 2:

                cout<< "\n NÚMERO A INSERTAR: "; cin>> _dato;
                insertarFinal(lista, _dato );
            break;

            case 3:

                cout << "\n\n MOSTRANDO LISTA\n\n";
                reportarLista(lista);
break;

system("pause");  system("cls");

    }while(op!=4);


system("pause");
   return 0;
}
```

Programa 1.4

Nota: Adaptado de "El Blog de Martín Cruz". Disponible en http://bit.ly/29PjO6J

TEMA N.º 2: ESTRUCTURAS DE CONTROL

Introducción al tema

En este y el siguiente tema se abordarán las denominadas estructuras de control. (Joyanes, 2005, p. 41) considera que "son métodos de especificar el orden en que las instrucciones de un algoritmo se ejecutarán. El orden de ejecución de las sentencias (lenguaje) o instrucciones determinan el flujo de control. Estas estructuras de control son, por consiguiente, fundamentales en los lenguajes de programación y en los diseños de algoritmos especialmente los pseudocódigos."

Las tres estructuras de control básicas son las siguientes:

- Secuencia
- Selección (o condicionada)
- Repetición

Aunque el nombre de este tema aparenta estar relacionado con el título de la asignatura, su contenido está ligado en mayor medida a los algoritmos. Las ventajas que ofrecen estas estructuras serán aprovechadas para implementar las operaciones a realizar sobre las listas, como son insertar, eliminar, buscar, etc.

El tema n.º 2 se concentra en la descripción e implementación de las estructuras de control condicionales.

1. ¿Por qué usar estructuras de control?

Al programar una computadora no se está haciendo más que traducir un algoritmo a un determinado lenguaje de programación. Internamente, el compilador lee una a una las líneas de código y las ejecuta en el mismo orden que fueron escritas.

Claramente, mientras se trate de un problema sencillo a resolver, el algoritmo correspondiente será secuencial, cada tarea se llevará a cabo una sola vez y en el mismo orden con el que fueron especificadas.

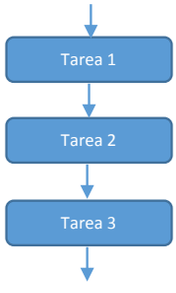
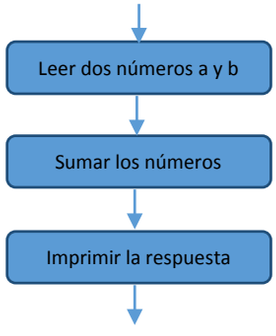


Figura 9. Representación de un algoritmo secuencial
Fuente: Autoría propia

Por ejemplo: el algoritmo para sumar dos valores sería el siguiente:



Lo cierto es que los problemas que resuelven los computadores son más complejos (con ese fin se crearon), por lo tanto, es necesario el empleo de algoritmos capaces de tomar decisiones, es decir, elegir qué tareas ejecutar, qué tareas repetir, etc. Para esto se cuenta con las denominadas estructuras de control, que permiten a los lenguajes de programación ejecutar ciertas instrucciones en el orden y momento especificados, en tiempo de ejecución.

Tales estructuras de control, pueden clasificarse de la siguiente manera:

- Estructuras condicionales
  - De control simple
  - De control doble
  - De control múltiple
- Estructuras repetitivas
  - Condicionada
  - Predefinida

2. La estructura de control condicionada simple

Este tipo de estructuras trabajan con las denominadas expresiones lógicas. Una expresión lógica es la comparación de dos datos (de cualquier tipo) empleando operadores matemáticos como igual, diferente, mayor que, menor que, mayor igual que y menor igual que.

Tal comparación puede ser interpretada como una pregunta literal. Por ejemplo:

EXPRESIÓN LITERAL	EQUIVALENTE MATEMÁTICO
¿5 es mayor que 3?	5>3
¿"Nombre" es igual a "NOMBRE"?	"Nombre" = "NOMBRE"
¿Verdadero es diferente que Falso?	Verdadero <> Falso
¿12/01/2016 es anterior a 15/01/2017?	'12/01/16' < '15/01/17'

Por lo tanto, tiene una respuesta, pudiendo ser esta: verdad o falso.

En palabras de Severance (2009, p. 33) "una expresión booleana es aquella que puede ser verdadera (True) o falsa (False)".

En consecuencia, las estructuras de control simple evalúan una expresión lógica dada, y si esta resulta verdadera, ejecuta determinadas acciones (una o varias); de lo contrario, las obvia, pasando a las siguientes instrucciones si las hubiere. La figura 10 ilustra lo expresado.

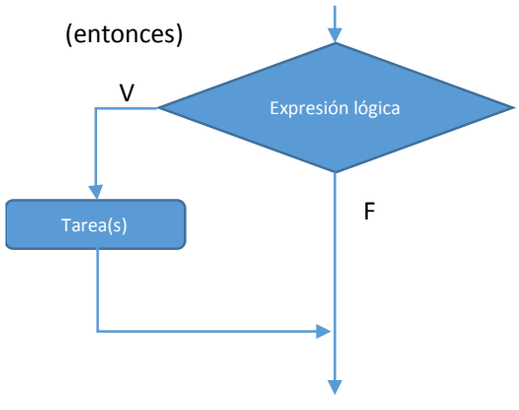


Figura 10. Representación de una estructura de control simple
Fuente: Elaboración propia

RECUERDA

"Verdadero y Falso son valores especiales que pertenecen al tipo *bool* (booleano); no son cadenas" (Severance, 2009, p. 33).

Su estructura en pseudocódigo es como se muestra a continuación:

Si (expresión lógica), entonces

{

Conjunto de instrucciones a realizar

}

Este tipo de estructura puede ser útil, por ejemplo, en un escenario en el que se desee restar el 10 % al valor de una venta, siempre y cuando esta sea mayor que 1000. Observe la figura 11.

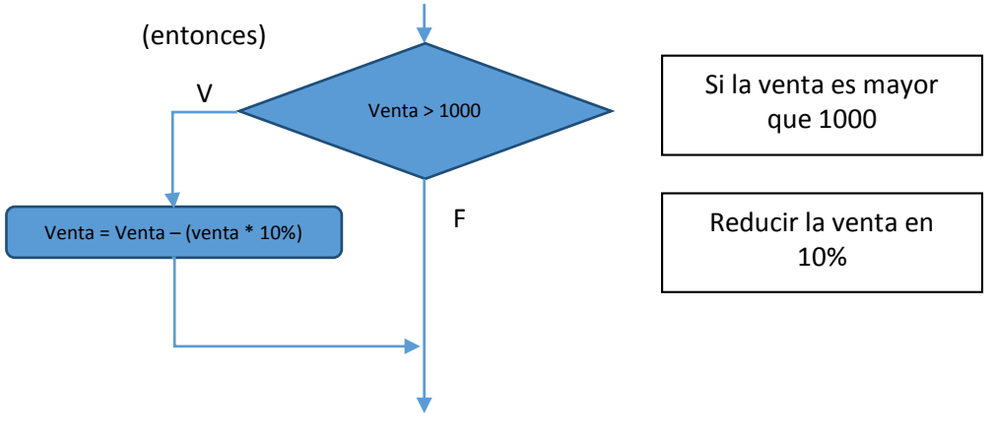


Figura 11. Ejemplo de estructura de control simple
Fuente: Elaboración propia

La variable "Venta" es contrastada con el operador ">" para verificar que sea mayor que 1000. Si se comprueba ello, inmediatamente se procede a reescribir la variable con el nuevo valor (descuento de 10 %).

El programa 2.1 muestra la implementación de este algoritmo en C++.

```
void main()
{
      float venta;
      cout<<"Digite un valor de venta:"<<endl;
      cin>>venta;
      if (venta > 1000)       // Expresión lógica o condición
      {
            venta = venta - (venta * 0.10);     // Tarea a realizar si
                                 // la expresión
                        // resulta verdadera
      }
      cout<<"El valor de la venta: "<<venta<<endl;
}
```

Programa 2.1

3. La estructura de control condicional doble

A diferencia de la estructura de control simple donde se escriben instrucciones únicamente para el caso en que la expresión lógica resulte verdadera, en una estructura de control doble se deben especificar instrucciones también para el caso en que resulte falsa.

Su estructura en pseudocódigo es como se muestra a continuación:

```
Si (expresión lógica), entonces
{
Conjunto de instrucciones "A" a realizar
}
Sino
{
Conjunto de instrucciones "B" a realizar
}
```

Gráficamente es como se muestra en la figura 13.

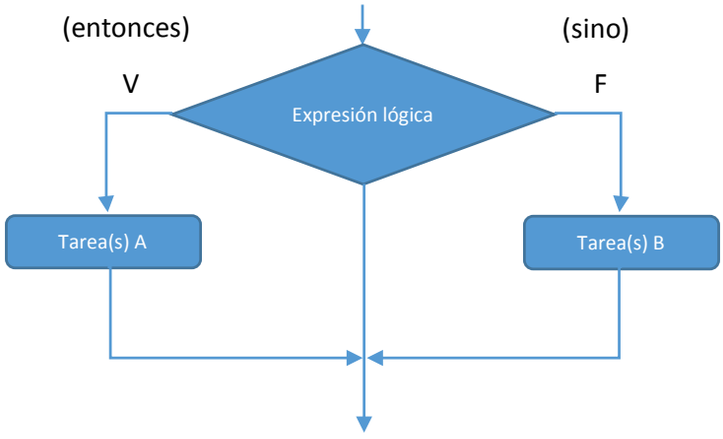


Figura 13. Representación de una estructura de control doble
Fuente: Elaboración propia

Ejemplo: Enviar un mensaje de alerta indicando si la edad ingresada corresponde a la de una persona mayor o menor de edad. Gráficamente sería:

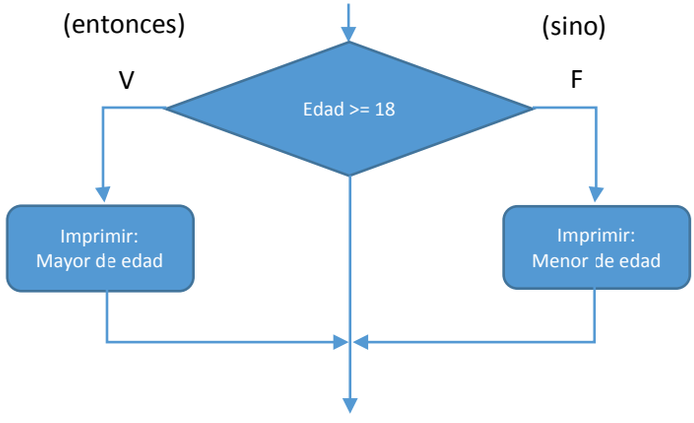


Figura 14. Ejemplo de una estructura de control doble
Fuente: Elaboración propia

El programa siguiente muestra su implementación:

```
void main()
{
      int edad;
      cout<<"Digite la edad de la persona"<<endl;
      cin>>edad;
      if (edad >= 18)  // Expresión lógica o condición
            cout<<"Mayor de edad"<<endl; // Expresión verdadera
      else
            cout<<"Menor de edad"<<endl; // Expresión falsa
}
```

Programa 2.2

4. La estructura de control múltiple

A diferencia de las dos estructuras anteriores, en las que la variable que participa de la expresión lógica se compara con un único valor, por ejemplo "edad>=18" o "sexo=Masculino", en una estructura de control múltiple la variable se compara simultáneamente con diferentes valores, y realiza aquellos bloques de instrucciones donde la comparación resulte verdadera.

Su estructura en pseudocódigo es como sigue:

Si (expresión lógica 1), entonces
{
Conjunto de instrucciones "A"
}
Sino, si (expresión lógica 2), entonces
{
Conjunto de instrucciones "B"
}
Sino, si (expresión lógica 3), entonces
{
Conjunto de instrucciones "C"
}
Sino,
{
Conjunto de instrucciones "n"
}

Gráficamente es como se muestra en la figura 15.

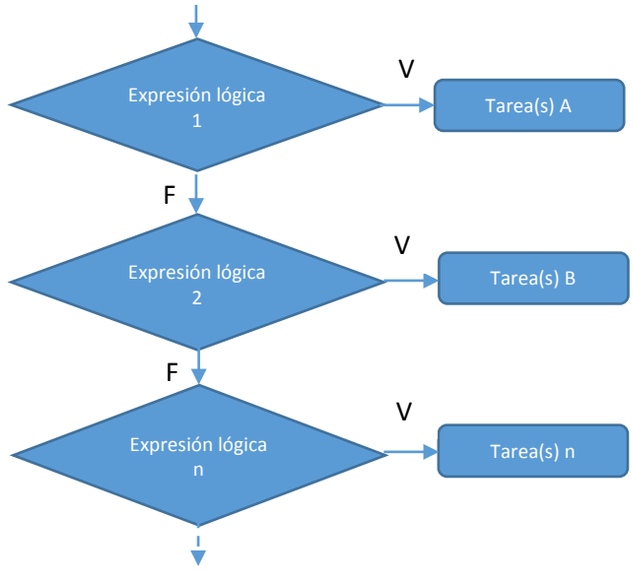


Figura 15. Representación de una estructura de control múltiple
Fuente: Elaboración propia

Ejemplo: convertir un número ingresado entre 1 y 5 a su vocal equivalente.

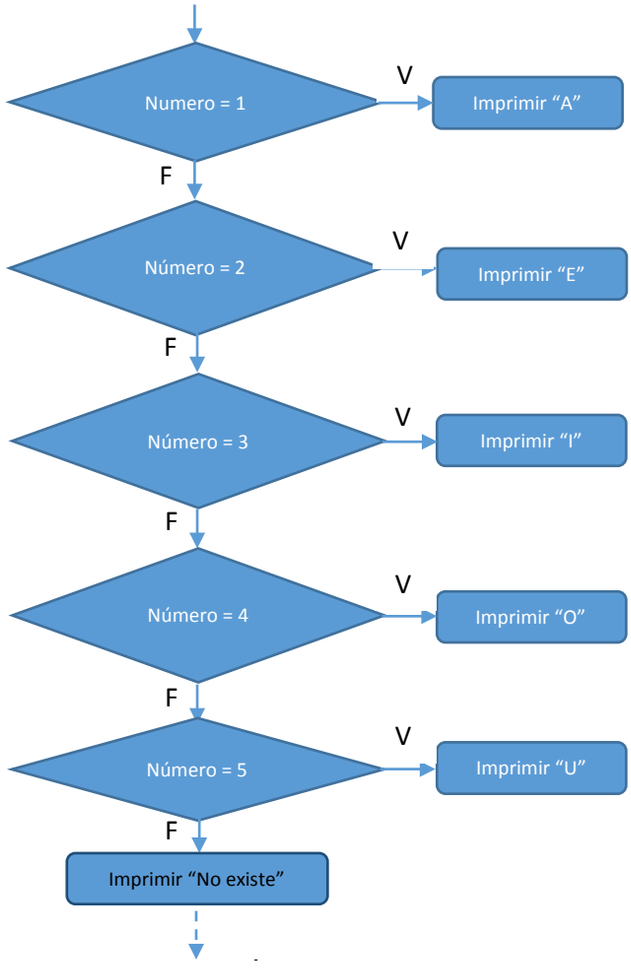


Figura 16. Ejemplo de una estructura de control múltiple
Fuente: Elaboración propia

Su versión en código de C++ es como se muestra en el programa 2.3.

```
#include <iostream>
#include <string>
using namespace std;

void main()
{
	int valor;
	string vocal;
	cout<<"Digite un valor: ";
	cin>>valor;
	if (valor == 1)
		vocal = "a";
	else if (valor == 2)
		vocal = "e";
	else if (valor == 3)
		vocal = "i";
	else if (valor == 4)
		vocal = "o";
	else if (valor == 5)
		vocal = "u";
	else
		vocal = "No existe";

	cout<<"La vocal es: "<<vocal<<endl;
	system("Pause");
}
```

Programa 2.3

Otra alternativa es emplear el operador *Switch-Case* como se aprecia en el siguiente programa:

```
#include <iostream>
#include <string>
using namespace std;

void main()
{
	int valor;
	string vocal;
	cout<<"Digite un valor: ";
	cin>>valor;
	switch(valor)
	{
	case 1: vocal = "a"; break;
	case 2: vocal = "e"; break;
	case 3: vocal = "i"; break;
	case 4: vocal = "o"; break;
	case 5: vocal = "u"; break;
	default: vocal = "No existe"; break; //Si se digita un
                                          valor
		//fuera del rango [1, 5]
	}
	cout<<"La vocal es: "<<vocal<<endl;
	system("Pause");
}
```

Programa 2.4

En C++ la instrucción *Switch-Case* ejecutará todos aquellos bloques de código donde la expresión lógica resulte verdadera. Por ello es necesario agregar la palabra reservada "Break" ("detener" en español) al final de cada instrucción.

TEMA N.º 3: ESTRUCTURAS DE CONTROL REPETITIVAS

Introducción al tema

La reutilización de código es una forma de optimizar tanto el tiempo de programación, el de mantenimiento y, por supuesto, el de ejecución de un aplicativo. En esa línea, las estructuras de control repetitivas permiten ejecutar un bloque de instrucciones tantas veces sea necesario o hasta que una condición dada se cumpla (o sea verdadera en términos técnicos).

Con lo expuesto a continuación, usted podrá conocer el funcionamiento y modo de implementación de este tipo de estructuras.

1. Estructuras de control repetitivas

Como ya se mencionó, este tipo de estructuras realizan una o varias tareas repetidas veces mientras se cumpla una condición dada.

Su estructura puede darse de dos formas dependiendo de la ubicación de la expresión lógica:

```
Hacer mientras (expresión lógica)
{
Conjunto de instrucciones
}
```

```
Hacer
{
Conjunto de instrucciones
}
Mientras (expresión lógica)
```

Gráficamente es como se muestra en la figura 17.

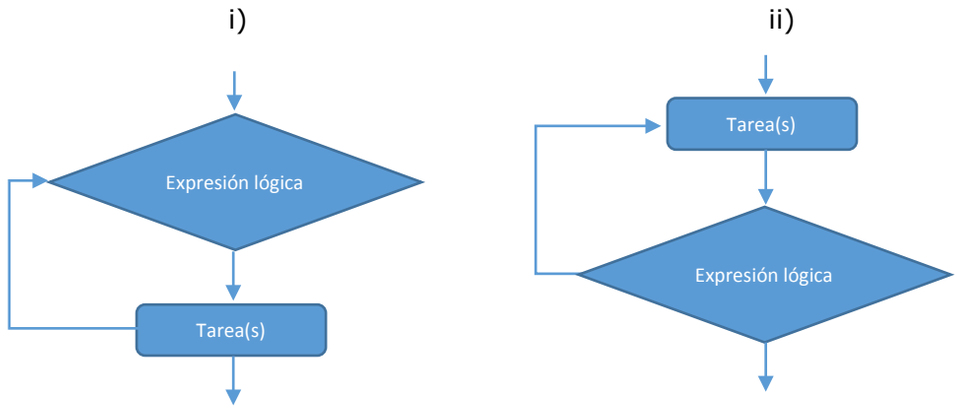


Figura 17. Representaciones de una estructura de control repetitiva condicionada
Fuente: Elaboración propia

C++ permite implementar tres tipos de estructuras repetitivas, cuya diferencia se describe en este cuadro:

Tabla 1. Tipos de estructuras repetitivas

INSTRUCCIÓN	DESCRIPCIÓN
For	El uso más frecuente de este tipo de bucles se produce cuando el número de repeticiones se conoce por anticipado y la condición del bucle puede ser controlada por un contador.
While	El uso más frecuente se produce cuando la repetición del bucle no está controlada por un contador, sino por una cierta condición (simple y compleja). El bucle puede que no se ejecute ninguna vez.
Do-while	Se utiliza en las mismas condiciones que el while y, además, cuando se debe asegurar que el bucle se ejecute al menos una vez. Ejemplo: menú de opciones con filtro.

Nota: Tomada de Práctica 5. Sentencias de control repetitivas, Fundamentos de informática. Recuperado de http://www.uhu.es/04004/material/Practica5_Curso0809.pdf

A continuación, se muestran ejemplos de implementación de estructuras repetitivas:

Ejemplo 1:

Imprimir una palabra en pantalla hasta que el usuario digite "Salir".

```
#include <iostream>.
using namespace std;

void main()
{
	string palabra;
	palabra = "";
	do
	{
		cout<<"Digite la palabra a imprimir: ";
		cin>>palabra;
		cout<<"La palabra es: "<<palabra<<endl;
	}
	while (palabra!="Salir");
	system("Pause");
};
```

Programa 3.1

Ejemplo 2:

Imprimir los divisores de un número especificado.

```
#include <iostream>
using namespace std;

void main()
{
	int número;
	int divisor;
	int i;
	cout<<"Indique número: ";
	cin>>número;
	divisor = número;
	i = 1;
	while (divisor > 0)
	{
		if (número%divisor==0) //El operador % devuelve el resto de
					//una división
		{
			cout<<"Divisor["<<i<<"]: "<<divisor<<endl;
			i++; //Incrementar en una unidad
		}
		divisor--; //Disminuir en una unidad
	}
	system("Pause");
};
```

Programa 3.2

2. La instrucción FOR

La instrucción FOR es una variante de estructura repetitiva, la cual puede usarse cuando se conoce de antemano la cantidad de repeticiones que debe tener el bucle.

La figura 18 describe su estructura:

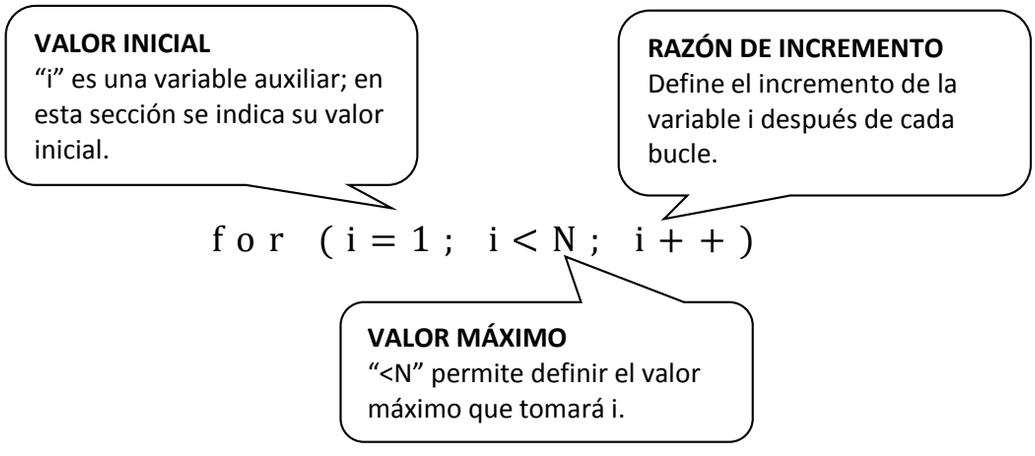


Figura 18. Representaciones de una estructura de control repetitiva condicionada
Fuente: Autoría propia

En los siguientes ejemplos se detalla cómo implementar esta instrucción:

Ejemplo 3:

Imprimir una serie numérica hasta un número especificado.

```
void main()
{
	int i;
	int cantidad;
	cout<<"Indique cantidad: ";
	cin>>cantidad;
	for (i=1;i<=cantidad;i++)
		cout<<i<<endl;
	system("Pause");
};
```

Programa 3.3

Ejemplo 4:

Imprimir la tabla de multiplicar de un número dado. Note que en este caso la declaración de la variable auxiliar "i" se realiza en la misma instrucción FOR.

```
void main()
{
	int numero;
	cout<<"Indique número de reportar: ";
	cin>>numero;
	for (int i=0;i<=10;i++)
		cout<<número<<"*"<<i<<"="<<número*i<<endl;
	system("Pause");
};
```

Programa 3.4

Es posible anidar dos instrucciones FOR para realizar recorridos simultáneos. En el programa 3.5 se aprecia cómo aprovechar esta cualidad para realizar una impresión de datos en forma tabular.

```
#include <iostream>
using namespace std;

void main()
{
	int columnas;
	int filas;
	columnas = 9;
	filas = 5;

	for (int i=0; i<=filas; i++)
		for (int j=0; j<=columnas; j++)
		{
			cout<<i<<j<<" ";
			if (j==columnas)
				cout<<endl; //Salto de línea si se llegó a la última colum-
na
		}
	system("Pause");
};
```

Resultado:

00	01	02	03	04	05	06	07	08	09
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59

Como se verá a partir de la siguiente unidad, los bucles son ideales para recorrer por cada uno de los elementos de un arreglo (de una o más dimensiones) mientras se hacen operaciones con o sobre ellos.

Lectura seleccionada n.º 1:

El concepto de datos estructurados

Leer el apartado 5.1.: "El concepto de datos estructurados" (p. 171)

Quetglás, G., Toledo, F., & Cerverón, V. (2002). Estructura de datos. En *Fundamentos de Informática y Programación* (pp. 171–211). Disponible en http://robotica.uv.es/pub/Libro/PDFs/CAPI5.pdf

Actividad N.º 1

Participe en un foro de discusión sobre listas basadas en arreglos y basadas en punteros.

Instrucciones:

a) **Lea y analice detenidamente los siguientes casos:**

Se desea implementar dos programas que, mediante listas, permitan el registro de

i. Los nombres de los jugadores participantes de un partido de futbol. Solo un partido, una vez finalizado este, los datos pueden ser eliminados.

ii. Las placas de los vehículos que llegan a diario a una caceta de control policial en carretera. Una vez terminado el día, la lista puede ser eliminada.

b) **Basado en su análisis, responda en el foro:**

Entre los dos tipos de listas: basada en arreglos y basada en punteros, ¿cuál de ellas se adecúa mejor a cada caso? Fundamente su respuesta.

Glosario de la Unidad I

A

Algoritmo. Conjunto ordenado de pasos que llevan a la solución de un problema.

C

C++. Es un lenguaje de programación diseñado en la década de 1980. A diferencia de su predecesor C, este permite manipular objetos.

D

Dato. Unidad mínima de información que carece de significado por sí misma.

I

Información. Conjunto de datos ordenados y con sentido semántco.

M

Memoria (en informática). Es un dispositivo que almacena datos informáticos durante algún intervalo de tiempo.

P

Procesador. De la familia de los circuitos integrados, es un dispositivo electrónico responsable de convertir datos de entrada en datos de salida.`I + I

Programa (en informática). Conjunto de instrucciones que realizan una tarea específica.

V

Variable. En un lenguaje de programación es un espacio de memoria, identificado por un nombre y reservado para almacenar un dato específico a la vez.

Bibliografía de la Unidad I

Cairo, Osvaldo y Guardati, Silvia. (2010). *Estructuras de datos* (3.ª ed.). Editorial McGraw Hill.

Charles, Severance. (2009). *Python para informáticos: Explorando la información*.

Cruz Huerta, Dora. (s.f.). *Apuntes de la asignatura: Estructura de datos*. México: Tecnológico de Estudios Superiores de Ecatepec. Disponible en http://myslide.es/documents/manual-estructura-de-datos.html

Joyanes, Luis & Zahonero, Ignacio. (2005). *Programación en C: Metodología, algoritmos y estructura de datos*. España: McGraw Hill / Interamericana

Sahni, Sartaj. (2005). *Data structures, algorithms, and applications in C++* (2ª. ed.). Hyderabad, India: Editorial Silicon Press.

Universidad de Valencia, Instituto Universitario de Investigación de Robótica y Tecnologías de la Información y Comunicación. (s.f.) *Estructura de datos*. Valencia, España: IRTIC. Disponible en http://robotica.uv.es/pub/Libro/PDFs/CAPI5.pdf.

Weiss, Mark Allen. (2000). *Estructura de datos en Java: Compatible con Java 2.* Madrid: Pearson Education.

Autoevaluación n.º 1

1. ¿Qué es un dato?

   a. Conjunto de información

   b. Unidad mínima de información

   c. Valores de un arreglo

   d. Forma de representar información

2. ¿Cuál es la diferencia entre un dato simple y un dato estructurado?

   a. Los datos simples ocupan solo una casilla de memoria; los datos estructurados hacen referencia a un conjunto de casillas de memoria identificadas con un nombre único.

   b. Los datos simples ocupan varias casillas de memoria; los datos estructurados hacen referencia a un único valor a la vez.

   c. Los datos simples ocupan solo una casilla de memoria; los datos estructurados son similares a los simples, pero cada uno de ellos opera con su propio nombre.

   d. Los datos simples ocupan varias casillas de memoria; los datos estructurados son similares, pero a diferencia de esos están interrelacionados entre sí.

3. ¿Cuáles de las siguientes listas son ejemplos de estructuras de datos?

   a. Arreglos, grafos, datos

   b. Grafos, punteros, arreglos

   c. Arreglos, colas, pilas

   d. Colas, pilas, bucles

4. ¿Cuál es diferencia entre una lista lineal y una no lineal?

   a. En la lista no lineal, cada elemento es precedido solo por un elemento. En una lineal, dos elementos pueden compartir un elemento que les precede.

   b. En la lista lineal, cada elemento es precedido solo por un elemento. En una no lineal, dos elementos pueden compartir un elemento que les precede.

   c. En la lista lineal, los elementos no tienen elementos predecesores. En una no lineal, los elementos pueden tener varios elementos predecesores.

   d. En la lista no lineal, los elementos no tienen elementos predecesores. En una lineal, los elementos pueden tener varios elementos predecesores.

5. ¿En qué consiste una lista basada en arreglos?

   a. Es aquella que emplea elementos dispuestos en línea.

   b. Es aquella que emplea arreglos para disponer los elementos.

   c. Es aquella que emplea punteros para disponer los elementos.

   d. Es aquella que emplea punteros hacia arreglos para disponer los elementos.

6. ¿En qué consiste una lista basada en punteros?

   a. Es aquella que emplea elementos dispuestos en línea.

   b. Es aquella que emplea nodos para disponer los elementos.

   c. Es aquella que emplea nodos basados en punteros para disponer los elementos.

   d. Es aquella que emplea punteros hacia arreglos para disponer los elementos.

7. ¿Qué es una expresión lógica?

   a. Es una comparación matemática que tiene dos posibles respuestas: verdadero y falso.

   b. Es una comparación literal que tiene múltiples respuestas, pudiendo ser estas de cualquier tipo de datos.

   c. Es una comparación matemática que tiene múltiples respuestas, pudiendo ser esta de tipo lógico o booleano.

   d. Es una comparación literal que tiene dos posibles respuestas: verdadero o falso.

8. ¿Cómo opera una estructura condicional?

   a. Trabaja con una expresión lógica, y según su respuesta repite una instrucción un número indeterminado de veces.

   b. Resuelve una expresión lógica: si resulta verdadera, realiza una determinada instrucción; si resulta falsa, realiza otra.

   c. Trabaja con una expresión lógica: si resulta verdadera, repite una determinada instrucción un número determinado de veces.

   d. Resuelve una expresión lógica: si resulta verdadera, realiza una determinada tarea; si resulta falsa, realiza otra.

9. ¿Cuál es la diferencia entre emplear la sentencia "if…Else if" y la sentencia "Switch" en C++?

   a. La sentencia "*if…Else if*" realiza solo aquella tarea donde la expresión lógica resultó falsa, mientras que la instrucción "*Switch*" realiza todas aquellas tareas donde la expresión lógica resultó verdadera.

   b. La sentencia "*if…Else if*" realiza todas aquellas tareas donde la expresión lógica resultó verdadera, mientras que la instrucción "*Switch*" realiza todas aquellas tareas donde la expresión lógica resultó falsa.

c. La sentencia "*if…Else if*" realiza solo aquella tarea donde la expresión lógica resultó verdadera, mientras que la instrucción "*Switch"* realiza todas aquellas tareas donde la expresión lógica resultó verdadera.

d. La sentencia "Switch" realiza solo aquella tarea donde la expresión lógica resultó verdadera, mientras que la instrucción "*if…Else if"* realiza todas aquellas tareas donde la expresión lógica resultó verdadera.

10. ¿Cómo opera una estructura repetitiva?

a. Repite una o varias instrucciones un número indeterminado de veces.

b. Repite una o varias instrucciones un número determinado de veces.

c. Repite una o varias instrucciones mientras una condición sea verdadera.

d. Repite una o varias instrucciones mientras una condición sea falsa.

UNIDAD II

ARREGLOS Y MATRICES

DIAGRAMA DE ORGANIZACIÓN DE LA UNIDAD II

ORGANIZACIÓN DE LOS APRENDIZAJES

Resultado de aprendizaje de la Unidad II:

Al finalizar la unidad, el estudiante será capaz de seleccionar las estructuras de arreglos y matrices, según la problemática propuesta.

CONOCIMIENTOS	HABILIDADES	ACTITUDES
Tema n.º 1: Arrays unidimensionales, bidimensionales y multidimensionales. Operaciones con ellos. **Tema n.º 2: Matrices** **Lectura seleccionada n.º 2:** **Autoevaluación de la Unidad II**	• Identifica y realiza operaciones con arrays unidimensionales. • Describe y resuelve problemas utilizando los arrays bidimensionales. • Desarrolla problemas aplicados a los arrays multidimensionales. • Resuelve las prácticas de laboratorio. **Actividad N.º 2** Foro de discusión	• Demuestra perseverancia y esfuerzo durante el desarrollo de los ejercicios. • Toma conciencia de la importancia de la asignatura en su formación profesional. • Valora las relaciones entre sus compañeros.

TEMA N.º 1:
ARRAYS UNIDIMENSIONALES, BIDIMENSIONALES Y MULTIDIMENSIONALES. OPERACIONES CON ELLOS

INTRODUCCIÓN

En la primera unidad de este módulo, se describieron los fundamentos y aplicaciones de los arreglos (o *array* por su denominación en inglés). A continuación, se tratará sobre su implementación, para lo cual se emplean ejemplos con lenguaje C++ que demuestran desde cómo declararlos (crearlos) hasta las posibles operaciones a realizar con ellos. Finalmente, se abordarán los arreglos multidimensionales tanto simples como basados en punteros.

1. Definición

Un arreglo "se define como una colección finita, homogénea y ordenada de elementos" (Cairo & Guardati, 2010, p. 4). A continuación, se explicará brevemente una de estas características.

- Finita: Todo arreglo tiene un límite; es decir, se debe determinar cuál será el número máximo de elementos que formarán parte del arreglo.
- Homogénea: Todos los elementos de un arreglo son del mismo tipo. Por ejemplo, todos serían enteros o booleanos, entre otros.
- Ordenada: Se puede determinar de manera sistemática el ordenamiento de los elementos; en otras palabras, establecer cuál es el primero, el segundo, el tercer elemento, etc.

Asimismo, un arreglo puede tener una o más dimensiones para almacenar los datos como se aprecia en la siguiente figura:

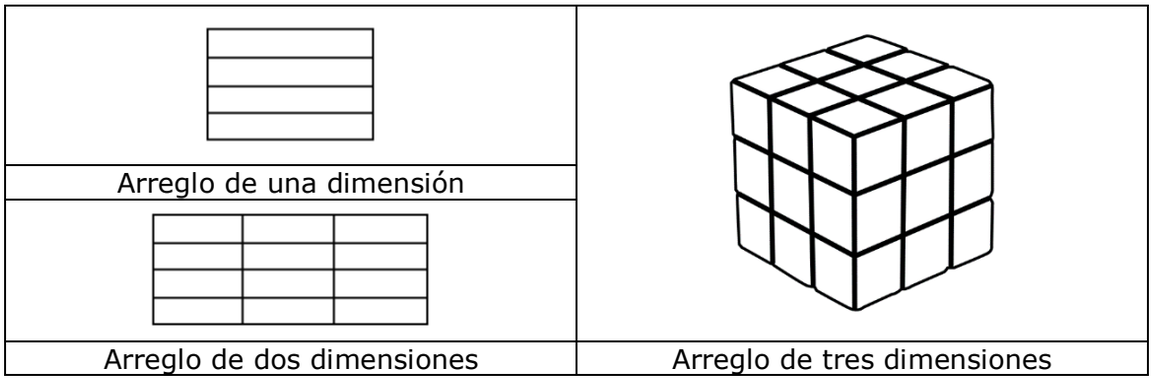


Figura 19. Representación gráfica de las dimensiones de un arreglo

Aunque la representación gráfica de cuatro a más dimensiones se dificulta, un arreglo puede soportar muchas más en función del lenguaje de programación que se utilice. Por ejemplo, C++ (de Visual Studio 2012) admite hasta 29 dimensiones para arreglos de tipo entero.

RECUERDA:

Desde el punto de vista técnico, una variable simple es un espacio de memoria que opera bajo un determinado nombre, donde se puede almacenar un solo dato a la vez, mientras que un arreglo es un conjunto de espacios de memoria identificados por un mismo nombre.

Dado que los elementos de un arreglo se identifican con un nombre en común, para hacer referencia a alguno de ellos en particular, será necesario emplear un índice que trabaja a modo de coordenada e identifica unívocamente a cada casilla del arreglo.

2. Creación de arreglos

En C++, al igual que cualquier otra variable, es necesario declarar (o crear) un arreglo antes de utilizarlo. El programa 1.1 muestra cómo hacerlo tal como se aprecia a continuación:

```
void main()
{
    int arregloUd[5]; // Arreglo unidimensional que permite
       almacenar hasta 5 números enteros
    bool arregloUd[10]; // Arreglo unidimensional que permite
            almacenar hasta 10 datos lógicos
    int arregloBd[4][3]; // Arreglo bidimensional que permite
            almacenar hasta 12 números enteros
    char arregloTd[3][2][4]; // Arreglo tridimensional que permite
                  almacenar hasta 24 caracteres
    // Arreglo de punteros
    // a. Arreglo de punteros simple
    char *días[7] = {"Lun", "Mar", "Mie", "Jue", "Vie", "Sab", "Dom"};

    // b. Arreglo de punteros doble
    int **tablero = new int *[4];
}
```

Programa 1.1

Como puede apreciarse, la declaración de un arreglo implica el uso de los caracteres [] (corchetes) seguido del nombre. Cabe decir que entre los corchetes se digita el número máximo de elementos que tendrá cada dimensión.

Un arreglo de punteros simple almacena como elementos direcciones de memoria a variables; mientras que el arreglo a punteros doble, r direcciones de memoria a otros arreglos. Este último se describe al detalle más adelante (en el numeral 5).

3. Operaciones con arreglos

3.1. Leer y escribir datos

Cuando Sahni (2005, p. 223) define la representación abstracta de un arreglo, considera únicamente dos operaciones sobre él: leer (*get*) y escribir (*set*). Observe la siguiente representación:

Arreglo
{
Instancias
Conjunto de pares de la forma (índice, valor)
Operaciones
Get (índice) : retorna el valor del par cuyo índice es proporcionado.
Set (índice, valor) : agrega un par al arreglo en la posición del índice proporcionado. Si existe un valor en esa posición será sobrescrito.
}

Figura 20. Representación abstracta de un arreglo
Fuente: Autoría propia

Por ejemplo, considere la opción de crear un arreglo que permita registrar la temperatura máxima de cada día de la semana.

TemperaturaMax = {(lunes, 82), (martes, 79), (miércoles, 85), (jueves, 92), (viernes, 88), (sábado, 89), (domingo, 91)}

Ya que los índices de un arreglo deben ser de tipo numérico, la representación será tal como se aprecia a continuación:

TemperaturaMax = {(0, 82), (1, 79), (2, 85), (3, 92), (4, 88), (5, 89), (6, 91)}

a) Para actualizar el valor del día lunes a 85 la operación será set(0, 85)

b) Para recuperar el valor del día viernes la operación será get(4)

El siguiente programa muestra -de forma muy básica- como registrar (*set*) e imprimir (*get*) los elementos de un arreglo.

```
using namespace std;

void main()
{
	// Declarar un arreglo unidimensional de enteros con capacidad de 5 elementos
	int arregloU[5];
	// Guardar los enteros 14 y 23 en las dos primeras posiciones del arreglo
	arregloU[0] = 14;
	arregloU[1] = 23;
	// Guardar números digitados por el usuario en las posiciones 3, 4 y 5 del
arreglo
	cout<<"Ingrese el tercer elemento: ";cin>>arregloU[2];
	cout<<"Ingrese el cuarto elemento: ";cin>>arregloU[3];
	cout<<"Ingrese el quinto elemento: ";cin>>arregloU[4];
	// Imprimir los elementos del arreglo
	cout<<endl;
	cout<<"Elementos del arreglo"<<endl;
	cout<<"----------------------"<<endl;
	cout<<"Primer elemento: "<<arregloU[0]<<endl;
	cout<<"Segundo elemento: "<<arregloU[1]<<endl;
	cout<<"Tercer elemento: "<<arregloU[2]<<endl;
	cout<<"Cuarto elemento: "<<arregloU[3]<<endl;
	cout<<"Quinto elemento: "<<arregloU[4]<<endl;
	system("pause");
}
```

Programa 1.2

Por su parte, el programa 1.3 implementa los métodos *get* () y *set* () propiamente, tal como se describe en la representación abstracta:

```
#include "iostream"
using namespace std;

// Declarar una constante que permitirá definir la capacidad del arreglo
const int CapacidadMax=5;

// Declarar un arreglo de enteros con capacidad de 5 elementos
int arregloU[CapacidadMax];
// Procedimiento para registrar un elemento en el arreglo
 void set(int índice, int valor)
 {
 arregloU[índice] = valor;
 }

// Función que retorna un elemento del arreglo
int get(int índice)
{
      return arregloU[índice];
}

void main()
{
      int valor;
      // Registrar los elementos del arreglo
      for (int i=0; i<CapacidadMax; i++)
      {
            cout<<"Ingrese el elemento["<<i<<"]: ";cin>>valor;
            set(i,valor);
      }

      // Imprimir los elementos del arreglo
      cout<<endl;
      cout<<"Elementos del arreglo"<<endl;
      cout<<"----------------------"<<endl;
      for (int i=0; i<CapacidadMax; i++)
            cout<<"Elemento["<<i<<"]: "<<get(i)<<endl;

      system("pause");
}
```

Programa 1.3

Aunque la representación de Sahni toma en cuenta solo dos operaciones, Cairo, O. y Guardati S. (2010, p. 7) consideran adicionalmente las siguientes: Lectura, escritura, asignación, actualización (inserción, eliminación, modificación), ordenación y búsqueda.

Debido a que las primeras operaciones de la lista son variantes de *get ()* y *set ();* a continuación, nos enfocaremos en las de ordenación y búsqueda.

3.2. Ordenar datos

Existen muchos algoritmos para ordenar los elementos de un arreglo. En las siguientes líneas, en términos generales, se describe uno de ellos:

a) De forma ordenada capturar uno a uno los valores del arreglo e ir comparándolos con los restantes en busca del valor más alto.

b) Tal valor máximo toma la posición del dato que fue capturado inicialmente y viceversa.

Para su mejor entendimiento observe la figura 21:

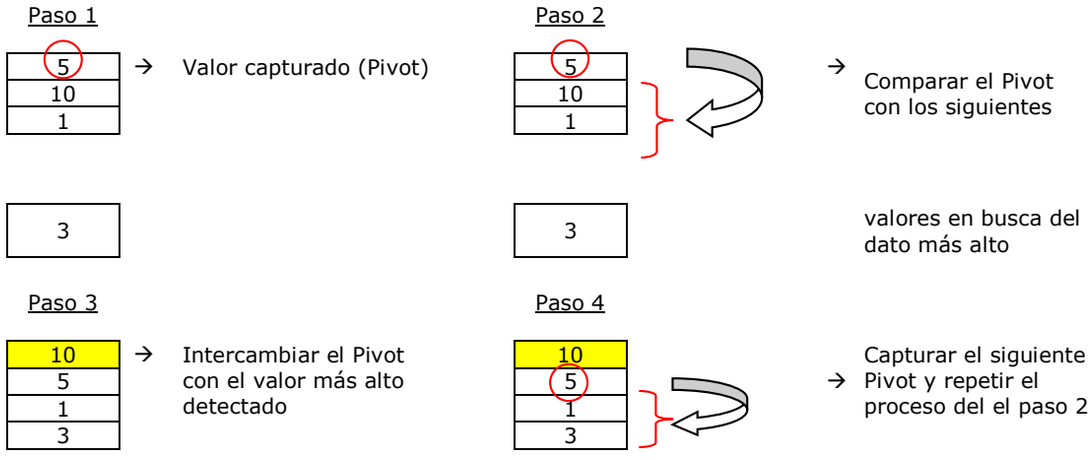


Figura 21. Pasos para ordenar los elementos de un arreglo

En código C++:

```
#include "iostream"
using namespace std;

const int MAX = 4;
int arreglo[MAX];

void imprimirArreglo()
{
   cout<<endl;
   cout<<"Valores del arreglo"<<endl;
   cout<<"--------------------"<<endl;
   for (int i=0; i<MAX; i++)
         cout<<"Elemento ["<<i<<"]: "<<arreglo[i]<<endl;
}

 void main()
{
   int valor;
   int valorMAX; // Valor máximo detectado
   int posMAX; // Posición del valor máximo detectado

   //Registrar datos
```

```
        cout<<endl;
        cout<<"Ingresando valores al arreglo"<<endl;
        cout<<"-----------------------------"<<endl;
        for (int i=0; i<MAX; i++)
        {
              cout<<"Arreglo["<<i<<"]: ";
              cin>>arreglo[i];
        }

        // Imprimir arreglo
        imprimirArreglo();

        // Ordenar los elementos
        for (int i=0; i<MAX; i++)
        {
              valorMAX = arreglo[i];
              posMAX = i;

              for (int j=i; j<MAX; j++)
              {
                    if (valorMAX < arreglo[j])
                    {
                          valorMAX = arreglo[j];
                          posMAX = j;
                    }
              }
              // Intercambiar Pivot y valorMAX
              valor = arreglo[i];
              arreglo[i] = valorMAX;
              arreglo[posMAX] = valor;
        }

        // Imprimir arreglo
        imprimirArreglo();
        system("Pause");
  }
```

Programa 1.4

3.3. Búsqueda de datos

La operación de búsqueda deberá retornar el índice del dato encontrado. El siguiente programa busca un dato y en caso de no encontrarlo retorna -1.

```
#include "iostream"
using namespace std;

const int MAX = 4;
int arreglo[MAX];

// Función de búsqueda
int buscar(int datoBuscado)
```

```
{
      int índice=-1;
      for (int i=0; i<MAX; i++)
            if (arreglo[i] == datoBuscado)
                  índice = i;
      return índice;
}

void main()
{
      int DatoABuscar;

      //Registrar datos
      cout<<endl;
      cout<<"Ingresando valores al arreglo"<<endl;
      cout<<"----------------------------"<<endl;
      for (int i=0; i<MAX; i++)
      {
            cout<<"Arreglo["<<i<<"]: ";
            cin>>arreglo[i];
      }

      cout<<endl;
      cout<<"Especifique el valor que desea buscar: ";cin>>DatoABuscar;

      if (buscar(DatoABuscar)==-1)
            cout<<"Dato no encontrado"<<endl;
      else
            cout<<"El dato se encuentra en la posición "<<buscar(DatoABuscar)<<endl;
      system("Pause");
}
```

Programa 1.5

4. Conversión de arreglos multidimensionales a unidimensionales

A fin de simplificar la programación de algunas aplicaciones, es posible representar un arreglo de varias dimensiones en uno unidimensional.

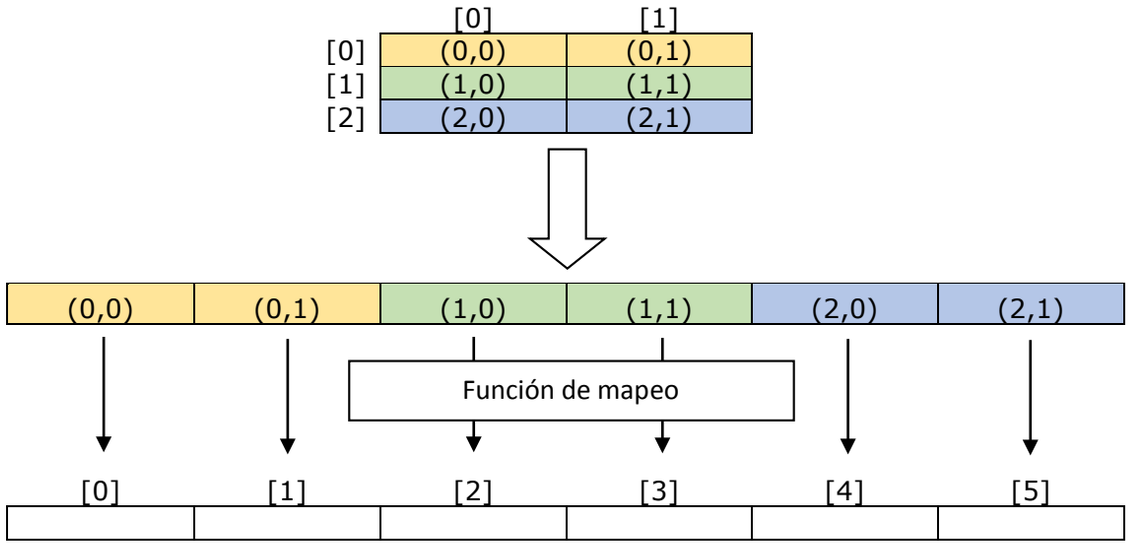
Al acto de convertir el índice de un arreglo multidimensional a un valor único se denomina mapeo y a la operación matemática que la realiza, "Función de mapeo".

RECUERDE:

El índice de un arreglo multidimensional está representado por varios números en forma de coordenadas.

4.1. Mapeo por filas

Consiste en descomponer el arreglo por filas y disponerlas una a continuación de otra como se aprecia en la figura 22.



La función de mapeo es $map(i_f, i_c) = i_f C + i_c$

Donde:
if = índice de fila
ic = índice de columna
C = número de columnas del arreglo (No confundir con el índice máximo)

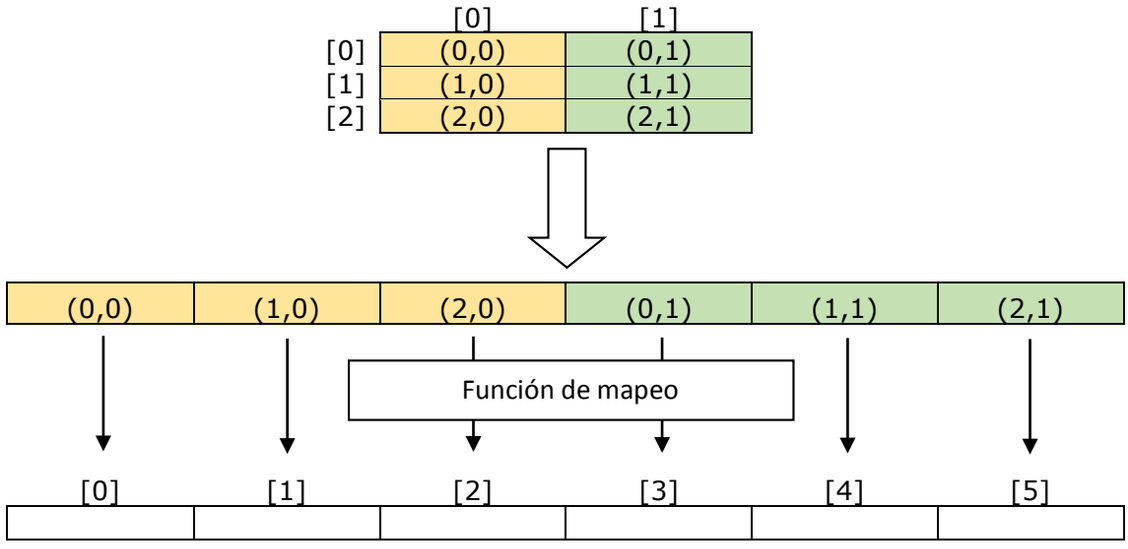
Por ejemplo:

a. $map(2,1) = 2(2) + 1 = 5$
b. $map(0,1) = 0(1) + 1 = 1$
c. $map(1,1) = \_(\_) + \_\_ = \_\_$
d. $map(2,0) = \_(\_) + \_\_ = \_\_$

Figura 22. Secuencia de mapeo por filas

4.2. Mapeo por columnas

Por el contrario, en este tipo de mapeo se descompone el arreglo por columnas y se disponen sus elementos, tomándolos de arriba abajo, uno a continuación del otro. Observe la figura 23.



En este caso, la función de mapeo $map(i_f, i_c) = i_c F + i_f$

Donde:
if = índice de fila
ic = índice de columna
F = número de filas del arreglo (No confundir con el índice máximo)

Por ejemplo:

a. $map(2,0) = \_(\_) + \_\_ = \_\_$
b. $map(2,1) = \_(\_) + \_\_ = \_\_$
c. $map(0,1) = \_(\_) + \_\_ = \_\_$
d. $map(1,1) = \_(\_) + \_\_ = \_\_$

Lo propio se puede hacer con un arreglo de tres dimensiones. Su función de mapeo se representa de la siguiente manera:

$$map(i_x, i_y, i_z) = i_x YZ + i_y Z + i_z$$

Figura 23. Secuencia de mapeo por columnas

El programa 1.6 muestra cómo simular un arreglo bidimensional empleando uno unidimensional (mapeo por filas).

```
#include <iostream>
using namespace std;

// Número de filas
const int NroFilas = 4;
// Número de columnas
const int NroColumnas = 3;
// Declaración del arreglo
int arreglo[NroFilas * NroColumnas];
```

```
// Método para insertar un valor en el arreglo
void set(int iFil, int iCol, int valor)
{
      // Conversión de índice mediante la función de mapeo
      int indice = iFil * NroColumnas + iCol;
      // Asignación del valor
      arreglo[índice] = valor;
}

// Función que recuperar un valor del arreglo
int get(int iFil, int iCol)
{
      // Conversión de índice mediante la función de mapeo
      int índice = iFil * NroColumnas + iCol;
      // Retornar el valor del arreglo según el índice calculado
      return arreglo[índice];
}

// PROGRAMA PRINCIPAL
void main()
{
      // Asignación de valores
      set(0,1,5);
      set(1,1,7);
      set(2,0,10);

      // Lectura de valores
      cout<<get(0,1)<<endl;
      cout<<get(1,1)<<endl;
      cout<<get(2,0)<<endl;

      system("Pause");
}
```

Programa 1.6

5. Arreglo de arreglos

Matrices bidimensionales basadas en punteros

Otra forma de generar un arreglo bidimensional es creando un arreglo de arreglos. El siguiente ejemplo de un *array* de 3 filas por 4 columnas es muestra de lo mencionado anteriormente.

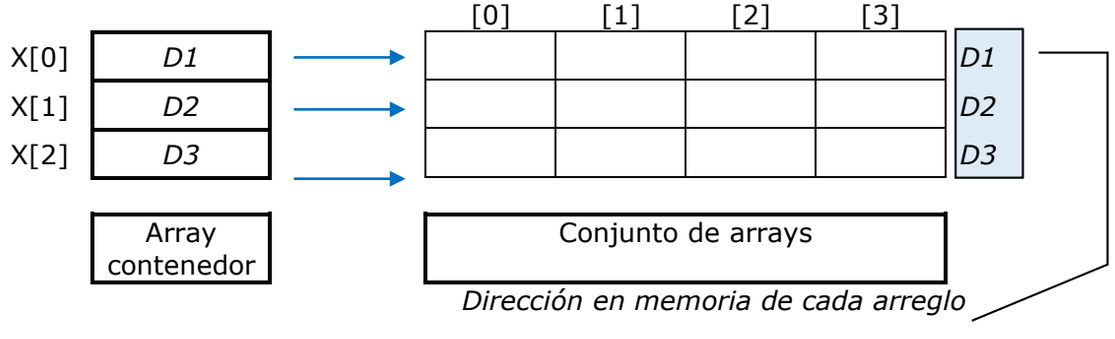


Figura 24. Estructura en memoria de un arreglo bidimensional basado en punteros

Pero para que esto sea posible, deberá declarar un *array* contenedor de tipo puntero doble (ver numeral 2) y, a continuación, almacenar en cada casilla la dirección de memoria de cada subarreglo. El programa 1.7 muestra cómo realizarlo en las siguientes líneas:

```
#include "iostream"
using namespace std;

// Tamaño del arreglo contenedor;
// También define el número de filas del arreglo bidimensional
const int NroFilas = 3;

// Tamaño de cada subarreglo;
// También define el número de columnas del arreglo bidimensional
const int NroColumnas = 4;

// Programa principal
void main()
{
      // Declarar el arreglo contenedor
      int **arrayBidimensional = new int *[NroFilas];

      // Asignar un subarreglo a cada casilla del arreglo contenedor
      for (int i=0; i<NroFilas; i++)
            arrayBidimensional[i] = new int [NroColumnas];

      // Asignar algunos valores al arreglo
      arrayBidimensional[0][0] = 4;
      arrayBidimensional[1][2] = 7;
      arrayBidimensional[2][3] = 9;

      // Imprimir los valores asignados
      cout<<endl;
      cout<<arrayBidimensional[0][0]<<endl;
      cout<<arrayBidimensional[1][2]<<endl;
      cout<<arrayBidimensional[2][3]<<endl;
      system("Pause");
}
```

Programa 1.7

TEMA N.º 2: MATRICES

1. Definición

En estructura de datos, una matriz es un tipo especial de arreglo bidimensional. Se diferencia principalmente por las operaciones que se pueden realizar con ella.

De acuerdo con Sahni (2005), una matriz de m x n es una tabla con m filas y n columnas; son usadas a menudo para organizar data. Por su parte, Pérez (2004 refuerza esta idea al expresar que "las matrices se utilizan (…) como elementos que sirven para clasificar valores numéricos atendiendo a dos criterios o variables" (p. 84). Además, diferencia entre matriz de información y matriz de relación. La primera simplemente recoge datos de un problema determinado como el mostrado en el ejemplo 1.

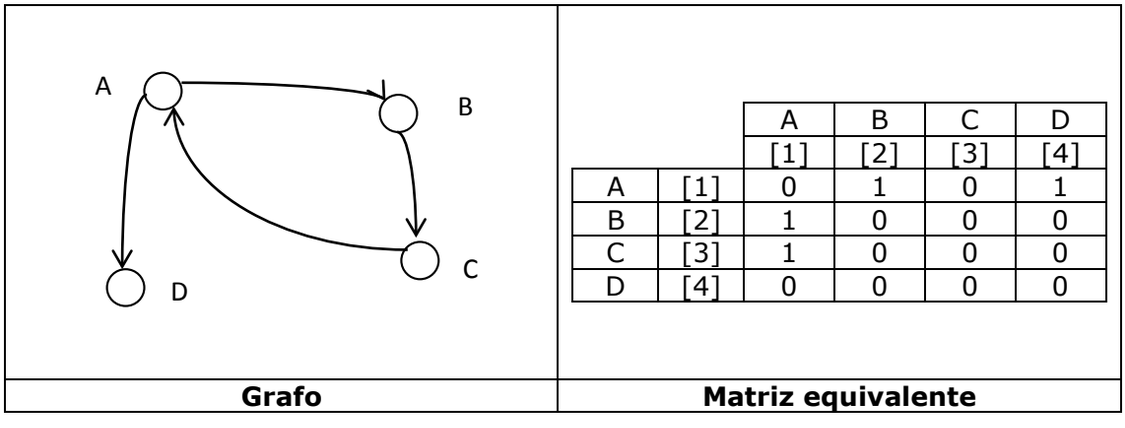
Ejemplo 1. Mantener un registro de la cantidad de minerales que producen anualmente tres países (miles de toneladas):

			País		
			País 1	País 2	País 3
			[1]	[2]	[3]
Mineral	Oro	[1]	10	11.3	10
	Plata	[2]	7.5	9	5.4
	Cobre	[3]	23	12.4	9

Figura 25. Ejemplo de una matriz de información

Por su parte, la matriz de relación indica si ciertos elementos están o no relacionados entre sí. Tal relación se expresa con 1 (uno) y su ausencia con 0 (cero). Estos números se usan para la trasladar la información dada por un grafo. Observe el ejemplo dado en la siguiente figura:

Ejemplo 2.



		A	B	C	D
		[1]	[2]	[3]	[4]
A	[1]	0	1	0	1
B	[2]	1	0	0	0
C	[3]	1	0	0	0
D	[4]	0	0	0	0

Grafo | **Matriz equivalente**

Figura 26. Ejemplo de una matriz de relación

Cabe decir que sobre los grafos se tratará al detalle en la Unidad III.

Como habrá notado en los dos ejemplos, otra característica de las matrices respecto a los arreglos es que sus índices inician en 1 y no en 0. Por lo tanto, habrá que hacer la respectiva conversión al momento de su declaración y/o uso.

2. Implementación

El programa 2.1 muestra cómo implementar una matriz aprovechando las ventajas de la programación orientada a objetos.

```
#include <iostream>
#include <stdexcept>

using namespace std;

class matriz
{
    private:
        int NroFilas;          // Nro de filas de la matriz
        int NroColumnas; // Nro de columnas de la matriz
        int **elementos; // Puntero hacia un arreglo bidimensional
                    // (del tipo "Arreglo de arreglos")
                    // para almacenar los elementos de la matriz

    public:
        // Constructor
        matriz(int pNroFilas, int pNroColumnas)
        {
            // Validar la cantidad de filas y columnas indicada por el usuario
            if (pNroFilas <= 0 || pNroColumnas <= 0)
                throw invalid_argument("La cantidad de filas y columnas deben ser may-
or que cero.");

            // Crear la matriz
            this->NroColumnas = pNroColumnas;
            this->NroFilas = pNroFilas;
            this->elementos = new int *[NroFilas];
            for (int i=0; i<NroFilas; i++)
                elementos[i] = new int [NroColumnas];
        }

        // Destructor
        ~matriz() {delete [] elementos;}

        // Otras operaciones
        void set(int pFil, int pCol, int pElemento) // Insertar un elemento
        {
            // Validar que el índice no esté fuera de rango
            if (pFil <= 0 || pFil > NroFilas + 1 || pCol <= 0 || pCol > NroColumnas
+ 1)
                throw invalid_argument("El número de fila o columna especificado está
fuera de rango");
```

```
            elementos[pFil-1][pCol-1] = pElemento;
        }

        int get(int pFil, int pCol) // Recuperar un elemento
        {
            // Validar que el índice no esté fuera de rango
            if (pFil <= 0 || pFil > NroFilas + 1 || pCol <= 0 || pCol > NroColumnas
+ 1)
                throw invalid_argument("El número de fila o columna especificado está
fuera de rango");
            return elementos[pFil-1][pCol-1];
        }
};

// PROGRAMA PRINCIPAL
void main()
{
    int elemento;
    int nfilas, ncolumnas;
    cout<<endl;
    cout<<"Creando la matriz..."<<endl;
    cout<<"---------------------"<<endl;
    cout<<"Indique número de filas de la matriz; ";cin>>nfilas;
    cout<<"Indique número de columnas de la matriz; ";cin>>ncolumnas;

    try
    {
        matriz m(nfilas,ncolumnas);
        cout<<endl;
        cout<<"Matriz creada..."<<endl;
        cout<<endl;

        cout<<"Digite los valores para la matriz"<<endl;
        cout<<"--------------------------------"<<endl;
        cout<<endl;

        for (int i=1; i<=nfilas; i++)
            for (int j=1; j<=ncolumnas; j++)
            {
                cout<<"Elemento ["<<i<<"]["<<j<<"]: ";cin>>elemento;
                m.set(i, j, elemento);
            }

        cout<<endl;
        cout<<"Matriz"<<endl;
        cout<<"------"<<endl;

        for (int i=1; i<=nfilas; i++)
        {
            for (int j=1; j<=ncolumnas; j++)
                cout<<m.get(i, j)<<"  ";
            cout<<endl;
        }
```

```
    }
    catch (invalid_argument& e)
    {
        cerr<<e.what()<<endl;
    }
    system("Pause");
}
```

Programa 2.1

A continuación, se describen algunas características del código.

- Se implementó empleando una clase. La ventaja que ofrece es que se podrán declarar en tiempo de ejecución tantas matrices como se necesiten y hacer operaciones con ellas y entre ellas.
- El constructor permite inicializar las variables que almacenan la cantidad de filas y columnas, a la vez que crea la matriz basada en punteros. ¿Por qué se opta por este método y no se declara la matriz directamente? Porque la clase permite crear varias matrices y de diferentes tamaños; si se crea directamente (sin punteros), obligaría a definirlas de un tamaño fijo.

`int nrofilas = 4;`
`int nrocols = 5;`
 	
`int matriz[nrofilas][nrocols];`
	Ejemplo de creación de una matriz de 4 filas y 5 columnas con tamaño predefinido e invariante en el tiempo.

`void crearmatriz(int nrofilas, int nrocols)`
`{`
`  int **matriz`
`  matriz = new int *[nrofilas]`
`  for (int i=0; i<5; i++)`
`  matriz[i] = new int [nrocols];`
`}`
	Ejemplo de procedimiento que permite crear en tiempo de ejecución matrices de diferentes tamaños; dependerá de los valores que se especifiquen para número de filas (nrofilas) y número de columnas (nrocols).

Otra alternativa es declarar un puntero simple (y no doble) a un arreglo unidimesional y simular la matriz mediante las funciones de mapeo, las cuales se explicaron en el numeral 4 de la unidad anterior.

- El destructor permite liberar el arreglo de la memoria.

3. Operaciones con matrices

"Las operaciones más comunes con matrices son transponer, suma y multiplicación" (Sahni, 2005, p. 232).

3.1. Transpuesta de una matriz

La transpuesta de una matriz A *(m x n)* es otra matriz T *(n x m)*. Matemáticamente, se expresa de la siguiente manera:

$$M^T(i,j) = M(j,i), 1 \leq i \leq n, 1 \leq j \leq m$$

Gráficamente, es como se muestra en la figura que se muestra a continuación:

$$Si \quad A = \begin{bmatrix} 1 & 3 \\ 0 & -5 \\ 2 & 7 \end{bmatrix} \quad entonces\ A^T = \begin{bmatrix} 1 & 0 & -2 \\ 3 & -5 & 7 \end{bmatrix}$$

$$Si \quad B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad entonces\ B^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$Si \quad C = \begin{bmatrix} 9 & 0 & 4 \\ 10 & 3 & -2 \end{bmatrix} \quad entonces\ C^T = \begin{bmatrix} 9 & 10 \\ 0 & 3 \\ 4 & -2 \end{bmatrix}$$

Figura 27. Ejemplos de transpuesta de una matriz

El programa 2.2 muestra un ejemplo sobre cómo realizar esta operación. Tome en cuenta que se emplea la clase del programa 2.1.

```
void imprimir(matriz *m)
{
   cout<<endl;
   cout<<"Valores de la matriz"<<endl;
   cout<<"--------------------"<<endl;
   cout<<endl;
   for (int i=1; i<=m->getNroFilas(); i++)
   {
         for(int j=1; j<=m->getNroColumnas(); j++)
               cout<<m->get(i,j)<<"   ";
         cout<<endl;
   }
   cout<<endl;
}

void transponer(matriz & m, matriz & resultado)
{
   int nroFilas = m.getNroFilas();
   int nroColumnas = m.getNroColumnas();

   for (int i=1; i<=nroFilas; i++)
         for (int j=1; j<=nroColumnas; j++)
               resultado.set(j, i, m.get(i, j));
}

void main()
{
   // Variables necesarias
   int elemento;

   // Crear matriz original e insertar los datos
   int nfilas = 2;
```

```
      int ncolumnas = 3;
      matriz m(nfilas, ncolumnas);

      cout<<"Digite los valores para la matriz"<<endl;
      cout<<"--------------------------------"<<endl;
      cout<<endl;

      for (int i=1; i<=nfilas; i++)
            for (int j=1; j<=ncolumnas; j++)
            {
                  cout<<"Elemento ["<<i<<"]["<<j<<"]: ";cin>>elemento;
                  m.set(i, j, elemento);
            }

      // Imprimir matriz original
      imprimir(&m);

      // Crear matriz transpuesta
      matriz T(ncolumnas, nfilas);
      transponer(m, T);

      // Imprimir matriz resultado
      imprimir(&T);

      system("Pause");
   }
```

Programa 2.2

3.2. Suma de matrices

Esta operación es más sencilla, ya que consiste únicamente en sumar cada uno de los elementos de dos matrices de iguales dimensiones; es decir, el mismo número de filas y columnas. Matemáticamente, se define como se observa a continuación:

$$S(i,j) = A(i,j) + B(i,j), 1 \leq i \leq n, 1 \leq j \leq m$$

Gráficamente, es como se aprecia en la figura.

$$\begin{bmatrix} 1 & 3 \\ 0 & -5 \\ 2 & 7 \end{bmatrix} + \begin{bmatrix} 2 & 4 \\ 3 & -1 \\ 1 & 6 \end{bmatrix} = \begin{bmatrix} 3 & 7 \\ 3 & -6 \\ 3 & 13 \end{bmatrix}$$

Figura 28. Ejemplo de suma de dos matrices

El siguiente programa muestra un procedimiento para realizar esta operación; ese procedimiento cual puede acoplarse a alguno de los programas anteriores para probar su funcionamiento.

```
void sumar(matriz & m1, matriz & m2, matriz & resultado)
{
      int nroFilas = m1.getNroFilas();
      int nroColumnas = m1.getNroColumnas();

      for (int i=1; i<=nroFilas; i++)
```

```
            for (int j=1; j<=nroColumnas; j++)
                  resultado.set(i, j, m1.get(i, j) + m2.get(i, j));
}
```

Programa 2.3

3.3. Producto de matrices

El producto de dos matrices A (*m* x *n*) y B (*p* x *q*) es posible si y solo si el número de columnas de A es igual número de filas de B; es decir, *n* = *p*. Cuando esa condición se da, la operación está definida como se observa a continuación:

$$P(i,j) = \sum_{k=1}^{n} A(i,k) * B(k,j), 1 \leq i \leq n, 1 \leq j \leq q$$

Gráficamente, es como se muestra en la siguiente figura:

Figura 29. Productos de matrices

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \\ ex + fy \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \end{bmatrix}$$

Figura 29. Productos de matrices

```
void producto(matriz & m1, matriz & m2, matriz & resultado)
{
   int m = m1.getNroFilas();
   int n = m1.getNroColumnas();

   int p = m2.getNroFilas();
   int q = m2.getNroColumnas();


   // Verificar que la condición establecida se cumpla (n=p)
   if (n == p)
      for (int i=1; i<=m; i++)
         for (int j=1; j<=p; j++)
         {
            int sum = 0;

            for (int k=1; k<=p; k++)
               sum = sum + m1.get(i, k) * m2.get(k, j);
            resultado.set(i, j, sum);
         }
}
```

Programa 2.4

4. Matrices especiales

Sahni (2005, p. 239) diferencia entre cuatro tipos de matrices:

a) Matriz diagonal

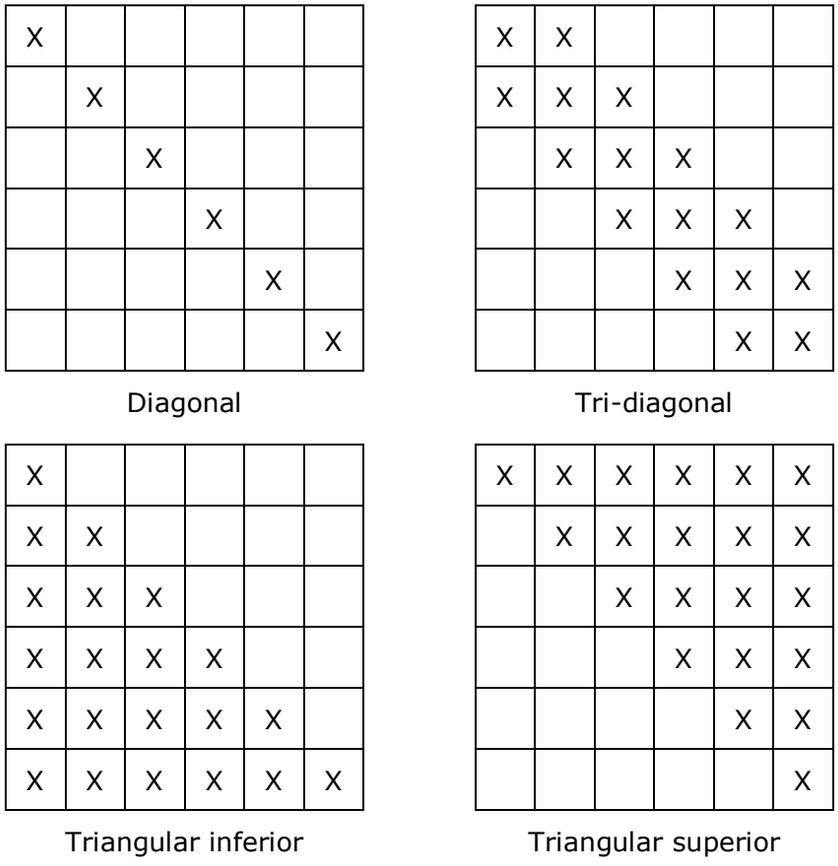
b) Matriz tri-diagonal

c) Matriz triangular inferior

d) Matriz triangular superior

Las siguientes figuras muestran sus características; el lugar marcado con "X" representa algún valor diferente de 0 (cero).

Figura 30. Tipos de matrices



Lectura seleccionada n.º 02: Gestión dinámica de memoria

La siguiente lectura describe cómo operan los punteros, su relación con la memoria del computador y su importancia en la implementación de aplicaciones tanto con variables simples como con arreglos. Aunque las indicaciones que expone la lectura son de uso general, se deben tomar en consideración al crear arreglos de punteros (numeral 2) y matrices bidimensionales basadas en punteros (numeral 5).

Leer hasta el apartado 8.6: Vectores dinámicos.

Barber, F., & Ferrís, R. Tema 8: *Gestión dinámica de memoria*. Disponible en http://informatica.uv.es/iiguia/AED/oldwww/2004_05/AED.Tema.08.pdf

Actividad N.° 02

Foro de discusión sobre implementación de operaciones con matrices

Instrucciones:

a) Ingrese al foro y participe con comentarios sobre la siguiente pregunta: ¿cuál de los programas mostrados a continuación reduce el código de implementación de las operaciones con matrices: ¿transponer, sumar y multiplicar? Argumente su respuesta en tres líneas como máximo.

Nota: Se recomienda implementar los programas y hacer las pruebas respectivas antes de participar.

PROGRAMA 1
ARREGLO BIDIMENSIONAL BASADO EN PUNTERO (ARREGLO DE ARREGLOS)

```
// Declarar el arreglo contenedor
int **arrayBidimensional = new int
*[NroFilas];

// Asignar un subarreglo a cada
casilla del arreglo contenedor
for (int i=0; i<NroFilas; i++)
  arrayBidimensional[i] = new int
[NroColumnas];

// Método para insertar un valor en
el arreglo
void set(int iFil, int iCol, int val-
or)
{
  arrayBidimensional[iFil][iCol] =
valor;
}

// Función que recupera un valor del
arreglo
int get(int iFil, int iCol)
{
return arreglo[iFil][iCol];
}
```

PROGRAMA 2
ARREGLO UNIDIMENSIONAL QUE SIMULA SER UNO BIDIMENSIONAL (MAPEO)

```
// Número de filas
const int NroFilas = 4;
// Número de columnas
const int NroColumnas = 3;
// Declaración del arreglo
int arreglo[NroFilas * NroColumnas];

// Método para insertar un valor en
el arreglo
void set(int iFil, int iCol, int val-
or)
{
  // Conversión de índice mediante la
función de mapeo
  int indice = iFil * NroColumnas +
iCol;
  // Asignación del valor
  arreglo[índice] = valor;
}

// Función que recupera un valor del
arreglo
int get(int iFil, int iCol)
{
  // Conversión de índice mediante la
función de mapeo
  int indice = iFil * NroColumnas +
iCol;
  // Retornar el valor del arreglo
según el índice calculado
  return arreglo[índice];
}
```

GLOSARIO DE LA UNIDAD II

A

Arreglo. Conjunto de espacios de memoria que operan bajo un mismo nombre.

D

Dimensión (de arreglo). Cantidad numérica de cada una de las magnitudes que conforman un arreglo. La multiplicación de estas cantidades permite determinar el total de espacios de memoria que tiene reservados.

E

Escribir (datos). Asignar un valor a una variable.

I

Índice (de arreglo). Número entero que representa cada una de las casillas de un arreglo.

L

Leer (datos). Recuperar el valor de una variable.

M

Magnitud. Propiedad física que puede ser medida.

P

Programa (en informática). Conjunto de instrucciones que realizan una tarea específica.

T

Tipo de dato. Cada uno de los valores que puede almacenar un computador. Se distinguen entre textos, números, lógicos, fecha y hora, etc.

V

Variable. En un lenguaje programación es un espacio de memoria, identificado por un nombre y reservado para almacenar un dato específico a la vez.

Bibliografía de la Unidad II

Cairo, O. & Guardati, S. (2010). *Estructuras de datos* (3.ª ed.). México: Editorial McGraw Hill.

Charles, S. (2009). *Python para informáticos: Explorando la información* [en línea]. Disponible en http://do1.dr-chuck.net/py4inf/ES-es/book.pdf

Cruz Huerta, D. (2011). *Apuntes de la asignatura: Estructura de datos*. México: Tecnológico de Estudios Superiores de Ecatepec. Disponible en http://myslide.es/documents/manual-estructura-de-datos.html

Pérez, J. (2004). *Matemáticas*. España: Instituto Nacional de Tecnologías Educativas y de Formación del Profesorado. Disponible en http://sauce.pntic.mec.es/~jpeo0002/Archivos/PDF/T06.pdf

Sahni, S. (2005). *Data structures, algorithms and applications in C++* (2.ª ed.) Hyderabad, India: Universities Press. Disponible en https://sandeepdasari.files.wordpress.com/2013/01/ds.pdf

Universidad de Valencia, Instituto Universitario de Investigación de Robótica y Tecnologías de la Información y Comunicación. (s.f.). *Estructura de datos*. IRTIC. Disponible en http://robotica.uv.es/pub/Libro/PDFs/CAPI5.pdf.

Autoevaluación de la Unidad II

1. ¿Qué es un arreglo?

   a. Una variable simple con un nombre determinado

   b. Una variable compuesta que opera bajo diferentes nombres

   c. Conjunto de variables que operan bajo diferentes nombres

   d. Conjunto de variables que operan bajo un mismo nombre

2. ¿Cuál de las siguientes representa mejor la sintaxis para declarar en C++ un arreglo de tipo entero de n dimensiones?

   a. int arreglo[n]

   b. arreglo[int n]

   c. int arreglo[x][y][z]…

   d. int arreglo[int n]

3. El método set de un arreglo debe recibir como parámetro…

   a. El valor a insertar en el arreglo

   b. El valor a insertar en el arreglo y el índice

c. El nombre del arreglo y el número de índice

d. El nombre del índice y el valor a insertar

4. El método get de un arreglo debe recibir como parámetro…

a. El número del índice del valor a recuperar

b. El valor del índice y el valor a recuperar

c. El número a recuperar

d. El número de índice y el nombre del arreglo

5. ¿Cuáles son las funciones de mapeo por filas y por columnas?

a. $map(i_f, i_c) = i_f C + i_c \qquad map(i_f, i_c) = i_c F + i_f$

b. $map(i_c, i_f) = i_f C + i_c \qquad map(i_c, i_f) = i_c F + i_f$

c. $map(i_f, i_c) = i_c F + i_f \qquad map(i_f, i_c) = i_c F + i_c$

d. $map(i_c, i_f) = i_f F + i_c \qquad map(i_f, i_c) = i_c C + i_f$

6. ¿Cuál es la forma de declarar un puntero que soporte crear arreglos bidimensionales?

a. int **arrayBidimensional = new int *[NroFilas]

b. **arrayBidimensional = int *[NroFilas]

c. int **arrayBidimensional = *int [NroFilas]

d. **arrayBidimensional = new int *[NroFilas]

7. ¿Cuál es la característica de una matriz respecto a un arreglo bidimensional?

a. Sus índices inician en 1

b. Son de tamaño fijo

c. Solo pueden ser rectangulares

d. No permite guardar valores enteros

8. Mencione los dos tipos de matrices propuesta por Sahni

a. Matriz numérica y matriz de caracteres

b. Matriz de información y matriz de relación

c. Matriz de datos y matriz de información

d. Matriz rectangular y matriz cuadrada

9. ¿Por qué es preferible emplear una clase para la creación de matrices?

   a. Para que desde el programa principal se puedan hacer operaciones con una sola matriz.

   b. Para que desde el programa principal se puedan crear matrices de tamaño fijo.

   c. Para que desde el programa principal se puedan eliminar las matrices que dejen de usarse.

   d. Para que desde el programa principal se pueda crear todas las matrices que se requieran.

10. Identifique dos tipos de matrices especiales:

   a. Matriz cuadrada, matriz rectangular

   b. Matriz binaria, matriz numérica

   c. Matriz diagonal, matriz inferior

   d. Matriz tri-diagonal, matriz mediana

UNIDAD III

ESTRUCTURAS LINEALES Y NO LINEALES DE DATOS

DIAGRAMA DE ORGANIZACIÓN

ORGANIZACIÓN DE LOS APRENDIZAJES

Resultado de aprendizaje de la Unidad III:

Al finalizar la unidad, el estudiante será capaz de seleccionar las estructuras lineales y no lineales en la solución de diversos problemas.

CONOCIMIENTOS	HABILIDADES	ACTITUDES
Tema n.º 1: Pilas, colas y listas 1. Pilas 1.1 Definición 1.2 Representación abstracta Tema n.º 2: Grafos **Tema n.º 3: Árboles** Lectura seleccionada n.º 3: La magia de los grafos **Autoevaluación de la Unidad III**	• Aplica y relaciona propiedades de las pilas, colas y listas. • Analiza las propiedades de los grafos y los aplica en las diversas situaciones de la vida real. Actividad N.º 1: Foro de discusión • Aplica y relaciona propiedades de los árboles. Actividad N.º 2 Foro de discusión • Resuelve las prácticas de laboratorio.	• Demuestra perseverancia y esfuerzo durante el desarrollo de los ejercicios. • Toma conciencia de la importancia de la asignatura en su formación profesional. • Valora las relaciones entre sus compañeros.

TEMA N.º 1: PILAS, COLAS Y LISTAS

Las pilas, colas y listas constituyen estructuras de datos especiales que basan su funcionamiento en arreglos o punteros. Las pilas permiten la inserción y eliminación de elementos solo por uno de sus extremos y encuentran su aplicación práctica en llamadas a subprogramas, recursividad u ordenamiento, entre otros. El problema planteado por las denominadas "Torres de Hanoi", por ejemplo, usa el principio de las pilas para su resolución. Por su parte, las colas son estructuras cuyos elementos se insertan por uno de sus extremos y salen por el otro; su modo de operación se parece a la cola de espera que se genera frente a una ventanilla de atención. Finalmente, las listas son estructuras que hacen uso más eficiente de la memoria, empleando solo la que necesitan en función de la cantidad de elementos que poseen. Si bien en este manual las pilas y colas se implementaron sobre arreglos por fines didácticos, la práctica indica que pueden ser creadas también sobre listas basadas en punteros.

1. Pilas

1.1 Definición

"Una pila representa una estructura lineal de datos en la que se puede agregar o quitar elementos únicamente por uno de sus lados. En consecuencia, los elementos de una pila se eliminarán en orden inverso al que fueron insertados" (Cairo, & Guardati, 2010, p. 75). Su representación gráfica se observa en la figura 31.

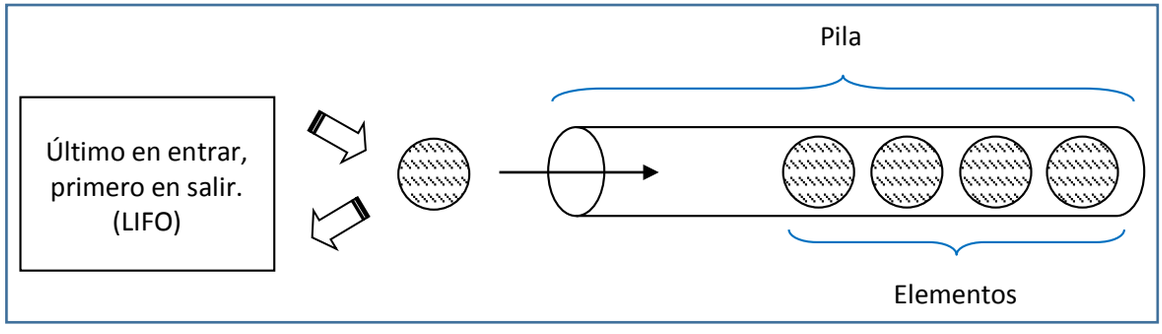


Figura 31. Representación gráfica de una pila
Elaboración propia.

Otra forma de ayudar a la mente a entender el funcionamiento de esta estructura es homologándola a una pila de platos en un restaurante, en la que el encargado de lavarlos los va acomodando uno encima de otro, mientras que quien sirve los platillos va retirándolos desde la parte superior; es decir, el último plato colocado por el lavandero será el primero en ser utilizado o, en términos técnicos, "el último en entrar será el primero en salir" (en inglés *Last-Input*, *First-Out*, *LIFO*).

Sahni (2005) expresa que las pilas y las colas (estructuras que se verán más adelante) son versiones particulares de listas lineales. Es decir, están implementadas a partir de listas basadas en arreglos o en punteros.

1.2 Representación abstracta

```
{
        Instancias
          Lista lineal de elementos
        Operaciones
          empty()      :  retorna Verdadero si la pila está vacía
          size()       :  retorna el número de elementos de la pila
          top()        :  retorna el primer elemento de la pila
          erase()      :  elimina el primer elemento de la pila
          insert(x)    :  inserta el elemento "x" a la pila
}
```

Figura 32. Representación abstracta de una pila
Tomada de *Data Structures, Algorithms, and Applications in C++, por* Sahni, (2005, p. 247).

1.3 Ejemplo de implementación

A continuación, se muestra el ejemplo de implementación de tales operaciones que emplean como soporte para la pila un arreglo simple.

```
const int tamaño_pila = 5;

class pila
{
  private:
    int cima;
    int arreglo_pila[tamaño_pila];
  public:
    // Constructor
    pila()
    {
      // Inicializar el indicador
      cima = -1;
      // Inicializar el arreglo
      for (int i=0; i<tamaño_pila; i++)
        arreglo_pila[i] = NULL;
    }
    bool empty()
    {
      if (cima==-1)
        return true;
      else
        return false;
    }

    int size()
    {
      return cima;
    }

    int top()
    {
      if (cima>-1)
        return arreglo_pila[cima];
    }
```

```
    void erase()
    {
       if (cima>-1)
          cima = cima - 1;
    }

    void insert(int elemento)
    {
       arreglo_pila[cima+1] = elemento;
       cima = cima + 1;
    }

    void imprimir()
    {
       cout<<endl;
       cout<<"Elementos de la pila"<<endl;
       cout<<"--------------------"<<endl;
       cout<<endl;
       for (int i=0; i<=cima; i++)
          cout<<"Elemento["<<i+1<<"]: "<<arreglo_pila[i]<<endl;
    }
};
```

Programa 1.1

En este ejemplo, los elementos ingresan y salen de la pila por el índice de mayor valor. La variable "cima" almacena la posición del último elemento insertado, mientras que cuando la pila está vacía su valor es -1.

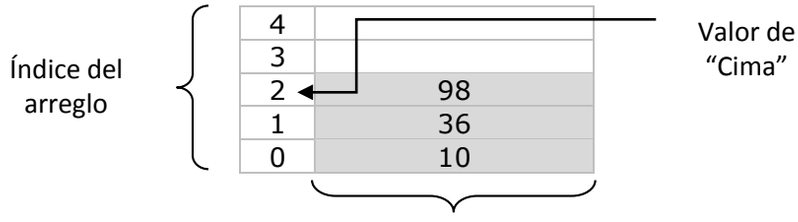


Figura 33. Representación de la pila del programa 1.1
Fuente: Elaboración propia

Ahora, implemente esta clase y cree instancias para probar su funcionamiento.

2. Colas

2.1 Definición

Una cola es otro tipo especial de lista en el cual los elementos se insertan en un extremo (el posterior) y se suprimen en el otro (el anterior o frente). Las colas se conocen también como listas "FIFO" (*first-in first-out*) o listas "primero en entrar, primero en salir". Las operaciones para una cola son análogas a las de las pilas, aunque las diferencias sustanciales consisten en que las inserciones se realizan al final de la lista y no al principio (Aho, 1983).

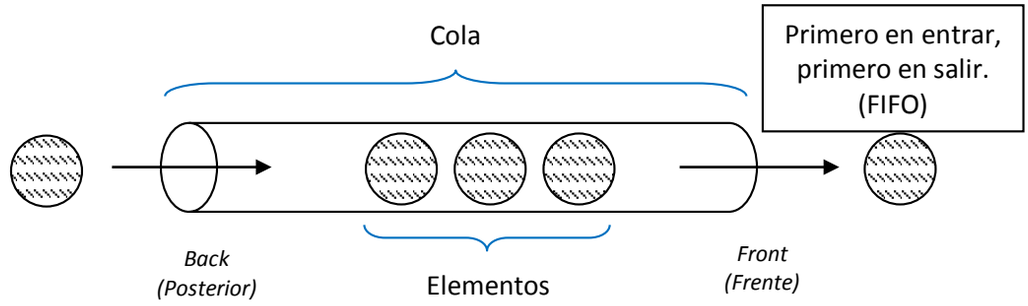


Figura 34. Representación gráfica de una cola
Elaboración propia

Su comportamiento se homologa al conjunto de personas que aguardan para ser atendidos en una ventanilla.

2.2 Representación abstracta

Cola
{
Instancias
Lista lineal de elementos
Operaciones

empty()	:	retorna Verdadero si la cola está vacía.
size()	:	retorna el número de elementos de la cola
front()	:	retorna el elemento de la parte frontal
back()	:	retorna el elemento de la parte posterior
erase()	:	elimina el primer elemento de la parte frontal
insert(x)	:	inserta el elemento "x" a la cola (parte posterior)

}

Figura 35. Representación abstracta de una cola
Tomado de *Data Structures, Algorithms, and Applications in C++,* por Sahni, 2005, p. 321.

2.3 Ejemplo de implementación

A continuación, se muestra el ejemplo de implementación de tales operaciones, las cuales utiliza un arreglo simple como soporte para la cola.

```
class cola
{
    private:
      int frente, posterior;
      int arreglo_cola[tamaño_cola];

    public:
      cola()
      {
        // Inicializar los indicadores
        frente = -1;
        posterior = -1;
        // Inicializar el arreglo
        for (int i=0; i<tamaño_cola; i++)
          arreglo_cola[i] = NULL;
      }
```

```
bool empty()
{
   if (posterior==-1)
     return true;
   else
     return false;
}

int size()
{
   if (posterior!=-1)
     return posterior - frente + 1;
}

int front()
{
   if (posterior!=-1)
     return arreglo_cola[frente];
}

int back()
{
   if (posterior!=-1)
     return arreglo_cola[posterior];
}

void erase()
{
   if (frente < posterior)
     frente++;
   else if (frente == posterior)
   {
     frente = -1;
     posterior = -1;
   }
}

void insert(int elemento)
{
   if (posterior==-1)
   {
     arreglo_cola[0] = elemento;
     posterior++;
     frente++;
   }

   else if (posterior < tamaño_cola)
   {
     arreglo_cola[posterior+1] = elemento;
     posterior++;
   }

   if (posterior == tamaño_cola)
   {
     if (frente > 1)
     {
       for (int i=frente; i<tamaño_cola; i++)
```

```
            {
              arreglo_cola[i-frente] = arreglo_cola[i];
              arreglo_cola[i] = NULL;
            }
            posterior = tamaño_cola - frente - 1;
            frente = 0;

            arreglo_cola[posterior+1] = elemento;
            posterior++;
          }
        }
      }

      void imprimir()
      {
        cout<<endl;
        cout<<"Elementos de la cola"<<endl;
        cout<<"--------------------"<<endl;
        cout<<endl;
        for (int i=frente; i<=posterior; i++)
          cout<<"Elemento["<<i+1<<"]: "<<arreglo_cola[i]<<endl;
      }
};
```

Programa 1.2

El programa crea una cola sobre un arreglo unidimensional. Se controla el inicio y final de la misma con las variables "frente" y "posterior", siendo los valores del primero menores que los del segundo.

Si al insertar un nuevo elemento, la cola está copada hasta la última posición y tiene espacios libres en la parte frontal (como se muestra en el paso "a" de la figura 36), entonces todos los elementos deben "correr" hacia la parte izquierda para recién insertar el nuevo ítem. Observe la figura que se muestra a continuación:

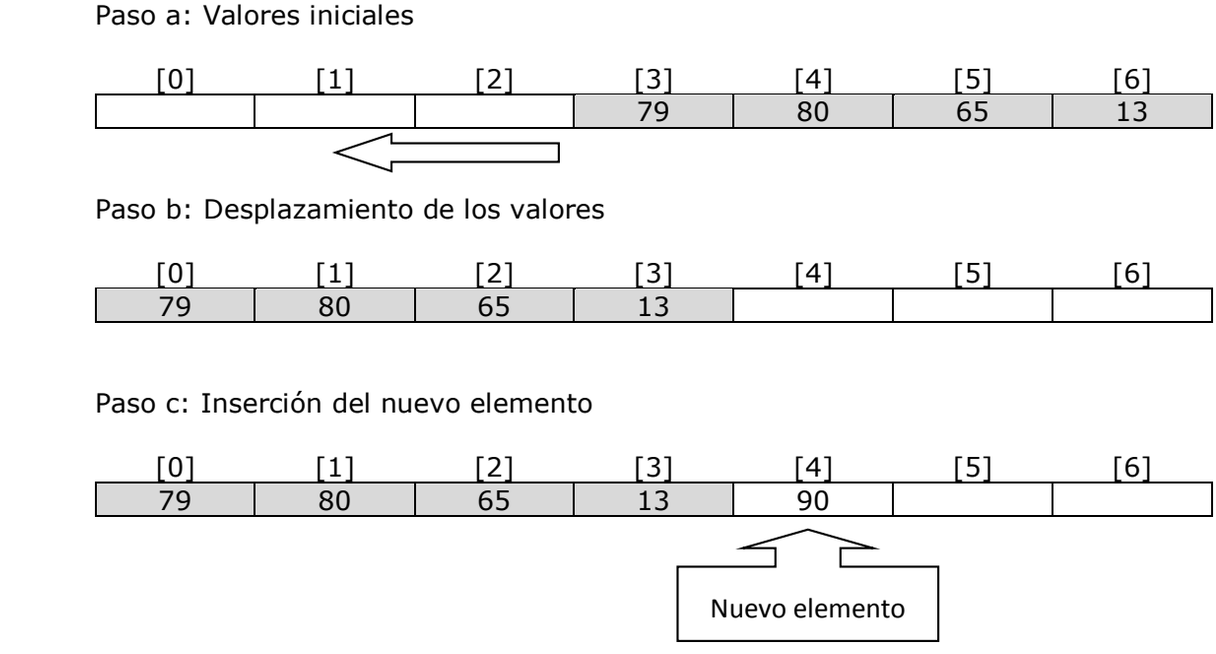


Figura 36. Proceso de inserción de elemento en una cola con valores hacia la derecha.
Fuente: Elaboración propia.

3. Listas

3.1 Definición

"Una lista es un tipo de estructura lineal y dinámica de datos. Es lineal porque a cada elemento le puede seguir solo otro elemento; dinámica, porque se puede manejar la memoria de manera más flexible, sin necesidad de reservar espacio con antelación" (Cairo & Guardati, 2010, p. 141).

Esto autores destacan que la ventaja de emplear listas en vez de arreglos radica en que se pueden adquirir espacios de memoria a medida que se necesiten; estas se liberan cuando ya no se requieren.

Como se reviso en la primera unidad (en el apartado "Listas basadas en punteros"), una lista ligada es una colección de nodos que almacenan dos tipos de datos: el valor en sí y la dirección de memoria al siguiente nodo.

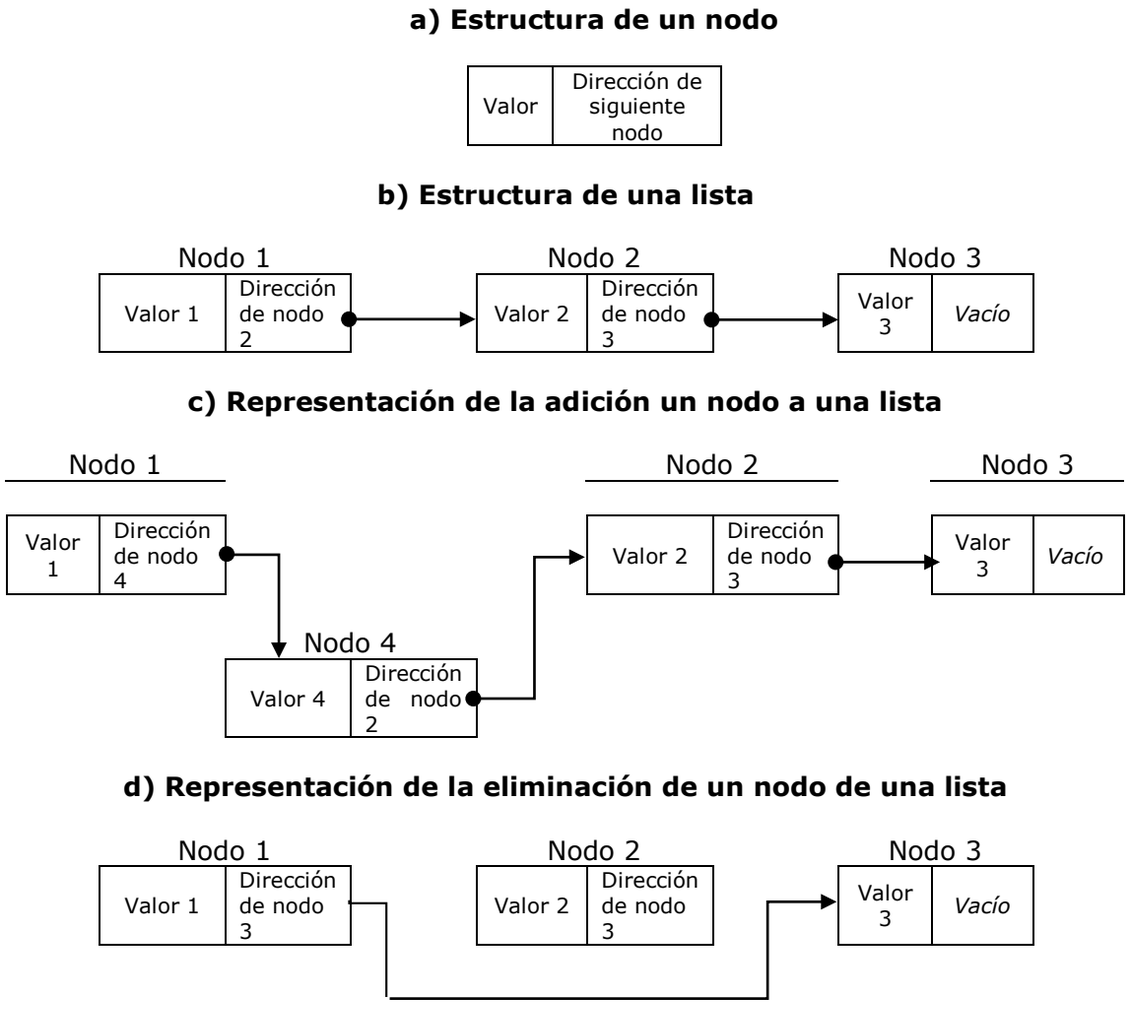


Figura 37. Estructura de una lista enlazada y las operaciones con ella
Fuente: Elaboración propia

3.3 Implementación

A continuación, se describen los pasos para crear una lista en C++. Cabe precisar que el código mostrado fue expuesto en el programa 1.4 de la Unidad I.

¿Qué necesitamos? Además de comprender cómo operan los punteros, es imprescindible tener pleno conocimiento de las denominadas estructuras y la definición de los tipos de datos personalizados.

a) Estructura

Joyanes y Zahonero (2003, p. 296) definen a "una estructura como una colección de uno o más tipos de elementos denominados miembros, cada uno de los cuales puede ser de un tipo de dato diferente". Por ejemplo, a diferencia de un *array*, las estructuras son útiles, por ejemplo, para guardar diversos tipos de datos de clientes como edad, nombre, número telefónico, etc., sin perder la cualidad de acceder a cualquiera de estos valores bajo un mismo nombre de variable.

El siguiente programa crea una estructura llamada "Persona" para el registro de cuatro datos.

```
struct persona
{
string nombre;
string apellido;
int edad;
bool vivo;
};
```

Creación de la estructura

```
void main()
{
struct persona p1, p2;
p1.nombre = "Juan";
p1.apellido = "Robles";
p1.edad = 23;
p1.vivo = true;
p2.nombre = "María";
p2.vivo = false;
}
```

Declaración de elementos (p1 y p2) del tipo de la estructura y asignación de valores a sus propiedades.

Programa 1.3

b) Definición de tipos de datos personalizados

Se sabe que la sintaxis para la declaración de una variable entera sin signo (solo valores positivos) es la siguiente:

```
unsigned int nombre_variable;
```

Ahora bien, considere que necesita crear muchas de estas variables (enteras sin signo) para un mismo programa. Entonces, para cada declaración, usted tendría que anteponer al nombre de la variable las palabras "*unsigned int*". Así como se aprecia a continuación:

```
unsigned int variable1;

unsigned int variable2;
```

Pues bien, C++ permite simplificar los nombres de los tipos de datos y asignarles un alias para lo cual se emplea la palabra reserva *"typedef"* como se muestra a continuación:

```
typedef unsigned int u_int
```

Tipo de dato | Alias

De modo que la declaración de variables enteras sin signo se simplifica. Por ejemplo: u_int variable1, variable2

Esta asignación de alias funciona para cualquier nombre de tipo de datos (*int, string, bool*, etc.), sin ser la excepción las estructuras. Como se puede apreciar en el programa 1.3, la declaración de elementos del tipo "estructura persona" amerita que se antepongan las palabras *"struct persona"*. Para simplificar esta sentencia se puede optar por lo siguiente:

```
// Asignación de alias
typedef struct persona TPersona;
(…)
// Declaración de variables de tipo estructura persona
TPersona p1, p2;
```

Teniendo en consideración ambas aclaraciones; a continuación, se enumeran los pasos para crear una lista.

a) Creación de una estructura para representar a los nodos

```
struct nodo
{
int nro;                // dato
struct nodo *sgte;      // puntero a nodo siguiente
};
```

La primera parte del nodo es para el dato en sí (un entero en este caso), mientras que la segunda es un puntero en la cual se almacenará la dirección de memoria del siguiente nodo. Para este último se usa el principio de recursividad; es decir, la estructura nodo apunta a otra estructura nodo (a sí misma).

b) Creación de un alias para la estructura

```
typedef struct nodo *TLista;
```

Este alias se crea por tres razones: primero, para simplificar la declaración de elementos; segundo, para tener un nombre de tipo de dato más cercano a lo que realmente se creará (una lista); y tercero, para especificar que lo que se necesita es un puntero (note el asterisco que precede a la palabra TLista), pues de no crearlo como tal obligaría a que cada elemento de tipo TLista tenga ya un nodo desde su declaración; pero esto no necesariamente es así, porque sabemos que una lista puede estar vacía.

c) Creación de la lista

```
void main()
{
      TLista lista1 = NULL;
}
```

En este ejemplo se crea la lista llamada "lista1". Como se explicaba en el literal b (motivo tres), se le asigna el valor *NULL* a "lista1", ya que aún no tiene ningún elemento (nodo). Por tanto, es un puntero vacío.

3.4 Operaciones con una lista

Las operaciones posibles son tres: inserción, borrado y búsqueda.

3.4.1 Inserción

La inserción de elementos en una lista puede darse tanto al inicio como al final de la misma. Los programas 1.4 y 1.5 muestran las subrutinas para ambas tareas.

```
void insertarInicio(Tlista &lis-
ta, int valor)
{
    Tlista q;
    q = new(struct nodo);
q->nro = valor;
    q->sgte = lista;
    lista = q;
}
```

Programa 1.4

```
void insertarFinal(Tlista &lista, int
valor)
{
    Tlista t, q = new(struct nodo);

q->nro  = valor;
    q->sgte = NULL;

    // Verificar si la lista está vacía
    if(lista==NULL)
    {
        lista = q;
}
Else
    {
        t = lista;
        while(t->sgte!=NULL)
        {
            t = t->sgte;
        }
        t->sgte = q;
    }
}
```

Programa 1.5

En el programa de la izquierda se crea una variable auxiliar "q", la cual recibe el dato en su campo "nro" y la dirección de la lista existente en el campo "sgte" de tal forma que se ubica al inicio. Por su parte, el programa de la derecha emplea la misma variable auxiliar; sin embargo, primero verifica si la lista está vacía y, si es el caso, inserta el nuevo nodo sin ninguna otra acción; mientras que si ya existen elementos, recorre toda lista hasta llegar al último, empleando para ello una variable auxiliar "t", y se asigna a "t" (último nodo) la dirección del nuevo elemento "q".

3.4.2 Borrado

El programa 1.6 muestra el código para eliminar un nodo según el valor que contiene. La representación gráfica de esta acción se encuentra en el literal "*d*" de la figura 38.

```
void eliminarElemento(Tlista &lista, int valor)
{
   Tlista p, ant;
   p = lista;

   if(lista!=NULL)
   {
      while(p!=NULL)
      {
         if(p->nro==valor)
         {
            if(p==lista)
               lista = lista->sgte;
            else
```

```
                ant->sgte = p->sgte;
                delete(p);
                return;
            }
            ant = p;
            p = p->sgte;
        }
    }
}
```

Programa 1.6

3.4.3 Búsqueda

El programa 1.7 muestra el código para buscar un determinado dato en la lista.

```
void buscarElemento(Tlista lista, int valor)
{
   Tlista q = lista;
   int i = 1;
   bool encontrado = false;

   while(q!=NULL)
   {
      if(q->nro==valor)
      {
         cout<<endl<<" Encontrada en posición "<< i <<endl;
         encontrado=true;
      }
      q = q->sgte;
      i++;
   }
   if(encontrado==true)
      cout<<"\n\n Número no encontrado..!"<<endl;
}
```

Programa 1.7

TEMA N.º 2: GRAFOS

Los grafos constituyen la primera estructura no lineal que se estudiará en este curso; su no linealidad se da porque cada uno de sus elementos puede estar precedido o sucedido por otros (no necesariamente uno como ocurre en las listas). Los grafos no son estructuras de datos predefinidas de algún lenguaje de programación, sino que son superestructuras que basan su funcionamiento en elementos ya descritos como son los *arrays* y los punteros. Luego de definirlos se describirán algunos algoritmos que están orientados a la solución de problemas reales más que a su implementación propiamente.

1. Definición

Mehta y Sahni (2005, p. 4-1) concuerdan en que un grafo *G= (V,E)* consiste en un conjunto finito de vértices (o nodos) $V=\{v_1, v_2,\ldots,v_n\}$ y un conjunto finito de bordes (o segmentos) $E=\{e_1, e_2,\ldots,e_m\}$ que los une. A cada borde "*e*" le corresponde un par de vértices *(u, v)*, a los cuales se dice que "*e* los incide".

Se habla de grafo dirigido (o *dígrafo* de forma abreviada) cuando los segmentos se representan mediante flechas que indican dirección; mientras que en un grafo no dirigido los segmentos son simples líneas. La figura 39 ilustra estas diferencias.

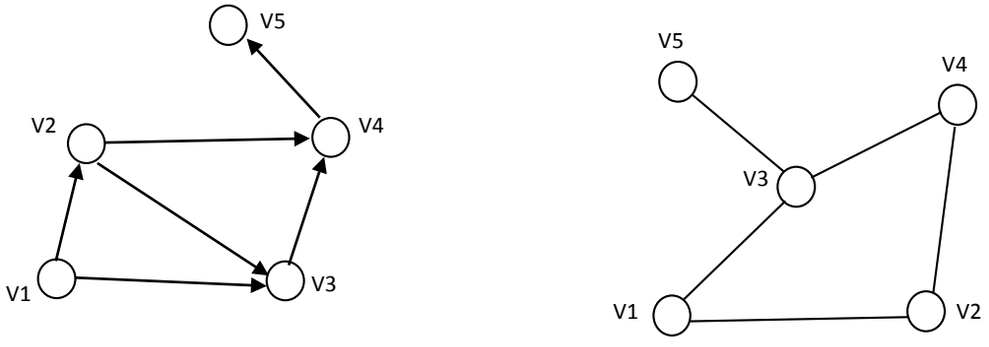


Figura 39. Representación de un grafo dirigido (izquierda) y uno no dirigido (derecha)
Fuente: Elaboración propia

Algunas diferencias entre ellos son las siguientes:

Tabla 1.
Diferencias entre grafos dirigidos y no dirigidos

GRAFO DIRIGIDO	GRAFO NO DIRIGIDO
• Grado de un vértice: o Grado de entrada se refiere a la cantidad de segmentos que llegan a un determinado vértice. o Grado de salida se refiere a la cantidad de segmentos que parten de él.	• Grado de un vértice: Número de segmentos que inciden en un determinado vértice (los bucles cuentan doble).
• Si existe un segmento (x,y), de x a y, el vértice x es llamado predecesor de y, mientras que el vértice y es llamado sucesor de x.	• Los vértices de un segmento se llaman adyacentes o vecinos.

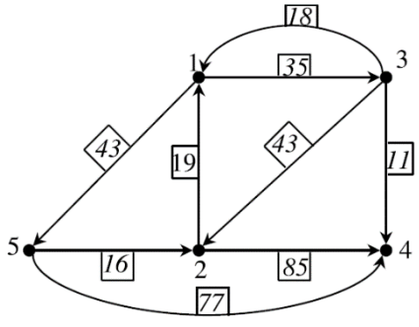
Elaboración propia

Un ejemplo de una situación real modelable con un grafo es el sistema de un aeropuerto. Cada aeropuerto es un vértice y dos vértices están conectados por una arista si hay un vuelo directo entre los aeropuertos. La arista podría tener un peso, que representa el tiempo, distancia o el costo del vuelo. Un segundo ejemplo es el flujo de tráfico, en el cual cada intersección de calles representa un vértice y cada calle es una arista. Los costos de las aristas podrían representar un límite de velocidad o una capacidad, número de carriles, entre otros. En ambos casos (aeropuerto o flujo de tráfico) podría determinarse la ruta más corta entre dos puntos, la ubicación más probable para un embotellamiento, entre otros (Weiss, 1995).

A continuación, se exponen definiciones básicas acerca de los grafos y sus componentes.

- Segmentos paralelos: Si dos segmentos tienen el mismo par de vértices.
- Multigrafo: Grafo con segmentos paralelos.
- Bucle: Cuando un segmento nace y termina en mismo vértice.
- Grafo simple: Cuando un grafo no tiene segmentos paralelos ni bucles.
- Grafo general: cuando tiene segmentos paralelos y bucles. Se denomina también pseudógrafo.
- Grafo ponderado: Cuando cada uno de sus segmentos tiene asignado un valor numérico, el cual representa el "ancho de segmento". En la práctica, este valor puede referirse a distancias de carreteras, costos de construcción, lapsos de tiempo, probabilidad, capacidad de carga o cualquier otro atributo (Ver figura 40).

Figura 40. Ejemplo de un grafo ponderado



Tomado de *Handbook of Data Structures and Applications, por* Mehta & Sahni, 2005.

- Densidad: Es la proporción entre el cuadrado de la cantidad de vértices (V^2) y el número de segmentos (E) de un grafo. En otras palabras, mientras E se aproxime a V^2 (proporción cercana a 1) se dice que se está ante un "grafo denso"; por el contrario, si los valores se alejan (proporción cercana a 0), se está ante un "grafo poco denso". Esta diferencia es importante resaltarla, ya que le permitirá elegir la estructura más adecuada para su implementación.

2. Representación de grafos

La representación abstracta de un grafo (componentes y operaciones posibles) se muestra a continuación:

Grafo
{
Instancias
Conjunto de vértices (*V*) y conjunto de segmentos (*E*)
Operaciones

NroVertices()	:	Retorna el número de vértices del grafo.
NroSegmentos	:	Retorna el número de segmentos del grafo.
ExisteSegmento(i,j)	:	Retorna verdadero si el segmento (i.j) existe en el grafo.
InsertarSegmento(*s*)	:	Insertar el segmento *s* en el grafo.
EliminarSegmento(i,j)	:	Elimina el segmento (i,j) del grafo.
Grado(i)	:	Retorna la cantidad de segmento del vértice *i* (solo grafos no dirigidos)
GradoEntrada(i)	:	Retorna la cantidad de segmentos que llegan al vértice *i*. (solo grafos dirigidos)
GradoSalida(i)	:	Retorna la cantidad de segmentos que parten del vértice *i*. (solo grafos dirigidos)

}

Figura 41. Representación abstracta de un grafo
Tomada de *Data Structures, Algorithms, and Applications in C++, por* Sahni, 2005, p. 625.

No obstante, a nivel de programación existen dos maneras de implementar los grafos, tal como lo proponen Mehta y Sahni (2005, p. 4-3):

a) **Lista de adyacencia:** Consiste en un *array* de n punteros, uno para cada vértice del grafo, tal que para *array[v]*, que hace referencia al vértice *v*, parten nodos enlazados por cada uno de sus vértices adyacentes.

b) **Matriz de adyacencia:** Consiste en una matriz numérica de *v* x *v* elementos, tal que la posición $a_{ij} = 1$ si existe un segmento que va del vértice i al j; sino $a_{ij} = 0$. Note que para representar un bucle bastará marcar con 1 la posición respectiva en la diagonal de la matriz, mientras que para representar segmentos paralelos se registrará un número mayor que 1 en la respectiva coordenada, pero hacer esto no es común.

La figura 42 ilustra estas dos formas de representación tanto para un grafo no dirigido como para uno dirigido.

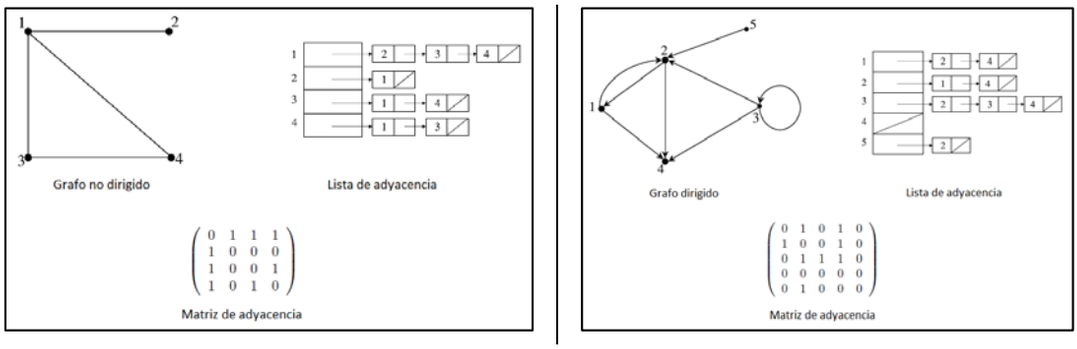


Figura 42. Representación de un grafo no dirigido (izquierda) y un grado dirigido (derecha)Tomada de *Handbook of Data Structures and Applications*, por Mehta & Sahni, 2005, p. 4-4.

Ahora bien, tanto la lista como la matriz de adyacencia pueden ser adaptadas para registrar los ponderados si así fuera necesario. La figura 43 muestra un ejemplo de ello a partir del grafo mostrado en la figura 40.

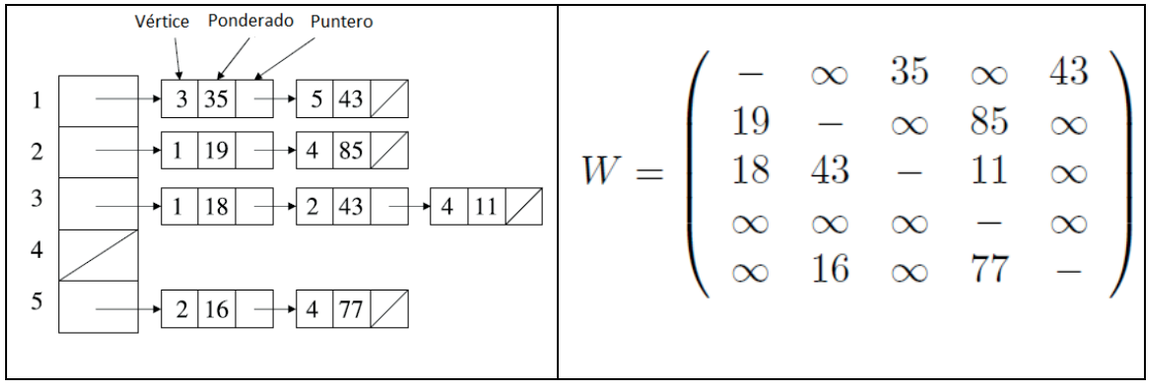


Figura 43. Lista de adyacencia y matriz de adyacencia para un grafo ponderado
Tomada de *Handbook of Data Structures and Applications, por* Mehta & Sahni, 2005, p. 4-5.

La razón para elegir la estructura más adecuada recae en la densidad del grafo. Los autores recomiendan implementar grafos poco densos con listas de adyacencia, mientras que para grafos muy densos sugieren hacerlo con matrices. Si usted elige una matriz para un grafo poco denso, tendrá varias casillas de las mismas marcadas con 0 (cero), ocupando espacios de memoria innecesariamente. Como expresa Weiss (2000), se trata de que nuestras estructuras de datos representen realmente los datos que haya y no los que no están presentes".

3. Operaciones con grafos

Las operaciones descritas en el numeral 2 (representación abstracta de un grafo) están orientadas hacia la programación, las cuales no ameritan mayor explicación ya que podrá implementarlas aplicando todo lo ya tratado anteriormente en este manual.

Esta sección, se ocupará más bien de exponer algunos algoritmos para aplicarlos en la solución de casos reales como los que se mencionan a continuación:

- Ordenación topológica
- Algoritmo del camino más corto

3.1. Ordenación topológica

Imagine un grafo dirigido cuyos vértices representan al conjunto de asignaturas que debe aprobar un estudiante antes de culminar su carrera y cada arista simboliza si una asignatura es requisito de otra. En este contexto, una ordenación topológica vendría a ser cualquier secuencia de asignaturas que no contravenga con la norma de requisito.

Ten en cuenta lo siguiente:

Una ordenación topológica no es posible si el grafo tiene ciclos y bucles. En el ejemplo, una asignatura no puede ser requisito de sí misma.

En términos generales, el algoritmo consiste en encontrar un nodo al que no entren aristas, se imprime y se elimina junto con estas; finalmente, se repite el proceso con los vértices restantes.

```
// Algoritmo para ordenamiento topológico
procedure ordenop(G:grafo);
   var C:COLA;
var contador: integer;
     v, w: vértice;
begin
   crear_nula(C);
   contar:=1;
   for cada vértice v do
     if gradient[v] = 0 then
       encolar(v, C);

   while not está_vacía(c) do
   begin
     v:=desencolar(c);
     núm_top[v]:=contador; [asigna el siguiente número]
     contador:=contador+1;

     for cada w adyacente a v do
     begin
       gradoent[w]:=gradoent[w]-1;
       if gradoent[w] = 0 then
         encolar(w, C);
     end;
   end;
   if Contador <= |V| then
     error("El grafo tiene bucles");
end;
```

Programa 2.1 Tomada de *Estructura de datos y algoritmos, por* Weiss, 1995, p. 300

3.2. Algoritmo del camino más corto

Generalmente resulta de interés encontrar los caminos, directos o indirectos, entre los vértices de grafo. A su vez, se requiere encontrar el camino más corto entre dos vértices. Los algoritmos más usados para este fin son los siguientes: Dijkstra, Floyd y Warshall.

Ten en cuenta

El camino más corto entre dos vértices de un grafo ponderado está determinado por la suma de los pesos de las aristas que unen tales vértices, mientras que en un grafo no ponderado está determinado por la cantidad de aristas entre ellos.

a) **Grafo no ponderado**

El algoritmo para encontrar el camino más corto en un grafo no ponderado se muestra a continuación:

```
procedure noponderado(var T:Tabla)
  var dist_act:integer;
    v, w:vértice;
begin
  for dist_act:=0 to NUM_VERT-1
    for cada vértice v
      if (T[v].conocido = false) and (T[v].dist = dist_act) then
```

```
        begin
          T[v].conocido = true;
          for cada e adyacente a v
            if (T[w].dist = MAXINT then
            begin
              T[w].dist:= dist_act + 1;
              T[w].camino := v;
            end;
        end;
end;
```

Programa 2.2 Tomado de *Estructura de datos y algoritmos*, por Weiss, 1995, p. 305

b) **Grafo ponderado**

Para este tipo de grafos la bibliografía propone tres algoritmos (Cairo & Guardati, 2010, p. 285-293):

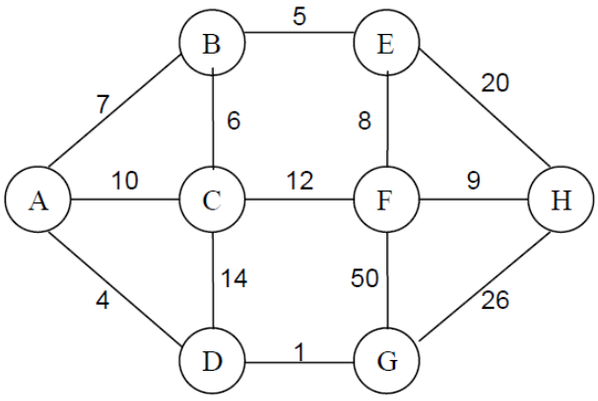
• Algoritmo de Dijkstra	- Encuentra el camino más corto de un vértice elegido a cualquier otro del dígrafo. - Las aristas deben tener peso no negativo.
• Algoritmo de Floyd	- Encuentra el camino más corto entre cada par de nodos del grafo. - La matriz de distancias sirve como punto de partida para este algoritmo. Se realizan k iteraciones sobre la matriz, por lo tanto, la k-ésima iteración, M[i,j] tendrá el camino de menor costo para llegar de i a j.
• Algoritmo de Warshall	- Encuentra, si es posible, un camino entre cada uno de los vértices del dígrafo. Es decir, la solución encontrada solo muestra si hay o no camino, más no la distancia entre los vértices.

El pseudocódigo del algoritmo Dijkstra lo describe Weiss (1995, p. 314) de la siguiente manera:

```
procedure dijkstra (var T:TABLA);
   Var i:integer;
     V, w: vértice;
begin
   for i:=1 to NUM_VÉRTICES
   begin
     v:= vértice con la distancia más corta desconocida;
     T[v].conocido = true;
     for cada w adyacente a v
       if T[w].conocido = false then
       begin
         if T[v].dist + c(v, w) < T[w].dist then // Actualiza w
         begin
           reducir(T[w].dist a T[v].dist + c(v, w));
           T[w].camino := v;
         end:
       end;
   end;
end;
```

Programa 2.3 Tomado de *Estructura de datos y algoritmos, por* Weiss, 1995, p. 314

A modo de ejemplo; a continuación, se presenta un grafo y su respectiva solución para encontrar el camino más corto a través del algoritmo Dijkstra.



	A	B	C	D	E	F	G	H
A	0	7	10	4	∞	∞	∞	∞
B	7	0	6	∞	5	∞	∞	∞
C	10	6	0	14	∞	12	∞	∞
D	4	∞	14	0	∞	∞	1	∞
E	∞	5	∞	∞	0	8	∞	20
F	∞	∞	12	∞	8	0	50	9
G	∞	∞	∞	1	∞	50	0	26
H	∞	∞	∞	∞	20	9	26	0

Figura 44. Grafo de ejemplo y su respectiva tabla
Tomada de *Apuntes elaborados. Tema 9. Grafos*, por Asencio, Quevedo y López (s.f.), p. 10.

Para realizar este algoritmo se requiere adicionalmente:

- Un arreglo "visitados" de tipo *boolean* y del mismo ancho que la matriz.
- Un puntero "actual" para recorrer el grafo.
- Un arreglo "costes" de tipo entero para almacenar la estimación de cada coste para llegar a cada nodo. Se requiere inicializarlo con valores altos de tal forma que todo número comparado con él sea menor. En este caso tal número será representado con la letra "i" (infinito).
- Un arreglo "ruta" para recordar el primer paso a dar para llegar a cada nodo por el camino de menor coste.

Desarrollo del algoritmo:

1. Se registra en "Actual" el nodo inicial: Actual = "A". Se marca A como visitado en el arreglo Visitados.

	A	B	C	D	E	F	G	H
Visitados	T	F	F	F	F	F	F	F

2. Se copian aquellos valores mayores que cero de la fila A de la matriz al arreglo Costes. Se rellenan las casillas correspondientes del arreglo Ruta con las direcciones de los nodos vecinos de "A".

	A	B	C	D	E	F	G	H
Costes	i	7	10	4	i	I	I	I
Ruta		B	C	D				

3. Del arreglo Costes se elige el menor valor de los que todavía no han sido visitados, es decir, "D", Actual = "D" y se le marca como visitado.

	A	B	C	D	E	F	G	H
Visitados	T	F	F	T	F	F	F	F

4. Se estima el coste (desde el nodo inicial "A") para llegar a cada uno de los nodos de "D" que aún no han sido visitados; es decir, "C" y "G".

	A	B	C	D	E	F	G	H
Costes	i	7	10	4	i	I	I	I
Ruta		B	C	D				

- Para llegar a C: 4 + 14 = 18 (4 se extrae del arreglo "costes", casilla "D").

  Se compara este valor con el coste actual registrado en "C"; si es menor, se reemplaza el dato.

  18 > 10 => continúa 10.

- Para llegar a G: 4 + 1 = 5.

  Se compara este valor con el coste actual registrado en "G"; si es menor, se reemplaza el dato.

  5 < i => se registra 5 en el arreglo costes; se copia el nodo usado para llegar a "G" en su respectiva casilla.

	A	B	C	D	E	F	G	H
Costes	i	7	10	4	i	I	5	I
Ruta		B	C	D			D	

5. Del arreglo costes se elige el menor valor de los que todavía no han sido visitados, es decir, "G" (Actual = "G"), y se le marca como visitado.

	A	B	C	D	E	F	G	H
Visitados	T	F	F	T	F	F	T	F

6. Se estima el coste para llegar a cada uno de los nodos de "G" que aún no han sido visitados.

- Para llegar a "F": 5 + 50 = 55 (5 se extrae del arreglo "costes", casilla "G")

  Se compara este valor con el coste actual registrado en "F"; si es menor, se reemplaza el dato.

  55 < i => se registra 55 en el arreglo costes; se copia el nodo usado para llegar a "F" en su respectiva casilla.

	A	B	C	D	E	F	G	H
Costes	i	7	10	4	i	55	5	I
Ruta		B	C	D		D	D	

- Para llegar a "H": 5 + 26 = 31.

  Se compara este valor con el coste actual registrado en "H"; si es menor, se reemplaza el dato.

  31 < i => se registra 31 en el arreglo costes; se copia el nodo usado para llegar a "H" en su respectiva casilla.

	A	B	C	D	E	F	G	H
Costes	i	7	10	4	i	55	5	31
Ruta		B	C	D		D	D	D

7. Del arreglo costes se elige el menor valor de los que todavía no han sido visitados, es decir, "B" (Actual = "B"), y se le marca como visitado.

	A	B	C	D	E	F	G	H
Visitados	T	T	F	T	F	F	T	F

8. Se estima el coste para llegar a cada uno de los nodos de "B" que aún no han sido visitados.

- Para llegar a "C": 7 + 6 = 13 (7 se extrae del arreglo "costes", casilla "B")

  Se compara este valor con el coste actual registrado en "C"; si es menor, se reemplaza el dato.

  13 > 10 => continúa 10.

- Para llegar a "E": 7 + 5 = 12.

  Se compara este valor con el coste actual registrado en "E"; si es menor, se reemplaza el dato.

  12 < i => se registra 12 en el arreglo costes; se copia el nodo usado para llegar a "E" en su respectiva casilla.

	A	B	C	D	E	F	G	H
Costes	i	7	10	4	12	55	5	31
Ruta		B	C	D	B	D	D	D

9. Del arreglo costes se elige el menor valor de los que todavía no han sido visitados; es decir, "C" (Actual = "C"), y se le marca como visitado.

	A	B	C	D	E	F	G	H
Visitados	T	T	T	T	F	F	T	F

10. Se estima el coste para llegar a cada uno de los nodos de "C" que aún no han sido visitados.

- Para llegar a "F": 10 + 12 = 22 (10 se extrae del arreglo "costes", casilla "C")

  Se compara este valor con el coste actual registrado en "F"; si es menor, se reemplaza el dato.

  22 < 55=> se registra 22 en el arreglo costes; se copia el nodo usado para llegar a "F" en su correspondiente casilla.

11. Del arreglo costes se elige el menor valor de los que todavía no han sido visitados, es decir, "E" (Actual = "E"), y se le marca como visitado.

	A	B	C	D	E	F	G	H
Visitados	T	T	T	T	T	F	T	F

12. Se estima el coste para llegar a cada uno de los nodos de "E" que aún no han sido visitados.

- Para llegar a "F": 12 + 8 = 20 (12 se extrae del arreglo "costes", casilla "E")

  Se compara este valor con el coste actual registrado en "F"; si es menor, se reemplaza el dato.

  20 < 22 => se registra 20 en el arreglo costes; se copia el nodo usado para llegar a "F" en su correspondiente casilla.

	A	B	C	D	E	F	G	H
Costes	i	7	10	4	12	20	5	31
Ruta		B	C	D	B	B	D	D

- Para llegar a "H": 12 + 20 = 32 (12 se extrae del arreglo "costes", casilla "E")

  Se compara este valor con el coste actual registrado en "H"; si es menor, se reemplaza el dato.

  32 > 31 => continúa 31.

13. Del arreglo costes se elige el menor valor de los que todavía no han sido visitados, es decir, "F" (Actual = "F"), y se le marca como visitado.

	A	B	C	D	E	F	G	H
Visitados	T	T	T	T	T	T	T	F

14. Se estima el coste para llegar a cada uno de los nodos de "F" que aún no han sido visitados.

- Para llegar a "H": 20 + 9 = 29 (20 se extrae del arreglo "costes", casilla "F")

  Se compara este valor con el coste actual registrado en "H; si es menor, se reemplaza el dato.

  29 < 31 => se registra 29 en el arreglo costes; se copia el nodo usado para llegar a "H" en la casilla correspondiente.

	A	B	C	D	E	F	G	H
Costes	i	7	10	4	12	20	5	29
Ruta		B	C	D	B	B	D	B

15. Del arreglo costes se elige el menor valor de los que todavía no han sido visitados, es decir, "F" (Actual = "F"), y se le marca como visitado.

	A	B	C	D	E	F	G	H
Visitados	T	T	T	T	T	T	T	T

16. Se estima el coste para llegar a cada uno de los nodos de "F" que aún no han sido visitados. Todos los nodos ya han sido visitados, por lo tanto, acaba el algoritmo.

	A	B	C	D	E	F	G	H
Costes	i	7	10	4	12	20	5	29
Ruta		B	C	D	B	B	D	B

Ruta más corta: A – B– E – F – H.

Tomada de *Apuntes elaborados. Tema 9: Grafos*, por Asencio, Quevedo y López (s.f.), pp. 11-14.

Implemente este algoritmo en C++ y realice las pruebas respectivas.

Por su parte los algoritmos de Floyd y Marshall siguen la siguiente secuencia:

```
// Algoritmo de Floyd
for k=1 hasta N
  for i=1 hasta N
    for j=1 hasta N
      if (Mik + Mkj< Mij) then
        Mij = Mik + Mkj
```

Tomada de *Estructuras de datos* (3.ª ed.), por Cairo y Guardati, 2010, p. 289.

Donde N es el número total de vértices del grafo; M es una matriz de N x N elementos y se inicia con los costos del dígrafo, y k, i, j son variables enteras.

```
// Algoritmo de Warshall
for k=1 hasta N
  for i=1 hasta N
    for j=1 hasta N
      if (A[i, j] = 0 then
        A[i, j] = A[i, k] y A[k, j];
```

Tomada de *Estructuras de datos* (3.ª ed.), por Cairo y Guardati, 2010, p. 293

Donde N es el número total de vértices del grafo; A es una matriz de N x N elementos, inicialmente es igual a la matriz del grafo, y k, i, j son variables enteras.

TEMA N.º 3: ÁRBOLES

Los árboles, aunque similares a los grafos, encuentran su aplicación en otros campos de la computación, principalmente en el diseño de compiladores y búsqueda de datos de forma ágil. Como en el tema 2, el contenido está orientado a brindar pautas y algoritmos referidos a la aplicación de los árboles en vez de su implementación.

1. Definición

Un árbol es un tipo especial de grafo, cuya característica diferenciadora es que cada nodo, a excepción del primero, tiene solo un predecesor. Un árbol es la representación natural de información jerárquica y, gráficamente, es como se muestra en la figura 45.

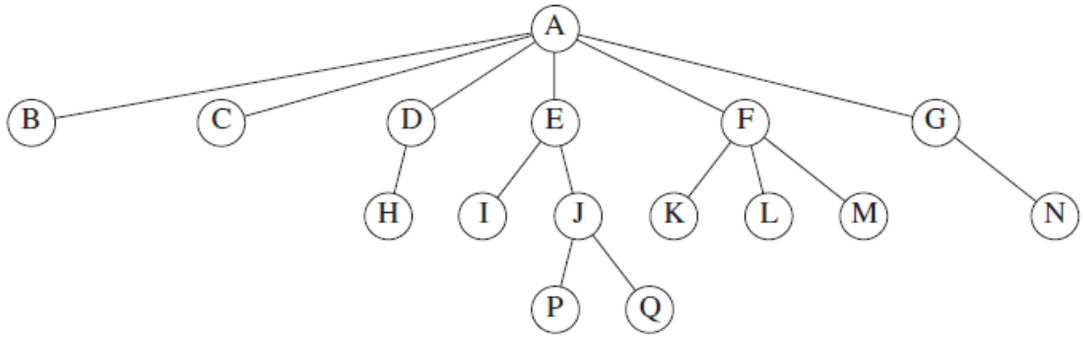


Figura 45. Ejemplo de árbol
Tomada de *Data structures and algorithm analysis in C++, por* Weiss,2014, p. 122

De esta estructura se desprenden los siguientes conceptos:

a) Nodo raíz: Primer nodo, sin predecesor (A).

b) Nodo padre: Nodo del cual parten otros nodos (A, D, E, F, G, J). Por ejemplo: "E" es padre de "I" y "J".

c) Nodo hijo: Nodo que sucede a otro. Por ejemplo "H" es hijo de "D".

d) Nodos hermanos: Nodos que comparten el mismo padre. Por ejemplo "K", "L" y "M" son hermanos.

e) Hojas: Nodos sin hijos. Por ejemplo "B", "C", "H", "I", etc.

De igual forma se diferencia entre:

a) Ruta: Secuencia de nodos que unen otros dos. Por ejemplo "A", "E", "J", "Q" es una ruta. Note que existe solamente una ruta desde la raíz de un árbol a una determinada hoja.

b) Longitud: Es el número de segmentos en una ruta.

c) Profundidad: Longitud desde la raíz a un determinado nodo.

d) Altura: Longitud de la ruta más larga desde un determinado nodo a una hoja. Note que la altura de un árbol es igual a la altura de la raíz.

2. Implementación de árboles

Ya que un árbol es un tipo de grafo, su implementación (en C++) también se basará en estructuras (*struct*), aunque con una lógica distinta.

A primera vista la lógica indica que el árbol se creará únicamente haciendo que un determinado nodo apunte a cada uno de sus nodos hijos; sin embargo, esto no necesariamente es así, ya que no es posible conocer con antelación la cantidad de los mismos. La solución es emplear una lista enlazada en la que cada nodo posea tres campos: *dato*, *primer_hijo* y *hermano_siguiente*. Observe la figura 46.

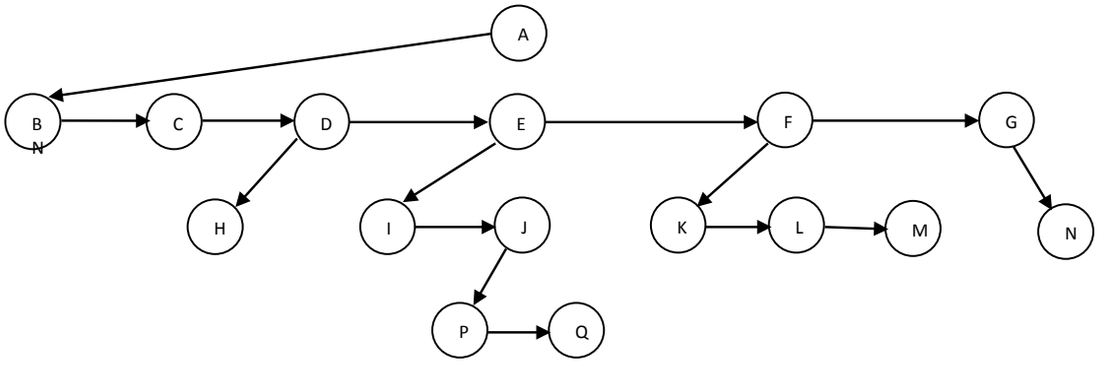


Figura 46. Representación del árbol de la figura 25 basado en nodos primer_hijo y siguiente_hermano
Tomada de *Data structures and algorithm analysis in C++, por* Weiss,2014, p. 123

Las flechas que se dirigen hacia abajo representan punteros hacia el *primer_hijo* de cada nodo, mientras que las que se dirigen hacia la derecha representan su *siguiente_hermano*.

En C++ la estructura que representa cada nodo es:

```
struct NodoArbol
{
      int dato;
      NodoArbol *primer_hijo;
      NodoArbol *sgte_hmano;
};
```

Las reglas para crear el árbol e insertar/eliminar nodos serán las mismas que las empleadas para las listas enlazadas. El programa principal deberá invocar recursivamente a la estructura (*struct*) a medida se agreguen nuevos elementos al árbol.

3. Árbol binario

En computación existe un tipo de árbol muy utilizado en la solución de diversos tipos de problemas: el árbol binario. Su particularidad radica en que cada nodo puede tener a lo más dos nodos hijos, de ahí la denominación de "*bi-nario*".

De forma genérica cada nodo representa una raíz (r), y sus nodos descendientes se llaman "árbol izquierdo" (A_{iz}) y "árbol derecho" (A_{de}) respectivamente (ver figura 47).

Figura 47. Representación simbólica de un árbol binario (izquierda) y un ejemplo del mismo (derecha)
Fuente: Elaboración propia

3.1. Implementación de árboles binarios

Ya que en este tipo de árbol se conoce de antemano la cantidad de nodos hijos, la estructura (*struct*) para su implementación es la siguiente:

```
struct NodoBinario
{
   int dato;
   NodoBinario *izquierdo;
   NodoBinario *Derecho;
};
```

3.2. Aplicación de árboles binarios

Este manual puntualizará en las aplicaciones:

a) Representación de expresiones matemáticas

b) Búsqueda

3.2.1. **Representación de expresiones matemáticas**

Esta aplicación de los árboles binarios, propia del diseño de compiladores, consiste en representar jerárquicamente una secuencia de caracteres que constituya una operación matemática. La representación finalizada se denomina árbol de expresión. La figura 48 muestra un ejemplo.

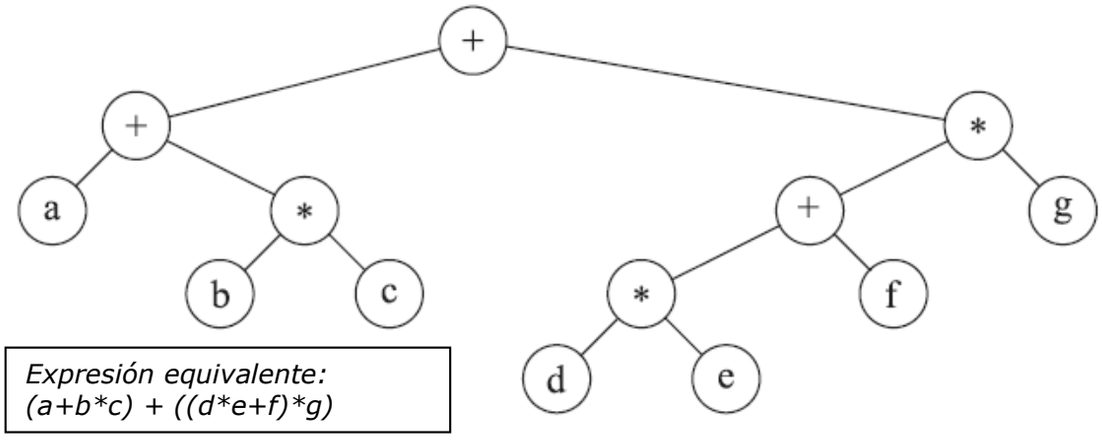


Figura 48. Ejemplo de árbol de expresión y su equivalente matemático
Tomada de *Data structures and algorithm analysis in C++, por* Weiss, 2014, p. 129.

Para crear un programa capaz de construir este tipo de árboles es imprescindible que antes se transforme la expresión a su representación posfija. Para ello, partiendo de la raíz, se visualiza recursivamente el árbol izquierdo, luego el árbol derecho y después el operador.

Para el árbol de la figura 48, la expresión posfija equivalente es la siguiente:

a b c * + d e * f + g * +

El algoritmo para crear un árbol a partir de una expresión posfija se muestra a continuación:

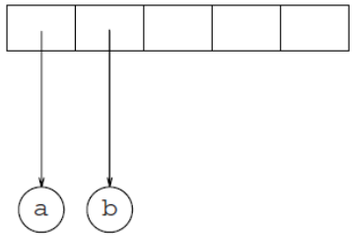
- Leer la expresión posfija símbolo por símbolo.

- Si el símbolo es un operando, se crea un nodo e inmediatamente un apuntador a él en una pila.
- Si el símbolo es un operador, se crea el nodo para tal y, de los operandos que aguardan en la pila, se copian sus direcciones en el nodo creado. Entonces, estas direcciones son eliminadas de la pila, quedando solo una que apunta hacia el operador.

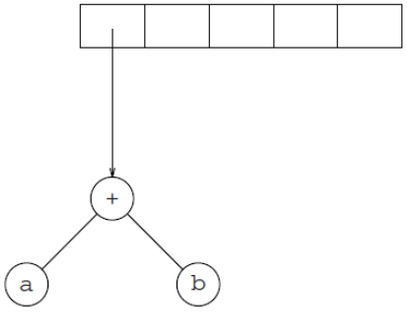
A continuación, un ejemplo del algoritmo: (el ejemplo y los gráficos fueron extraídos de Weiss, 1995, p. 102-104)

Expresión: a b + c d e + * *

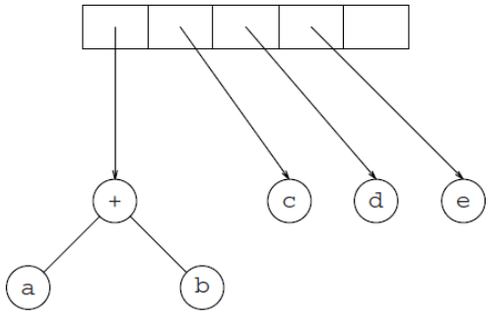
- Leer símbolo: "a". Como se trata de un operando, se agrega un puntero hacia él en la pila.
- Leer siguiente símbolo: "b". Como se trata de un operando, se agrega un puntero hacia él en la pila.



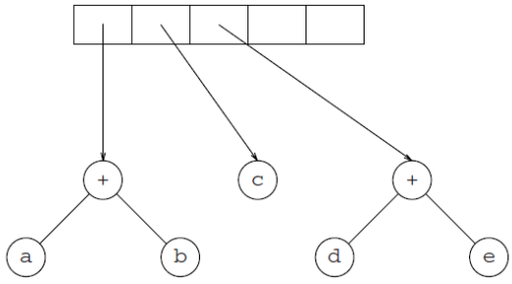
- Leer siguiente símbolo: "+". Como se trata de un operador, se copian los punteros en su respectivo nodo desde la pila. Entonces, se eliminan tales direcciones de la pila, quedando solo una hacia el último nodo creado.



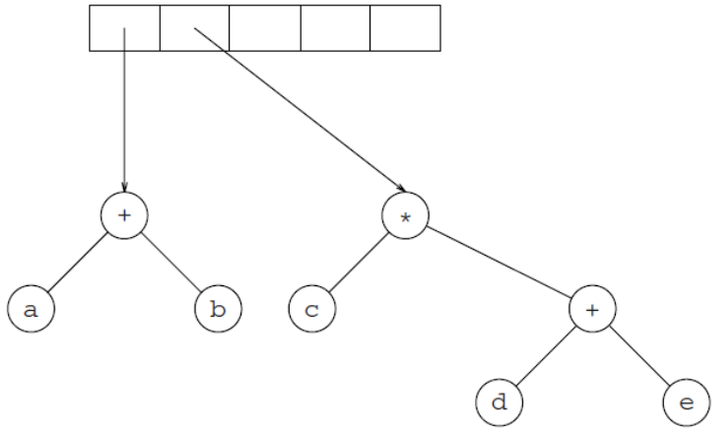
- Leer símbolos siguientes: "c", "d" y "e". Todos operandos.



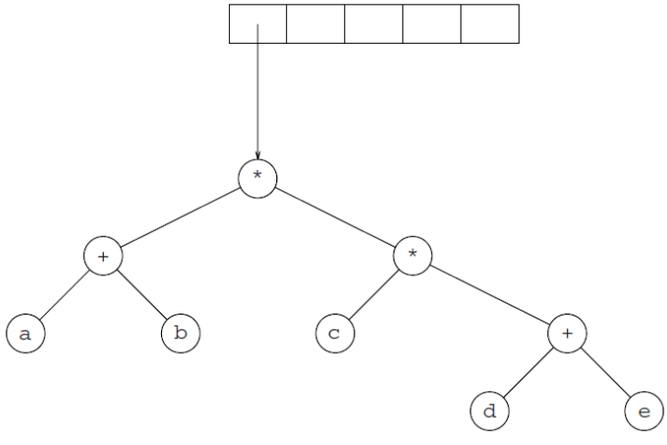
- Leer símbolo siguiente: "+". Se trata de un operador.



- Leer símbolo siguiente: "*".



- Leer símbolo siguiente: "*".



3.2.2. **Búsquedas en árboles binarios**

Para recorrer un árbol binario es necesario que cada uno de sus nodos posea un valor llave único, el cual puede ser un dato de tipo entero u otro más complejo.

La propiedad que convierte un árbol binario en un árbol binario de búsqueda, es que para cada nodo todos los valores de su subárbol izquierdo son menores que él, mientras que los del subárbol derecho son mayores. La diferencia se muestra gráficamente en la figura 49.

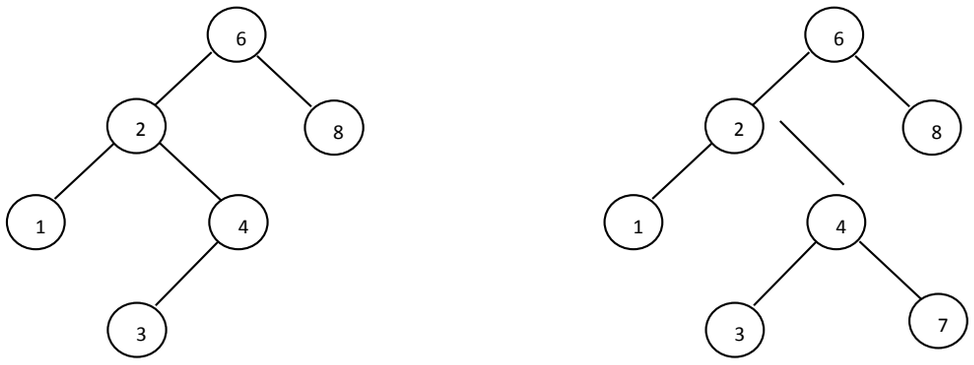


Figura 49. Árbol binario de búsqueda (izquierda) y árbol binario genérico (derecha)
Tomada de *Data structures and algorithm analysis in C++.* Weiss, 2014, p. 132

El árbol de la derecha no puede considerarse como uno de búsqueda debido al nodo 7, ya que si bien cumple la propiedad respecto a su nodo padre "4" (es mayor y posicionado a la derecha), no es así respecto al nodo raíz "6"; al encontrarse el nodo 7 en el subárbol izquierdo de este debería ser menor que la raíz.

Proceso de búsqueda

Al realizar la búsqueda de una determinada llave, el resultado debe ser el apuntador hacia el nodo que la posee, o *null* si no existe tal nodo. Weiss (1995, p. 106) propone el siguiente algoritmo para esta tarea:

```
        function buscar (x: integer; A: Árbol_de_búsqueda): ap_árbol;
        begin
            if A = null then
                  buscar := null;
            else
            if x < A^.elemento then
                  buscar := buscar(x, A^.izquierdo)
            else
            if x > A^.elemento then
                  buscar := buscar(x, A^.derecho)
            else
                  buscar := A;
end; [buscar]
```

Programa 3.1

Su correspondiente implementación y un ejemplo de cómo usar la función buscar para insertar un nuevo elemento al árbol, se muestra en el programa 3.2.

```
        struct nodoArbol
        {
            int dato;
            nodoArbol *a_izq;
            nodoArbol *a_der;
        };

        typedef struct nodoArbol *TArbol;

        nodoArbol *buscar(TArbol &arbol, int pDato)
        {
            if (arbol== NULL)
            {return NULL;}
```

```
    else if (pDato == arbol->dato)
    {return arbol;}

    else if (pDato < arbol->dato) // Si es menor buscar a la izquierda
    {buscar(arbol->a_izq, pDato);}

    else if (pDato > arbol->dato) // Si es mayor buscar a la derecha
    {buscar(arbol->a_der, pDato);}
}

void insertarEnNodo(TArbol &arbol, int pNodo, int pDato, char pLado)
{
    // Capturar dirección del nodo buscado
    nodoArbol *nodoBuscado;
    nodoBuscado = buscar(arbol, pNodo);

    if (nodoBuscado == NULL)
          cout<<"El nodo no existe"<<endl;
    else
    {
          // Crear nuevo
          nodoArbol *nuevonodo;
          nuevonodo = new struct nodoArbol;

          nuevonodo->dato = pDato;
          nuevonodo->a_izq = NULL;
          nuevonodo->a_der = NULL;

          if (pLado == 'I')
                nodoBuscado->a_izq = nuevonodo;
          else
                nodoBuscado->a_der = nuevonodo;
    }
}

void imprimir(TArbol &arbol, int n)
{
    if(arbol==NULL)
          return;

imprimir(arbol->a_der, n+1);

for(int i=0; i<n; i++)
          cout<<"   ";

    cout<< arbol->dato <<endl;

imprimir(arbol->a_izq, n+1);
}
```

Programa 3.2

Es preciso aclarar que el procedimiento de inserción del programa 3.2 solo busca el nodo solicitado y, si existe, a continuación inserta el nuevo nodo en la posición indicada (izquierda o derecha); no realiza la tarea de validar si el dato infringe la propiedad de un árbol binario de búsqueda (ver numeral 3.2.2), por lo que esta tarea se le deja a usted, estimado estudiante.

Asimismo, una adaptación de esta función de búsqueda permite devolver los valores mínimo y máximo de todo el árbol: el menor valor estará ubicado en el último nodo a la izquierda, mientras que el mayor estará en el último nodo a la derecha; por lo tanto, el programa deberá simplemente recorrer el árbol por alguno de estos lados dependiendo del dato deseado. Observe el programa 3.3

```
int buscar_minimo(TArbol &arbol)
{
      int valor_minimo;
      if (arbol == NULL)
            valor_minimo = -1;
      else if (arbol->a_izq == NULL)
            valor_minimo = arbol->dato;
      else
            valor_minimo = buscar_minimo(arbol->a_izq);
      return valor_minimo;
}

int buscar_maximo(TArbol &arbol)
{
      int valor_maximo;
      if (arbol == NULL)
            valor_maximo = -1;
      else if (arbol->a_der == NULL)
            valor_maximo = arbol->dato;
      else
            valor_maximo = buscar_minimo(arbol->a_der);
      return valor_maximo;
}
```

Programa 3.3

Finalmente, el programa 3.4 ilustra cómo imprimir los valores de los nodos tomando en cuenta los tres posibles tipos de ordenamiento: pre-orden (prefija), en orden y pos-orden (posfija).

```
void preOrden(TArbol arbol)
{
     if(arbol!=NULL)
     {
          cout << arbol->dato <<" ";
          preOrden(arbol->a_izq);
          preOrden(arbol->a_der);
     }
}

void enOrden(TArbol arbol)
{
     if(arbol!=NULL)
     {
          enOrden(arbol->a_izq);
          cout << arbol->a_dato<< " ";
          enOrden(arbol->a_der);
     }
}

void postOrden(TArbol arbol)
{
```

```
      if(arbol!=NULL)
      {
            postOrden(arbol->a_izq);
            postOrden(arbol->a_der);
            cout << arbol->dato<< " ";
      }
}
```

Programa 3.4. Adaptado del blog "Bloguando sobre programación". Recuperado de http://blog.martincruz.me/2012/11/arboles-binarios-de-busqueda-c.html

Lectura seleccionada n.º 3

Los grafos

Esta lectura le permitirá conocer la situación real que conllevó al origen de la teoría de grafos, así como su vasto campo de aplicación tanto en el campo de la ingeniería como de diversas ciencias.

García, F. (2004). La magia de los grafos. *Revista de La Asociación de Autores Científico-Técnicos y Académicos*, *34*, pp. 31–47. Disponible en el aula virtual.

Actividad N.º 3

Foro de discusión sobre grafos

Instrucciones:

a) Escriba un programa para efectuar la ordenación topológica de un grafo (numeral 3.1)

b) Ingrese al foro y participe respondiendo las siguientes preguntas:

Si se usa una pila en vez de una cola para el algoritmo de ordenación topológica, ¿se obtiene una ordenación topológica diferente? ¿En qué aspectos?

Actividad N.º 4

Foro de discusión sobre árboles

Instrucciones:

a) Ingrese al foro y participe respondiendo la siguiente pregunta:

Si se optara por implementar un árbol binario mediante listas enlazadas (estructuras de dos campos: dato, puntero), en la que cada nodo almacenara un carácter de su expresión posfija equivalente, ¿qué ventajas y desventajas ofrecería a las operaciones de búsqueda e inserción de datos? (Mencione 2 de cada una).

Glosario de la Unidad III

B

Binario. Compuesto de dos elementos, unidades o guarismos. (RAE, 2016)

N

Nodo. En un esquema o representación gráfica en forma de árbol, cada uno de los puntos de origen de las distintas ramificaciones. (RAE, 2016)

P

Ponderar (de Grafo ponderado). Determinar el peso de algo. (RAE, 2016)

S

Struct (de C++). La palabra clave struct define un tipo de estructura o una variable de un tipo de estructura. (MSDN, Microsoft)

T

Topología. Rama de las matemáticas que trata especialmente de la continuidad y de otros conceptos más generales originados de ella, como las propiedades de las figuras con independencia de su tamaño o forma. (RAE, 2016)

Bibliografía de la Unidad III

Aho, A., Ullman, J. & Hopcroft, (1983). *Data Structures and algorithms.* Nueva Delhi: Pearson Education.

Asencio, A., Quevedo, E. & López, R. (*s/f*). *Apuntes elaborados. Tema 9: Grafos* [en línea]. España: Universidad de las Palmas de Gran Canaria. Disponible en http://www.iuma.ulpgc.es/users/jmiranda/docencia/programacion/Tema9_ne.pdf

Cairo, O. & Guardati, S. (2010). *Estructuras de datos* (3.ª ed.). México: Editorial McGraw Hill.

Charles, S. (2009). *Python para informáticos: Explorando la información.*

Cruz, D. (*s/f*). *Apuntes de la asignatura: Estructura de datos* [diapositiva]. México: Tecnológico de Estudios Superiores de Ecatepec. Disponible en: http://myslide.es/documents/manual-estructura-de-datos.html

Cruz, M. (s.f.). *Arboles Binarios de Busqueda en C++. Recorrido por niveles (Amplitud)* [Blog post]. http://blog.martincruz.me/2012/11/arboles-binarios-de-busqueda-c.html

García F. (*s.f.*). *La magia de los grafos.* Autores científico-técnicos y académicos. Disponible en http://www.acta.es/medios/articulos/matematicas/034029.pdf

Ignacio, J., & Zahonero, I. (2003). *Programación en C. Metodología, Algoritmos y estructura de datos.* Málaga, España: Editorial McGraw Hill.

Mehta, D. P., & Sahni, S. (2005). *Handbook of Data Structures and Applications.* Florida, U.S.A.: Editorial Chapman & Hall/CRC. Disponibe en http://www.e-reading.club/bookreader.php/138822/Mehta_-_Handbook_of_Data_Structures_and_Applications.pdf

Pérez, J. (2004). *Capítulo 6. Matrices y determinantes.* España: Instituto Nacional de Tecnologías Educativas y de Formación del Profesorado. Disponible en: http://sauce.pntic.mec.es/~jpeo0002/Archivos/PDF/T06.pdf

Real Academia de la Lengua Española. (2016). Disponible en: http://www.rae.es/

Sahni, S. (*s.f.*). *Data structures, algorithms, and applications in C++. Hyderabad, India:* Universities Press.

Weiss, M. (1995). *Estructura de datos y algoritmos.* EE. UU: Addison-Wesley Iberoamericana.

Weiss, M. (2014). *Data structures and algorithm analysis in C++.* U.S.A.: Pearson Education. Dispoible en http://www.uoitc.edu.iq/images/documents/informatics-institute/Competitive_exam/DataStructures.pdf

Autoevaluación N.º3

1. ¿Qué es una pila?

   a. Estructura de tipo LIFO

   b. Estructura de tipo FOFI

   c. Estructura de tipo FIFO

   d. Estructura de tipo FOLI

2. ¿Qué es una cola?

   a. Estructura de tipo LIFO

   b. Estructura de tipo FOFI

   c. Estructura de tipo FIFO

   d. Estructura de tipo FOLI

3. Una pila o cola se pueden implementar con los siguientes elementos de programación:

   i) Estructuras (*struct*)

   ii) Matrices

   iii) Arreglos

   iv) Punteros

   a. Solo iii y iv

   b. Solo iii

   c. Solo i, iii, iv

   d. Todos

4. Emplea punteros para su implementación:

   a. Lista enlazada

   b. Arreglos

   c. Matrices

   d. Ninguno

5. Es un conjunto finito de aristas y segmentos que los unen:

   a. Matriz
   b. Lista
   c. Árbol
   d. Grafo

6. La instrucción typedef de C++ permite

   a. Crear variables de diferentes tipos
   b. Crear tipos de datos
   c. Crear datos pesonalizados
   d. Crear un alias para tipos de datos

7. Los componentes de un nodo (struct) de árbol binario son

   a. Dato, puntero
   b. Puntero, dato, puntero
   c. Dato, variable, puntero
   d. Puntero, variable, arreglo

8. La cantidad de segmentos de un árbol binario, respecto al número vértices es

   a. Vértices + 1
   b. Vértices–1
   c. Vértices / segmentos
   d. Vértices / caminos

9. Correlacione las propiedades de los árboles con su definición

i)	Ruta	w.	Longitud desde la raíz a un determinado nodo
ii)	Longitud	x.	Secuencia de nodos que unen otros dos
iii)	Profundidad	y.	Longitud de la ruta más larga desde un determinado nodo a una hoja
iv)	Altura	z.	Es el número de segmentos en una ruta

   a. i-z, ii-x, iii-w, iv-y
   b. i-x, ii-z, iii-y, iv-w
   c. i-x, ii-z, iii-w, iv-y

d. i-x, ii-w, iii-y, iv-z

10. Un árbol binario de búsqueda se diferencia de un árbol binario genérico en que

a. Todos los elementos están ordenados desde la raíz hacia las hojas.

b. Todos los elementos están ordenados de tal forma que los menores queden a la izquierda y los mayores a la derecha.

c. Todos los elementos están ordenados según su valor.

d. Todos los elementos están ordenados desde las hojas hacia la raíz.

UNIDAD IV

ORGANIZACIÓN DE DATOS Y ARCHIVOS

DIAGRAMA DE ORGANIZACIÓN

ORGANIZACIÓN DE LOS APRENDIZAJES

Resultado de aprendizaje de la Unidad IV:

Al finalizar la unidad, el estudiante será capaz de reconocer la organización de datos y archivos a través del estudio de casos.

CONOCIMIENTOS	HABILIDADES	ACTITUDES
Tema n.º 1: Tablas Hash. **Tema nº 2: Modelo de datos relacional.** **Tema n.º 3: Organización de archivos.** **Lectura seleccionada 1:** Sistemas de bases de datos frente a sistemas de archivos Autoevaluación de la Unidad IV	• Analiza y desarrolla algoritmos con Hash. **Actividad N.º 1** Foro de discusión • Analiza las herramientas de los modelos de datos relacionales y los aplica en las diversas situaciones de la vida real. • Identifica las ventajas del uso de la organización de archivos. • Identifica datos relacionales para un caso de estudios. **Actividad N.º 2** Foro de discusión	• Demuestra perseverancia y esfuerzo durante el desarrollo de los ejercicios. • Toma conciencia de la importancia de la asignatura en su formación profesional. • Valora las relaciones entre sus compañeros.

TEMA N.º 1: TABLAS HASH

La velocidad de inserción y búsqueda de un determinado dato en un arreglo es directamente proporcional al tamaño de este último; es decir, mientras más grande sea el arreglo, mayor será el tiempo que se necesite para alguna de estas actividades. Frente a este problema, surgen las tablas Hash, las cuales permiten acceder de forma directa -y por consiguiente rápida- a cualquiera de sus elementos; por tanto, representan una gran ventaja cuando se tiene que trabajar con grandes cantidades de datos.

1. Definición

"Una tabla Hash es una estructura de datos que permite almacenar objetos (o registros) de forma tal que una búsqueda determinada tenga un tiempo de recuperación constante, sin importar la cantidad de elementos que tenga la tabla". (Frittelli, s/f, p. 1)

En términos técnicos, una tabla Hash o tabla de dispersión está compuesta por un arreglo y un conjunto de operaciones matemáticas (llamada función de dispersión) que determinan la posición donde se insertará, eliminará o buscará un elemento en particular.

Por ejemplo, como se observa en la figura 50, la función de dispersión recibe el dato que desea insertarse en el arreglo (momento en el que el dato pasa a llamarse "Llave") y, luego de aplicarle una serie de operaciones, devuelve el índice donde será almacenado.

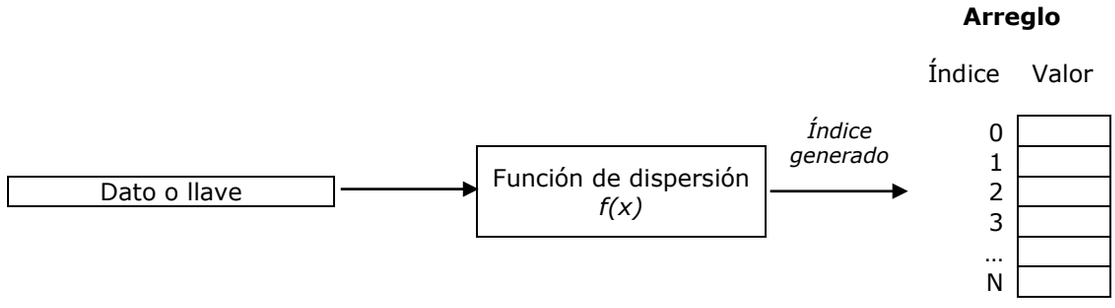


Figura 50. Proceso de inserción de un dato empleando una función de dispersión
Elaboración propia.

Concretamente, necesitamos una función que transforme llaves (números o cadenas) en enteros en el rango [0...M-1], donde M representa el número de registros del arreglo.

Por supuesto, las funciones de dispersión deberán ser implementadas de tal forma que distribuyan homogéneamente las llaves entre las celdas (Allen, 1995, p. 156); es decir, abarcar todo el arreglo y evitar las colisiones o, en palabras sencillas, que dos llaves no caigan en la misma posición.

2. Función de dispersión

a) Para llaves tipo entero:

- Método de la división: La función de dispersión devuelve el residuo de dividir la llave por el número casillas que tiene la tabla. Por ejemplo, si la tabla Hash es de tamaño 12 y la llave a insertar es 100, entonces:

$$h(k) = k \; mod \; m$$
$$h(100) = 100 \; mod \; 12 = 4$$

Por lo tanto, el valor 100 (la llave) será insertado en la posición 4 del arreglo.

De acuerdo con López (2002), es necesario tomar en cuenta las siguientes recomendaciones para el valor de "m" cuando se use el método de la división:

- Evitar que sea un número potencia de 2.
- Buenos valores son números primos y lejanos a potencias de 2.
  - Método de la multiplicación: Su algoritmo de operación se resume en dos pasos:

    Primero se multiplica la llave *k* por una constante *A* en el rango] 0,1 [y se extrae la parte fraccionaria del resultado. Luego, se multiplica este valor por *m* y se toma la parte entera del resultado.

    Por ejemplo, para una llave *m*=10000, *k*=123456 y *A*=0.618033…

$$h(k) = entero(m * ((kA \; mod \; 1)))$$
$$h(123456) = entero(10000 * (123456 * 0.618033 \; mod \; 1))$$
$$h(123456) = entero(10000 * (76300.0041151 \ldots mod \; 1))$$
$$h(123456) = entero(10000 * 0.0041151 \ldots) = 41$$

Entonces, el valor 10000 será insertado en la posición 41 del arreglo.

Recuerda:

Mod es un operador que devuelve el residuo de una división.

b) Para llaves tipo cadena:

Una opción es sumar el código ASCII de cada uno de sus caracteres. El siguiente algoritmo ilustra esta operación:

```
function dispersión (llave: tipo_cadena; tamaño_llave: integer): ÍNDICE;
 var val_dispersión, j: integer
 begin
    val_dipersión := codASCII(llave[1]);
    for j:=2 to tamaño_llave do
        val_dispersión := val_dispersión + codASCII(llave[j]);
    dispersión := val_dispersion mod tamaño;
    end
```

Aunque esta función opera con rapidez, no es una buena alternativa cuando se trata de arreglos grandes. Por ejemplo, si se tiene un arreglo un arreglo de tamaño 10007 y con palabras de 8 caracteres como máximo, entonces el mayor índice esperado será (122*8)=976, puesto que el valor más alto devuelto por *codASCII* será 122 (código de la letra zeta); por lo tanto, quedarán las casillas 977 a 10007 vacías.

Ahora observe y analice el siguiente algoritmo:

```
function dispersión (llave: tipo_cadena; tamaño_llave: integer): ÍNDICE;
begin
 dispersión := (codASCII(llave[1])+codASCII(llave[2])*27+codASCII(l-
lave[3])*729) mod tamaño;
```

En este caso, para una llave de 3 caracteres como máximo se intenta que el valor generado por la función de dispersión no pierda representatividad (tienda a 0) cuando la cadena tenga 1 o 2 caracteres (menor que 3). Para evitar ello, se multiplica cada posición con una potencia de 27 (que es el número de letras del abecedario).

$$llave[1] * 27^0 + llave[2] * 27^1 + llave[3] * 27^2$$

Aunque existen $27^3 = 19683$ combinaciones posibles con tres letras y se esperaría una distribución aproximadamente homogénea en el arreglo ya propuesto (de tamaño 10007), lo cierto es lo siguiente:

- No todas esas combinaciones son palabras válidas.
- Es una solución temporal, ya que si el arreglo creciera sobre 19683, se estaría ante el mismo problema del primer algoritmo.

Analice el tercer intento de una función de dispersión para llaves de tipo cadena.

```
function dispersión (llave: tipo_cadena; tamaño_llave: integer): ÍNDICE;
  var val_dispersión, j:integer;
begin
  val_dispersión := codASCII(llave[1]);
  for:=2 to tamaño_llave do
     val_dispersión:=(val_dispersion*32 + codASCII(llave[j]))
mod tamaño;
  dispersión:=val_dispersión
end
```

Este algoritmo ya considera a todos los caracteres de la llave (al margen de su tamaño) y se puede esperar una buena distribución, pues lleva el resultado al intervalo apropiado. La operación usa 32 (en vez de 27), ya que no es una multiplicación propiamente dicha, sino más bien se trata de un desplazamiento de cinco bits (2^5). Para evitar que el resultado genere desbordamiento, es decir, un resultado mayor que el tamaño del arreglo, es necesario usar el operador *mod* en cada iteración. La desventaja que ofrece es que tiende a ser lenta con llaves grandes; sin embargo, para superar esto algunos programadores optan por emplear solo partes de la llave como, por ejemplo, utilizan solo caracteres pares o solo los dos primeros caracteres de cada palabra, entre otros.

3. Resolución de colisiones

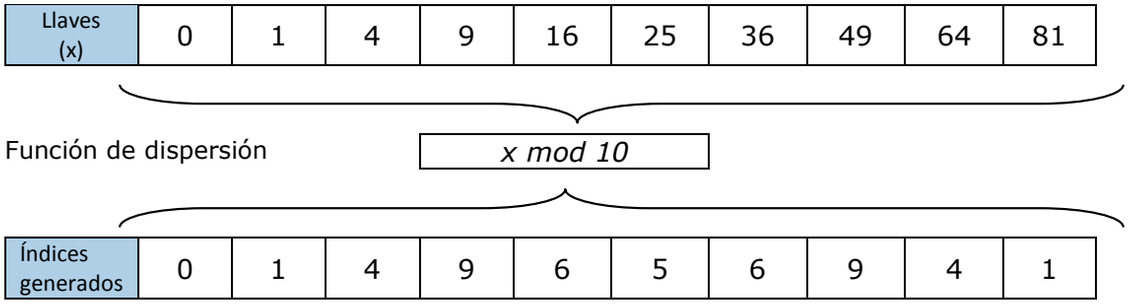
Una colisión ocurre cuando, al intentar insertar un nuevo dato al arreglo, la función de dispersión genera un índice en el cual ya existe un valor previo. Para superar este inconveniente existen dos métodos simples propuestos por Weiss (1995, p. 159):

- Dispersión abierta
- Dispersión cerrada

3.1. Dispersión abierta

Consiste en tener una lista de los elementos con el mismo valor de dispersión (o el mismo índice). Se le denomina también encadenamiento separado y gráficamente es como se muestra en la siguiente figura.

Cálculo de índices:



Asignación de datos:

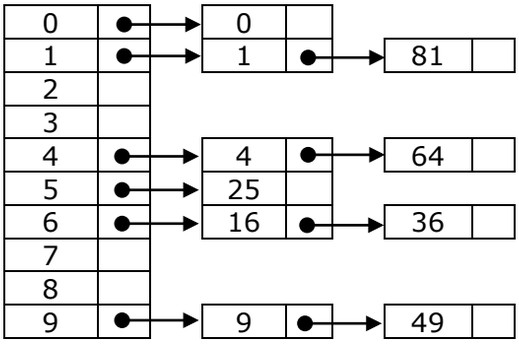


Figura 51. Ejemplo de resolución de colisiones mediante dispersión abierta.
Tomada de *Estructura de datos y algoritmos*, por Weiss, 1995.

Luego de aplicar la función de dispersión al conjunto de llaves, los resultados dan cuenta de un conjunto de colisiones como es el caso de las llaves "1" y "81", a las cuales se le asignó la casilla 1 del arreglo para almacenarse. Frente a ello, la solución que propone el método de la dispersión abierta es enlazar ambos elementos al *array* mediante punteros, en la posición que le corresponde, formando una cadena.

Para buscar un determinado elemento, simplemente se aplica la función de dispersión para determinar qué lista recorrer. A continuación, se recorre tal lista con cualquiera de los métodos ya revisados en los capítulos anteriores, devolviéndose la posición donde se encuentra.

El programa 1.1 muestra un ejemplo de implementación de este método.

```
#include <iostream>
using namespace std;

// Número de elementos del arreglo
const int NROELEMENTOS=10;

struct nodo
{
   int dato;
   struct nodo *sig;
};

void main()
{
   nodo *arreglo[NROELEMENTOS];
   nodo *q, *ant;
   nodo *nuevonodo;
   int valor;
   int pos;
   bool yaexiste=false;

   // Inicializar arreglo
   for (int i=0; i<NROELEMENTOS; i++)
        arreglo[i] = NULL;

   // Solicitar 5 datos
   for (int i=0; i<5; i++)
   {
        // Crear nuevo nodo e inicializar
sus valores
        nuevonodo = new(struct nodo);

        cout<<endl<<"Digite el valor a in-
sertar: ";
        cin>>nuevonodo->dato;
        nuevonodo->sig = NULL;

        // Calcular posición de inserción
(con función de dispersión)
        pos=nuevonodo->dato%NROELEMENTOS;
        cout<<endl<<"Índice generado:
"<<pos<<endl<<endl;
// Verificar si en la posición ya hay va-
lores registrados
        if (arreglo[pos]==NULL)
              arreglo[pos] = nuevonodo;
        else
        {
              //Verificar si el valor in-
gresado ya existe
              q = arreglo[pos];

              while (q!=NULL)
              {
                    if (q->dato==nuevono-
do->dato)
                          yaexiste =
true;
                    ant = q;
                    q = q->sig;
              }

              if (yaexiste)
                    cout<<"El elemento ya
existe en la lista"<<endl;
              else
                    ant->sig = nuevonodo;
              yaexiste = false;
        }
   }

   // Imprimir tabla
   for (int i=0; i<NROELEMENTOS; i++)
   {
        cout<<"P"<<i;
        if (arreglo[i]==NULL)
              cout<<" -> NULL"<<endl;
        else
        {
              q = arreglo[i];
              while (q!=NULL)
              {
                    cout<<" -> "<<q->da-
to;
                    q = q->sig;
              }
              cout<<endl;
        }
   }
system("Pause");
}
```

Programa 1.1

3.2. Dispersión cerrada

También llamado direccionamiento abierto. La solución que propone para las colisiones consiste en recorrer el arreglo hasta encontrar una celda vacía a través de la siguiente función:

$$d_i(x) = \left(dispersión(x) + f(i)\right) mod\ tamaño$$

En la fórmula anterior, la función *f* es la estrategia para la solución de colisiones, la cual puede ser de tres tipos:

TIPO DE EXPLORACIÓN	ESTRATEGIA	DESCRIPCIÓN	INCONVENIENTES
Exploración lineal	f(i) = i	Equivale a recorrer las celdas en secuencia (con vuelta al principio) en busca de una celda vacía. Requiere de arreglos relativamente grandes.	Se forman bloques de celdas ocupadas (agrupamiento primario).
Exploración cuadrática	**f(i) = i2**	Elimina el agrupamiento primario.	No garantiza encontrar una celda vacía cuando el arreglo está lleno a más de la mitad.
Dispersión doble	**f(i) = i2 * h2(x)**	h2(x) es una segunda función de dispersión. El resultado nunca debe ser cero; por ejemplo 99, mod 9.	Requiere que el tamaño del arreglo sea un número primo.

El programa 1.2 muestra la implementación de estos tres tipos de exploración como funciones de C++.

```
#include <iostream>
#include <math.h>
using namespace std;

const int MAXNROELEMENTOS = 17;
int arreglo[MAXNROELEMENTOS];

void inicializar_arreglo()
{
   for (int i=0; i<MAXNROELEMENTOS; i++)
        arreglo[i] = NULL;
}

int función_hash(int valor)
{
   return valor%MAXNROELEMENTOS;
}

int exploración_lineal(int pos)
{
   int pos_nueva, i=0;
   bool encontrado=false;

   // Recorrer el arreglo hasta encontrar un
casillero vacío
   pos_nueva = pos;
   while (!encontrado)
   {
        i++;
        pos_nueva=(pos_nueva+i)%MAXNROELEMEN-
TOS;
```

```
void menu_colisión()
{
   cout<<endl<<"Se ha producido una
colisión."<<endl;
   cout<<"¿Con qué método desea soluciona-
rlo?:"<<endl;
   cout<<"1. Exploración lineal"<<endl;
   cout<<"2. Exploración cuadrática"<<endl;
   cout<<"3. Dispersión"<<endl<<endl;
   cout<<"Opción?: ";
}

void main()
{
   locale::global(locale("spanish"));

   int pos=0;
   int opcion=0;

   inicializar_arreglo();

   // Solicitar 5 datos
   for (int i=0; i<5; i++)
   {
        // Solicitar dato al usuario
        int valor;
        cout<<endl<<"Digite el valor a inser-
tar: ";cin>>valor;
```

```
if (pos_nueva==pos) // Si se recorrió todos los elementos del arreglo sin éxito
                break;
        if (arreglo[pos_nueva]==NULL)
                {encontrado=true;break;}
   }
   if (encontrado)
        return pos_nueva;
   else
        return -1;
}
int exploración_cuadrática(int pos)
{
   int pos_nueva, i=0;
   bool encontrado = false;

   // Recorrer el arreglo hasta encontrar un casillero vacío
   while (!encontrado)
   {
        i++;
        pos_nueva=pos+int(pow(i,2))%MAXNROELEMENTOS;

        if (i==MAXNROELEMENTOS+1) // Detener si se hicieron los suficientes intentos
                break;

        if (arreglo[pos_nueva]==NULL)
                {encontrado=true;break;}
   }

   if (encontrado)
        return pos_nueva;
   else
        return -1;
}

int dispersión(int pos, int valor)
{
   int pos_nueva, i=0;
   bool encontrado=false;

   while (!encontrado)
   {
        i++;
        pos_nueva = pos + i + funcion_hash(valor);
        if (i==MAXNROELEMENTOS+1) // Detener si se hicieron los suficientes intentos
                break;

        if (arreglo[pos_nueva]==NULL)
                {encontrado=true; break;}
   }

   if (encontrado)
        return pos_nueva;
   else
        return -1;
}
```

```
// Calcular posición de inserción (Método de la división)
        pos = funcion_hash(valor);
        cout<<endl<<"Indice generado: "<<pos<<endl<<endl;

        // Verificar si se genera una colisión
        if (arreglo[pos]==NULL)
                arreglo[pos]=valor;
        else
        {
                menú_colisión();
                cin>>opcion;
                switch ( opción )
                {

                        case 1:
                                pos = exploración_lineal(pos);
                                break;

                        case 2:
                                pos= exploración_cuadrática(pos);
                                break;

                        case 3:
                                pos = dispersión(pos, valor);
                                break;

                        default:
                                cout<<"La opción no existe"<<endl;
                }

                if (pos==-1)
                        cout<<"No fue posible encontrar una casilla libre"<<endl;
                else
                        arreglo[pos]=valor;
        }
   }

   // Imprimir el arreglo
   cout<<endl<<"Datos"<<endl;
   cout<<"-----"<<endl;
   for (int i=0; i<MAXNROELEMENTOS; i++)
        cout<<"P"<<i<<": "<<arreglo[i]<<endl;

   system("Pause");
}
```

TEMA N.º 2: MODELO DE DATOS RELACIONAL

Todas las estructuras de datos revisadas a lo largo de este manual, como fueron las listas simples, las listas enlazadas, las matrices, etc., son medios de almacenamiento temporales; es decir, mantienen los datos en memoria mientras el programa que las genera está en ejecución. Esta característica resulta insuficiente para aplicaciones que deben trabajar en el entorno empresarial contemporáneo, el cual no solamente exige que estas sean precisas y veloces, sino también capaces de mantener un registro de cada transacción en un repositorio digital permanente: una base de datos.

El presente tema aborda uno de los tipos de estructuras que da soporte a una base de datos, el llamado modelo relacional.

1. Base de datos

"Es una colección de datos, lógicamente relacionados, que apoyan el acceso compartido de muchos usuarios y aplicaciones" (Loomis, 1991, p. 275).

Un sistema administrador de base datos (o DBMS por sus siglas en inglés) es un *software* que provee un conjunto de servicios para que tanto programadores como aplicaciones puedan acceder y utilizar las bases de datos sin enfrentarse a la complejidad que las subyace. La figura 52 muestra esta estructura.

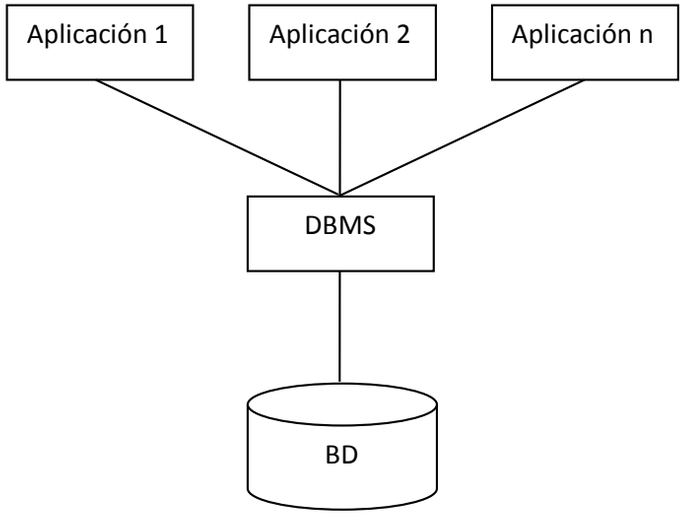


Figura 52. Estructura Aplicación-DBMS-BD
Tomada de *Estructura de datos y organización de archivos*, por Loomis (1991).

2. El modelo entidad-relación

La historia de la informática da cuenta de cuatro modelos que se han desarrollado como estructura de una base de datos: el de red, el jerárquico, el modelo relacional y el orientado a objetos. No obstante, en palabras de Silberschatz, Korth y Sudarshan (2002), el modelo relacional se ha establecido actualmente como el principal modelo de datos para las aplicaciones de procesamiento de datos. Su gran aceptación se debe a su simplicidad.

Este modelo es el producto final de un proceso de diseño que inicia con la elaboración del denominado "modelo entidad-relación", el cual a su vez se construye basado en la percepción de un mundo real que consiste en una colección de objetos básicos, denominados "entidades" y "relaciones". La figura 53 ilustra esta secuencia.

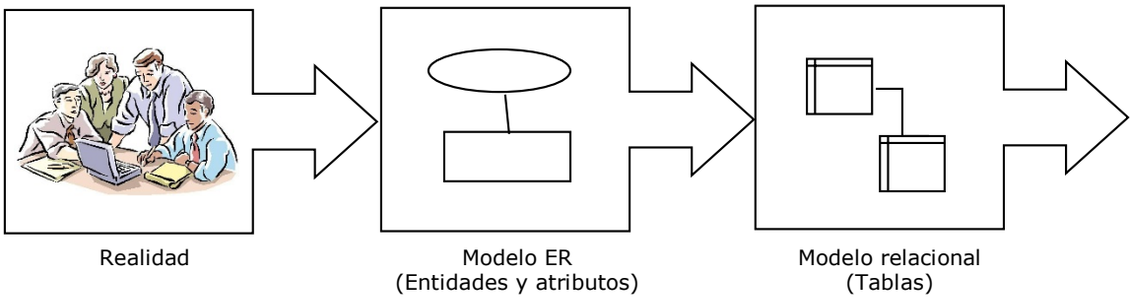


Figura 53. Etapas en el diseño del modelo relacional
Fuente: Elaboración propia

Silberschatz, et al. (2002, p. 19) consideran que "una entidad es una "cosa" u "objeto" del mundo real claramente distinguible de otros objetos". Por su parte, Ricardo (2004, p. 88) considera que es un "objeto que existe y se puede distinguir de otros objetos. Puede representar una persona, lugar, evento, objeto o concepto en el mundo real que se planea modelar en la base de datos". Entonces, una entidad puede ser real o abstracta y siempre estará vinculada a un contexto determinado.

Algunos ejemplos de entidades son los siguientes:

CONTEXTO: AGENCIA BANCARIA	CONTEXTO: CENTRO DE ESTUDIOS	CONTEXTO: RESTAURANT
Entidades: • Cliente • Préstamo • Cajero • Crédito • Analista • Cuenta (bancaria) • Etc.	Entidades: • Alumno • Docente • Asignatura • Horario • Etc.	Entidades: • Comensal • Platillo • Chef • Mozo • Cajero • Etc.

De acuerdo con el contexto, cada entidad se define con una serie de características o propiedades denominadas "atributos". A manera de ejemplo, a continuación, se enumeran posibles atributos para la entidad "Alumno":

- Número de DNI
- Nombres
- Apellidos
- Dirección domiciliaria
- Número de teléfono

La migración desde el modelo entidad-relación al modelo relacional se realiza siguiendo una serie de pautas que permiten convertir cada entidad detectada en una tabla y cada asociación entre ellas en una relación que garantice la integridad de los datos.

Por consiguiente, el modelo relacional presenta las siguientes características:

- Representa la estructura de una base de datos relacional.
- Se compone de dos elementos fundamentales: tablas y relaciones.
- Los datos se agrupan de acuerdo con la entidad a la que pertenecen y se asocian entre ellos gracias a las relaciones entre tablas.

Las bases de datos relacionales se han convertido en el mecanismo de almacenamiento de datos más común para las aplicaciones computacionales modernas.

3. Fundamentos del modelo entidad-relación

A continuación, se presenta un ejemplo muy simplificado sobre cómo crear este modelo mientras se van describiendo sus componentes.

Imagine que se necesitan registrar los datos de matrículas de un centro de estudios superior.

1.°. Se analiza el contexto y se identifican las entidades que intervienen en la actividad "matrícula".

- Alumno
- Asignatura

2.°. En función a las necesidades organizacionales, se anotan los atributos o características que se desean almacenar de cada entidad.

	ENTIDAD: ALUMNO	ENTIDAD: ASIGNATURA
ATRIBUTOS	• N.° de DNI • Nombres • Apellidos • Tipo sanguíneo • Observaciones	• Nombre • Categoría • N.° de créditos

3.°. Se procede a graficar las entidades y se anota el verbo (o acción) que las relaciona.

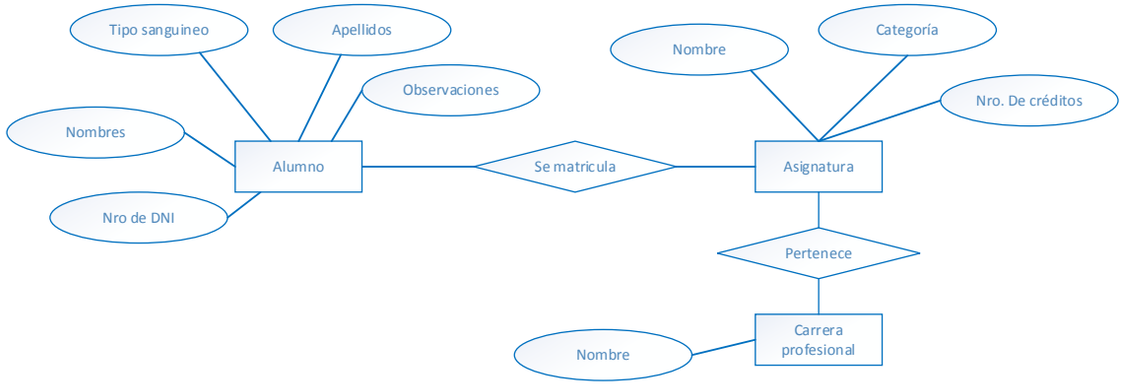


Figura 54. Ejemplo del modelo entidad relación

4.°. A continuación, se analiza la cardinalidad, es decir, qué cantidad de elementos de una entidad se relacionan solo con un elemento de la otra entidad. Esta cantidad se expresa como "uno" (1) o "muchos" (M).

En particular, refiriéndonos al ejemplo, las preguntas que permiten obtener la cardinalidad de cada par de entidades son las siguientes:

- ¿En cuántas asignaturas puede matricularse un alumno?

  Respuesta: Un alumno puede estar matriculado en muchas asignaturas.

- En una asignatura, ¿cuántos estudiantes pueden estar matriculados?

  Respuesta: En una asignatura pueden estar matriculados muchos alumnos.

- ¿A cuántas carreras profesionales pertenece una asignatura?

  Respuesta: Una asignatura pertenece a una sola carrera profesional.

- ¿Cuántas asignaturas puede tener una carrera profesional?

  Respuesta: Una carrera profesional tiene muchas asignaturas.

El gráfico quedaría de la siguiente manera:

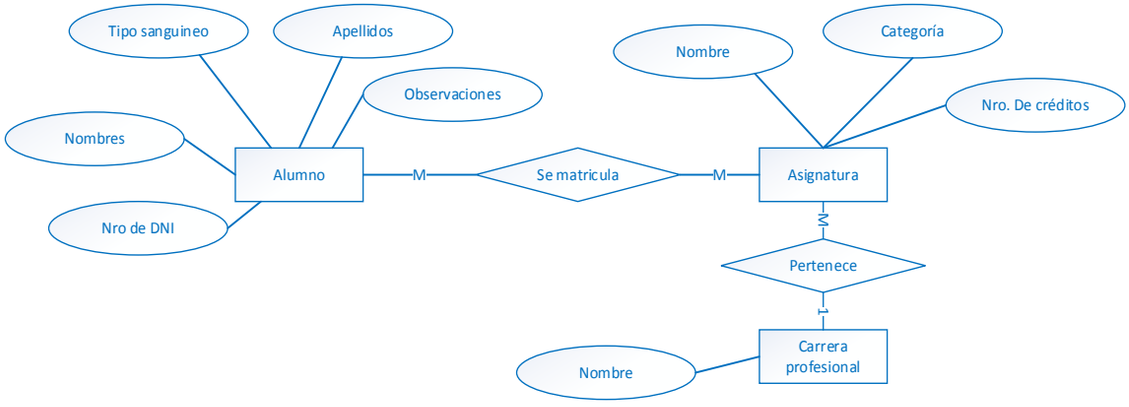


Figura 55. Ejemplo del modelo entidad-relación

Las entidades con cardinalidad "uno" se conocen como "entidades fuertes" y las que tienen cardinalidad "muchos", como "entidades débiles". Cuando una entidad fuerte se relaciona con una débil, se dice que entre ellas existe dependencia de existencia. Ricardo (2004, p. 106) expresa que una entidad es dependiente de la existencia de otra si no puede existir en la base de datos sin una instancia correspondiente de la otra entidad; por ejemplo, no tendría sentido almacenar datos acerca de órdenes de ventas (entidad débil) si no se tiene antes datos de clientes (entidad fuerte). En resumen:

- Las entidades fuertes existen por sí solas.
- Las entidades débiles necesitan de una entidad fuerte para que su existencia tenga sentido.

5.°. A continuación, es imprescindible identificar el atributo principal o llave para cada entidad. Los valores de este atributo jamás se repiten o quedan vacíos y se emplea precisamente para identificar a cada elemento que representa la entidad. Algunas consideraciones a tener en cuenta para elegir este atributo son las siguientes:

- Verificar si la entidad ya tiene un atributo con las características descritas (sin duplicados, sin vacíos). Por ejemplo, para la entidad alumno su atributo principal bien podría ser su número de DNI.
- Si ninguno cumple estas condiciones, se puede optar por lo siguiente:
  - Crear uno. Para el nombre de este nuevo atributo se suele usar el prefijo "id" (de "identificador") seguido del nombre de la entidad, por ejemplo, "IdAsignatura".
  - Agrupar varios atributos de tal forma que la concatenación de sus valores genere una clave.

Recuerde:

Una llave puede estar compuesta por uno o varios atributos.

Gráficamente, estos atributos especiales se representan subrayando su nombre.

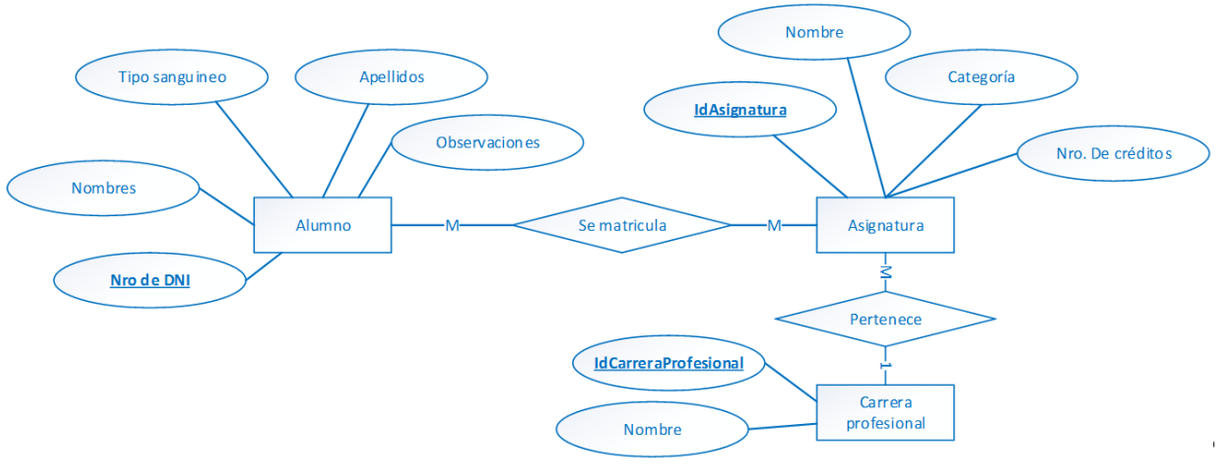


Figura 56. Ejemplo del modelo entidad-relación

4. Estructura de una base de datos relacional

La implementación física (en el DBMS) del modelo entidad-relación genera nuevos componentes y características que deben tenerse en consideración.

4.1. Tabla

En términos técnicos, una tabla es un almacén de datos con una estructura similar a la de un arreglo bidimensional. Gráficamente, se representa como se muestra en la figura 57.

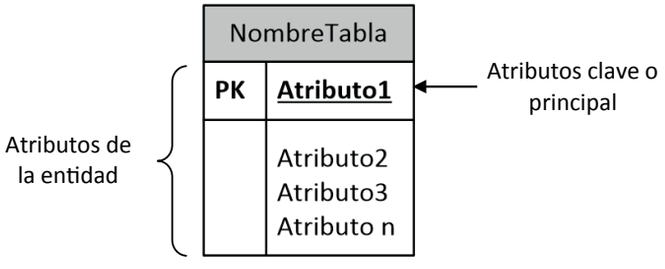


Figura 57. Partes de una tabla.
Fuente: Elaboración propia

Una vez finalizado el proceso de diseño de una tabla, esta deberá ser llenada con los datos que le corresponden. Para ello la mayoría de sistemas gestores de base de datos vienen equipados con dos vistas que facilitan este trabajo: la vista diseño y la vista de datos. Tal diferencia podrá notarlas en la figura 58.

tblAlumno	
PK	DNI
	Nombres Apellidos FechaNacimiento LugarResidencia

DNI	Nombres	Apellidos	Fecha Nacimiento	Lugar Residencia
42516191	Juan	Valdez Quiroga	01/07/1990	Huancayo
10203245	Pedro	Benegas Robles	15/08/1980	Lima
44553340	María	Roncal Pérez	31/12/1981	*NULL*
41918070	Lucía	Medrano López	*NULL*	Huancayo
10775655	Walter	Quijada Moyano	07/09/1989	Huánuco
...	...	...	...	...

Figura 58. Representación de la vista diseño (izquierda) y vista datos (derecha) de una tabla.
Fuente: Elaboración propia.

A cada fila de datos, le corresponde una y solo una entidad del mundo real, se le conoce también como "registro" o "tupla".

Para Silberschatz et al. (2002, p. 126) una tabla tiene las siguientes características:

- Cada celda de la tabla contiene solo un valor.
- Cada columna tiene un nombre distinto, que es el nombre del atributo que representa.
- Todos los valores en una columna provienen del mismo dominio, pues todos son valores del atributo correspondiente.
- Cada tupla o fila es distinta; no hay tuplas duplicadas.
- El orden de las tuplas es irrelevante.

4.2. Tipos de atributos

Una tabla puede estar compuesta de diversos tipos de atributos, destacando entre ellos:

- Atributo principal: O llave primaria (*primary key* en inglés); está destinado a identificar unívocamente a cada registro.
- Atributo secundario: O llave secundaria (*foreign key* en inglés); es un atributo propio de una entidad débil y sus valores los hereda de la llave primaria de donde proviene.
- Atributo común: Describe cualquier propiedad o característica de la entidad.
- Atributo candidato: Cualquier combinación de atributos que pueda considerarse una llave.

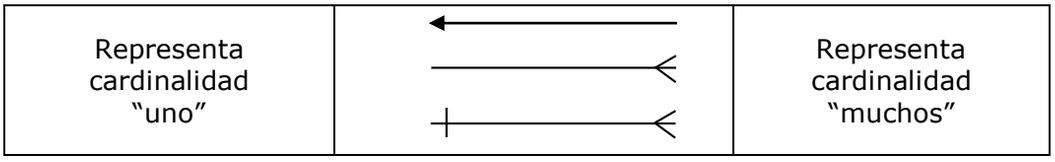
4.3. Relaciones entre tablas

A diferencia del modelo ER, en el modelo relacional solo existen relaciones del tipo "uno a muchos". Para su implementación es necesario que la entidad débil posea un atributo adicional con las mismas características que la llave principal de la entidad fuerte de la cual depende, por lo menos en lo que concierne a tipo y ancho de dato.

Por ejemplo, la implementación de la relación "asignatura-carrera profesional" (del modelo descrito anteriormente) queda como sigue:

Figura 59. Ejemplo de modelo relacional

Según la notación que se emplee, una relación entre dos tablas puede representarse, entre muchas otras, de las siguientes maneras:



a) Demostración que la relación "uno a muchos" del modelo ER realmente se refleja en la base de datos.

Observe las siguientes tablas:

tblAsignatura

IdAsignatura	Nombre
A002	Matemática
A005	Física I
A007	Fundamentos de electricidad
A009	Marketing
A010	Introducción a la administración

tblCarreraProfesional

IdCarreraProf	Nombre
C1	Ing. Eléctrica
C2	Administración

La cardinalidad indica que a cada carrera profesional le deben corresponder una o muchas asignaturas. Pero ¿cómo implementar ello?

Pues bien, aquí es donde entra a trabajar la llave foránea, aquel campo adicional de una entidad débil.

Ahora observe como este campo, simplemente copiando el valor de su llave principal "n" veces, permite reflejar lo siguiente:

- A la carrera profesional "Ing. Eléctrica" (uno) le corresponde las asignaturas "Matemática I", "Física I" y "Fundamentos de Electricidad" (muchos).
- A la carrera profesional "Administración" (uno) le corresponde las carreras "Marketing" e "Introducción a la Administración" (muchos).

tblAsignatura

IdAsignatura	Nombre	IdCarreraProf
A002	Matemática	C1
A005	Física I	C1
A007	Fundamentos de electricidad	C1
A009	Marketing	C2
A010	Introducción a la administración	C2

TblCarreraProfesional

IdCarreraProf	Nombre
C1	Ing. Eléctrica
C2	Administración

b) Demostración de la necesidad de mantener llaves primarias y secundarias.

Posiblemente usted, estimado estudiante, habrá reparado en que no es necesaria la tabla "tblCarreraProfesional", ya que al prescindir de ella el modelo aparentemente se hace más sencillo. Así:

TblAsignatura		
IdAsignatura	**Nombre**	**CarreraProf**
A002	Matemática	Ing. Eléctrica
A005	Física I	Ing. Eléctrica
A007	Fundamentos de electricidad	Ing. Eléctrica
A009	Marketing	Administración
A010	Introducción a la administración	Administración

Pues bien, en la teoría de base de datos existe un tema llamado "Normalización" el cual describe, entre muchas normas, la necesidad de mantener cada dato en su propia entidad; esto facilita en gran medida el mantenimiento de la base de datos y su adaptación a nuevas necesidades. En este curso no se profundizará en los temas de la normalización, pero se explicará con un ejemplo su importancia.

Imagine que la cada carrera profesional tuviese no uno sino más atributos, por ejemplo "Fecha de creación", "Dirección" y "Facultad", entonces la tblAsignatura tendría los siguientes datos:

TblAsignatura					
IdAsignatura	**Nombre**	**CarreraProf**	**FechaCreación**	**Director**	**Facultad**
A002	Matemática	Ing. Eléctrica	Ago-2001	Richard Robles	Ingeniería
A005	Física I	Ing. Eléctrica	Ago-2001	Richard Robles	Ingeniería
A007	Fundamentos de electricidad	Ing. Eléctrica	Ago-2001	Richard Robles	Ingeniería
A009	Marketing	Administración	Mar-2004	Mónica Munive	Ciencias de la empresa
A010	Introducción a la administración	Administración	Mar-2004	Mónica Munive	Ciencias de la empresa

¿Notó la importancia de mantener cada dato en su propia entidad? ¿Aún no? Primero, en esta tabla varios grupos de datos se repiten, lo que conlleva a utilizar espacio de disco innecesariamente. Segundo, ¿cómo actualizaría el nombre del Director de una carrera si este llegará a cambiar? ¡Exacto! Tendría que actualizar el nombre "n" veces. Ahora, ¿qué pasaría si alguna de las filas queda sin actualizar por un error? ¿Y si un director está a cargo de dos o más carreras? Complicado, ¿verdad?

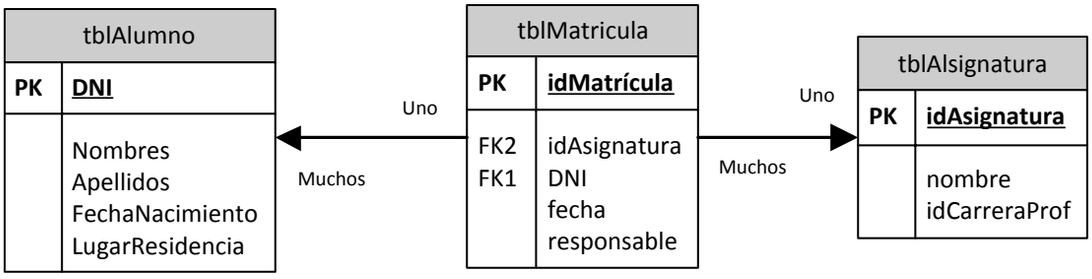
Todos estos inconvenientes se ven superados cuando la tabla Carrera Profesional se mantiene de manera independiente, tal como se le identificó en el modelo ER.

c) ¿Qué suceden con las relaciones de tipo "Muchos a muchos" identificadas en el modelo entidad-relación?

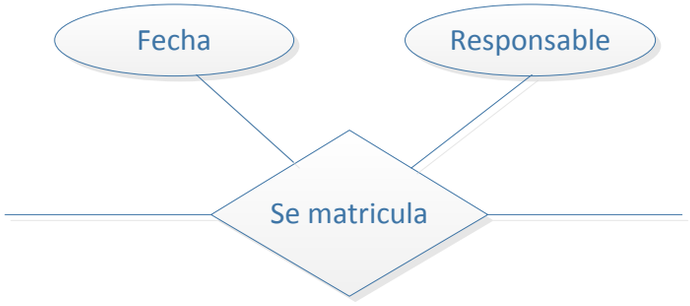
Ya que el modelo relacional solo permite la creación de relaciones de tipo "uno a muchos", las de tipo "muchos a muchos" que se generen en el ER deberán ser transformadas a través de las siguientes reglas:

- La relación, como tal, se convierte en una nueva tabla.
- La cardinalidad "muchos" se le asigna a esta nueva tabla, quedando las tablas que relacionaba con cardinalidad "uno".

Por ejemplo, la relación "alumno-asignatura" quedaría de la siguiente manera en el modelo relacional:



Por la propia naturaleza de la transformación, la tblMatrícula ya tiene dos campos (foráneos) por defecto: "idAsignatura" y "DNI"; no obstante, se pueden adicionar nuevos campos que permitan un mejor control de las matrículas, por ejemplo: una llave primaria, la fecha de la matrícula, el responsable de realizarla, etc. Es posible que estos campos adicionales, por llamarlos de alguna forma, sean ya detectados en el modelo ER por lo que suele representárseles de la siguiente forma:



5. Interactuación Aplicación–Base de datos

5.1. Creación de la base de datos y las tablas

Los sistemas gestores de datos actuales (como SQL Server) permiten manipular estos objetos, como su contenido, tanto de forma gráfica como a través de código, llamado *Transact-SQL*. El siguiente es un ejemplo de este código basado en las tablas tblCarreraProfesional y tblAsignatura tratadas anteriormente.

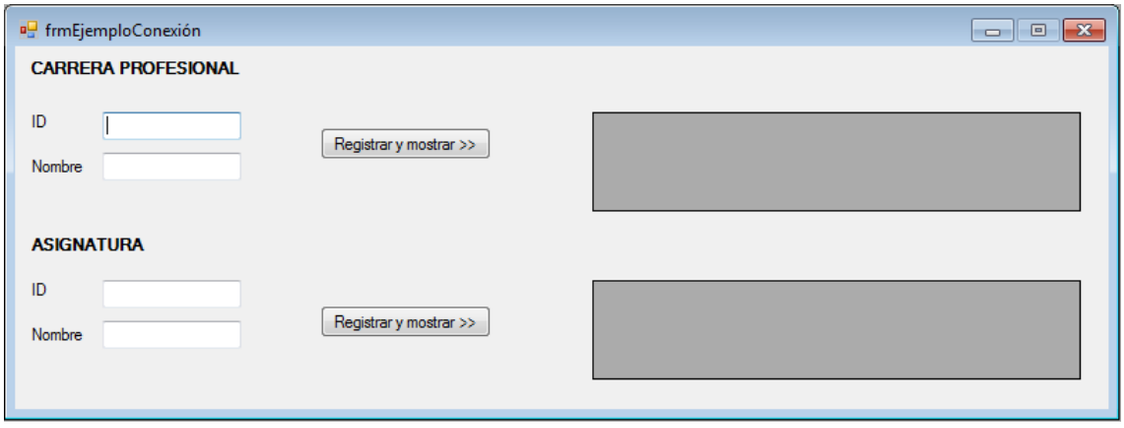
```
-- Creación de una base de datos llamada bdUniversidad
create database bdUniversidad
go

-- Apuntar hacia la base de datos recién creada
use bdUniversidad
go

-- Crear las tablas
create table tblCarreraProfesional (idCarreraProf varchar(2) Primary key,
                                    nombre varchar(50))
create table tblAsignatura (idAsignatura varchar(4) Primary key,
                                    nombre varchar(50),
                                    idCarreraProf varchar(2)
Foreign key (idCarreraProf) References tblCarreraProfesional (idCarreraProf))
go
```

Script Trasact-SQL Nro. 1

5.2. Escribir y leer datos en una tabla desde una aplicación

Muchos lenguajes de programación proveen de métodos de acceso a datos desde diversos motores (SQL Server, Access, DB2, Oracle, entre otros). Para el ejemplo siguiente, en el cual se muestra como escribir y leer datos en SQL Server, se empleó Visual Basic .Net 2012 por la simplicidad del código en esta tarea, lo cual facilitará su comprensión y posterior replicación e implementación en otros lenguajes.

Para probar el código, diseñe un formulario como el siguiente:



El proceso para guardar datos consiste en cuatro tareas:

a) Declarar las variables para establecer la conexión entre la aplicación el sistema gestor de base de datos:
   - SQLConection (contiene la cadena conexión que establece la dirección del servidor al que se conectará, el nombre de la base de datos, parámetros de seguridad entre otros).
   - SQLCommand (contiene la instrucción SQL que realizará la tarea ya sea de escritura o lectura de datos).

b) Capturar los nuevos valores digitados por el usuario.

c) Escribir las instrucciones de escritura para el comando.

d) Ejecutar el comando.

Ligeramente diferente es el proceso para recuperar la data que consiste en:

a) Declarar las variables para establecer la conexión entre la aplicación el sistema gestor de base de datos.
   - SQLConection
   - SQLCommand

b) Declarar los objetos que recibirán la información:
   - SQLDataReader (Para recibir los datos)
   - DataTable (Para mostrar al usuario a través de un control *gridview*)

c) Escribir las instrucciones de lectura para el comando.

d) Ejecutar el comando.

El programa 1.3 muestra el programa completo para registrar una carrera profesional.

```
' Importar la librería que contiene las clases SQLCONECTION y SQLCOMMAND
Imports System.Data.SqlClient

Public Class frmEjemploConexión

' 1. Declarar variables para la conexión
Dim conexion As New SqlConnection("SERVER=.;DATABASE=bdUniversidad;INTEGRATED
SECURITY=True")
Dim comando As SqlCommand

    Sub LeerDatos() ' Leer datos y mostrarlos en el GridView

        ' 2. Declarar objetos que recibirán la información
        Dim drCarrProfesional As SqlDataReader
        Dim dtCarrProfesional As New DataTable

        ' 3. Escribir instrucciones para el comando
comando = New SqlCommand("select * from tblCarreraProfesional", conexion)
comando.CommandType = CommandType.Text

        ' 4. Ejecutar el comando
        conexion.Open()
        drCarrProfesional = comando.ExecuteReader()
        dtCarrProfesional.Load(drCarrProfesional)
        dgCarreraProfesional.DataSource = dtCarrProfesional
        conexion.Close()

        ' Liberar la memoria y otros recursos de los objetos creados
        dtCarrProfesional.Dispose()
    End Sub

    Sub RegistrarDatos()
        ' 2. Capturar valores ingresados por el usuario
Dim idCarreraProf As String
        Dim NombreCarreraProf As String

idCarreraProf = txtIdCarrProf.Text
        NombreCarreraProf = txtNombreCarrProf.Text

        ' 3. Escribir instrucciones para el comando
        comando = New SqlCommand("insert into tblCarreraProfesional values ('"
& idCarreraProf & "','" & NombreCarreraProf & "')", conexion)
        comando.CommandType = CommandType.Text

        ' 4. Ejecutar el comando
conexion.Open()
        comando.ExecuteNonQuery()
        conexion.Close()

' Mensaje de confirmación
        MsgBox("El registro fue insertado en la base de datos")
End Sub

    Private Sub btnRegistrarCarrProf_Click(sender As Object, e As EventArgs)
Handles btnRegistrarCarrProf.Click
        RegistrarDatos()
        LeerDatos()
    End Sub
End Class
```

Programa 1.3

Lo invitamos estimado estudiante, a programar la escritura y lectura de datos de asignaturas siguiendo el mismo método expuesto.

¡Ten en cuenta!

Cuando registre datos en un modelo relacional, asegúrese de realizarlo primero en las tablas principales (de cardinalidad "uno") para luego pasar a las tablas secundarias (de cardinalidad "muchos").

TEMA N.º 3: ORGANIZACIÓN DE ARCHIVOS

Como se vio en el tema n.º 2, una base de datos compone un medio de almacenamiento estructurado respecto a un área o proceso de la organización. En este último tema se presenta otra alternativa de almacenamiento permanente, gracias al cual es posible guardar la información en entidades no estructuradas e independientes llamadas archivos.

1. Archivo

1.1. Definición

En principio, un archivo es un medio de almacenamiento permanente, como puede ser un documento generado por un procesador de texto, una fotografía digital capturada por una cámara, entre muchos otros.

Sin embargo, en organización de archivos solamente es posible trabajar con algunos de ellos, ya que los conceptos de registro y campo —vistos en el tema Nº 2— se mantienen. Es decir, los archivos a emplear deben conservar esta estructura de los datos. Aunque son múltiples las opciones, por su simplicidad se suelen emplear hojas de cálculo y también archivos de texto (txt).

Recuerda:

Un registro es un conjunto de campos en cada uno de los cuales se almacena un determinado valor.

Es preciso tomar en cuenta que mientras en una hoja de cálculo los registros estarán alojados en cada fila de la misma, y un campo se ubicará en una columna específica, en un archivo de texto estos elementos estarán divididos por saltos de línea y espacios respectivamente.

Finalmente, según Loomis (1995, p. 253) existen tres razones principales para estructurar un conjunto de datos en archivos:

a) Tener un medio de almacenamiento permanente e independiente de la ejecución de un programa.

b) Un archivo almacena grandes cantidades de información, en ocasiones, inclusive, mayores que la capacidad de la memoria principal.

c) La tercera razón es que las aplicaciones acceden a una porción pequeña de los datos en un momento determinado, y resulta ilógico almacenar toda esa información simultáneamente en la memoria principal.

1.2. Clasificación de los archivos

La autora citada continúa indicando la primera clasificación de los archivos, la cual se realiza por la función que desempeñan en un sistema de información:

- Archivo maestro: Representa una visión estática e histórica de algún aspecto del negocio en un momento dado. Por ejemplo, un archivo maestro de clientes, un archivo maestro de inventario, un archivo maestro de pagos realizados, etc.

- Archivo de transacciones: Contiene los datos para agregar, eliminar o modificar un registro en un archivo maestro. Cada registro de este archivo representa un evento o cambio.
- Archivo de reporte: Contiene datos ya formateados para su presentación ante el usuario. Un ejemplo es un archivo que se envía a una cola de impresión u otro que se utiliza para mostrarse en la pantalla de un terminal.
- Archivo de trabajo: Son archivos temporales que se emplean únicamente para pasar datos entre programas.
- Archivo de programa: Contiene instrucciones para procesar datos. Su contenido puede estar escrito en un lenguaje de alto o bajo nivel, o bien, ser producto de una compilación.
- Archivo de texto: Contiene datos alfanuméricos y gráficos generados a través de un programa editor de texto.

Otra forma de clasificar a los archivos es de acuerdo a la forma en que los programas acceden a ellos. Es decir:

- Archivo de entrada: Estos archivos son solamente de lectura y son utilizados por los programas para extraer datos. Por ejemplo, para una aplicación que convierte cantidades a diferentes denominaciones, un posible archivo de entrada sería uno que contenga la tabla de equivalencias entre las diferentes monedas mundiales.
- Archivo de salida: Es de solo escritura. Un claro ejemplo de este tipo es un archivo de reporte.
- Archivo de entrada/salida: Por su parte, este tipo de archivos son leídos y escritos durante la ejecución de un programa.

Es preciso aclarar que esta clasificación está determinada en mayor parte por el uso que le da una aplicación a un archivo, más que por el mismo archivo en sí. En otras palabras, un archivo que es salida de un programa puede ser entrada para otro.

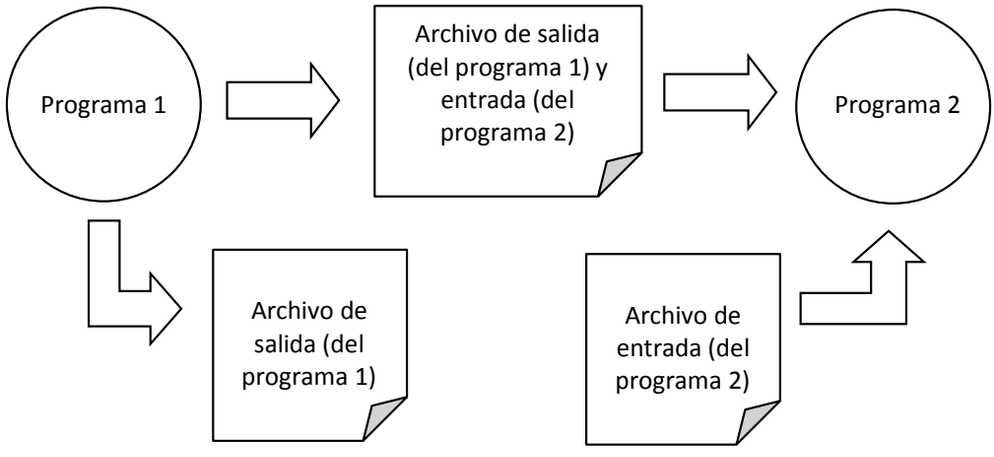


Figura 60. Representación de los archivos clasificados por su forma de acceso
Autor: Elaboración propia

2. Organización de archivos

Hace alusión al conjunto de técnicas utilizadas para representar y almacenar registros en los archivos, más que en estos últimos en sí mismos. Esta organización determina dos aspectos: la secuencia de los registros en el archivo y las operaciones necesarias para llegar a ellos.

La siguiente gráfica muestra las cuatro técnicas que conforman este capítulo con las respectivas diferencias entre ellas.

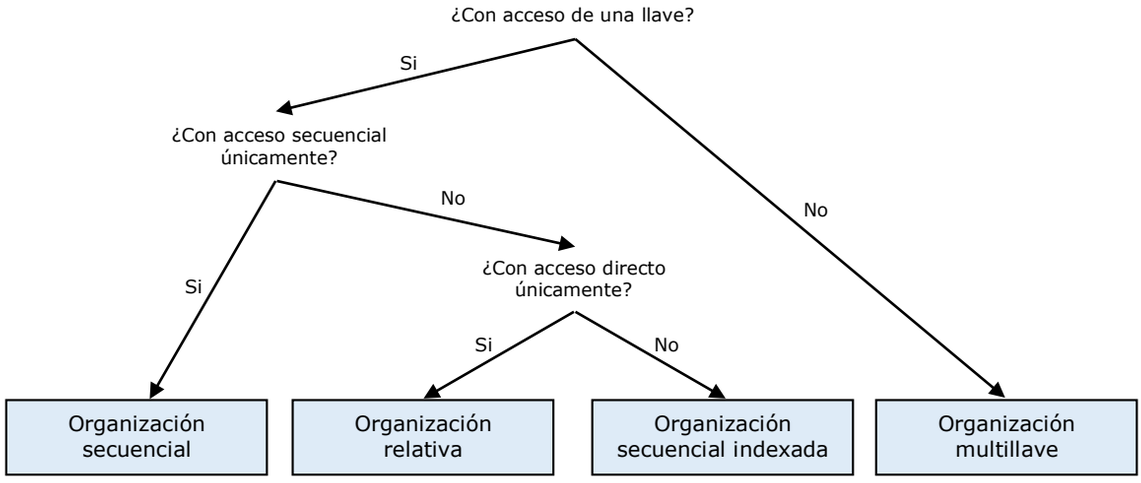


Figura 61. Distinción jerárquica entre las organizaciones fundamentales de archivos
Tomada de *Estructura de datos y organización de archivos,* Loomis, 1995.

2.1. Archivo secuencial

En un archivo secuencial los registros deben grabarse consecutivamente y leerse de la misma forma cuando el archivo se usa como entrada. La figura 62 ilustra lo descrito.

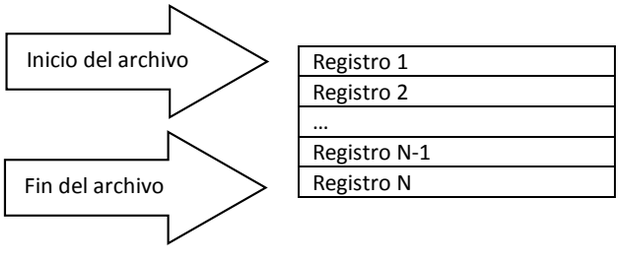


Figura 62. Estructura de un archivo secuencial
Tomada de *Estructura de datos y organización de archivos*, por Loomis,1995, p. 282.

Por su propia naturaleza los archivos secuenciales se utilizan con mayor frecuencia para procesamiento por lotes, ya que todos los registros grabados deben ser leídos al momento de su utilización.

Aunque su aplicación es basta, se suelen emplear en entornos de carga de datos *off-line*; es decir, los usuarios que generan la data no tienen acceso directo al sistema de información (por limitaciones cognitivas, técnicas, etc.), por lo que recurren a compartir los datos a través de archivos sencillos como los mencionados (*xls* o *txt*). Posteriormente, mediante aplicativos pre-programados, se copia su contenido a la base de datos central.

Ahora bien, un archivo organizado secuencialmente puede estar compuesto solo por un tipo de registro, o pueden agruparse varios según se requiera, siempre y cuando tengan un propósito funcional y común y tengan relación lógica; para esto es necesario que se adicione un campo que permita diferenciarlos. Como ejemplo considere que además de los datos personales de los empleados de una empresa, se requiere mantener un historial de las unidades u oficinas donde han prestado sus servicios. En este caso, el archivo ya requiere dos tipos de formato:

Formato de registro para datos personales de los empleados

Tipo de registro	Número de empleado	Nombre	Apellido	Sexo	Teléfono	Dirección domiciliaria
1						
1						

Formato de registro para los datos de las unidades u oficinas

Tipo de registro	Número de empleado	Nombre de la oficina/unidad	Fecha de ingreso	Fecha de salida
2				
2				

La principal ventaja que ofrece este tipo de archivos es su capacidad para acceder al "siguiente" registro de manera rápida.

En el programa 3.1 se muestra cómo crear un archivo y guardar en él registros conformados por los siguientes campos: nombre, apellido y edad de empleados; luego se recuperan y son mostrados en pantalla. Todo esto se realiza de forma secuencial.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

void main()
{
   // Declarar campos del registro
   char NombreEmp[20], ApellidoEmp[20];
   int EdadEmp;

   // ESCRIBIR ARCHIVO
   //   -----------------
   // Solicitar los datos de 5 empleados y reg-
istrarlos en un archivo txt
   for (int i=0; i<5; i++)
   {
        cout<<endl<<"Digite nombre del emplea-
do: ";cin>>NombreEmp;
        cout<<"Digite apellido del empleado:
";cin>>ApellidoEmp;
        cout<<"Digite edad del empleado:
";cin>>EdadEmp;

        // Declarar un objeto vinculado al ar-
chivo
        ofstream escritura("D:\RegistroEmplea-
do.txt", ios::out|ios::app);

        if (escritura.is_open()) // Abrir el
archivo y verificar que tal proceso

fue exitoso.
        {
                // Guardar datos en el archivo
                escritura<<NombreEmp<<" "<<Apel-
lidoEmp<<" "<<EdadEmp<<endl;
        }
Else
        cout<<"Error: El archivo no se pudo
abrir"<<endl;
}


// LEER ARCHIVO
// -----------

   ifstream lectura("D:\RegistroEmpleado.txt",
ios::out|ios::in);

   if (lectura.is_open())
   {
        lectura>>NombreEmp; // Leer primer reg-
istro

        while (!lectura.eof()) // Recorrer el
archivo
        {
                lectura>>ApellidoEmp;
                lectura>>EdadEmp;

                cout<<"Los datos del empleado
son: "<<endl;
                cout<<NombreEmp<<" "<<Apelli-
doEmp<<" "<<EdadEmp<<endl<<endl;

                lectura>>NombreEmp; // Leer
siguiente registro
        }
   }

   escritura.close();
   lectura.close();

   system("Pause");
}
```

Programa 3.1

Consideraciones técnicas sobre el código

En C++ la librería que permite acceder a archivos externos es "fstream" y trabaja creando objetos que apuntan a tales archivos. Por ejemplo, la instrucción "ofstream" ("o" inicial de *output*) abre un archivo de modo tal que se pueda escribir en él; por su parte "ifstream" ("i" inicial de *input*) abre un archivo en modo lectura.

La sintaxis de ambas instrucciones es similar:

[instrucción]<nombre de objeto> (<ruta de archivo>, *parámetros*)

Donde:

- <nombre de objeto>: Representa el nombre del objeto que se creará (como si de cualquier otro tipo de variable se tratara).
- <ruta de archivo>: Permite especificar la ruta, el nombre y la extensión del archivo externo al cual se accederá.
- *Parámetros:* Conjunto de valores para indicar el tipo de acceso al archivo. Por ejemplo:

ios::out|ios::in	Abre un archivo en modo lectura.
ios::out	Crea el archivo con la ruta y nombre especificados y lo habilita para escritura. Si el archivo ya existe, lo sobrescribe.
ios::out|ios::app	Crea el archivo con la ruta y nombre especificados y los habilita para escritura. Si el archivo ya existe, únicamente lo abre para agregar información (no borra lo anterior.)

Para guardar y recuperar información propiamente desde un archivo, se emplean los operadores "<<" y ">>" respectivamente, tal como se utilizan para imprimir e ingresar información a una aplicación con *cout* y *cin*.

2.2. Archivo relativo

Se emplea cuando existe la necesidad de acceder individual y directamente a cada registro del archivo. En estos tipos de archivo, existe una relación predecible entre la llave usada para identificar a cada registro y la localización del mismo.

De forma similar a como operan las tablas Hash (vistas al inicio de esta unidad), cuando se crea un archivo relativo debe establecerse una relación que será utilizada para obtener una dirección física a partir de un valor llamado "llave". Esta relación, llamada R, es una función de mapeo.

Entonces, al momento de grabar un registro en el archivo relativo, la función de mapeo R se usa para traducir el valor llave a una dirección, la cual indica dónde deberá almacenarse el registro; lo mismo si desea recuperarse un determinado valor.

Es preciso destacar que, a diferencia del archivo secuencial, en un archivo relativo se puede acceder directamente al registro deseado, sin necesidad de hacerlo de forma serial. Esta característica permite emplearlo normalmente en procesos interactivos.

Loomis (1995) describe el siguiente ejemplo: "Considere un sistema bancario en línea, para el cual los cajeros tienen terminales y acceso directos a las cuentas de clientes. Un archivo maestro simple podría tener registros con el siguiente formato" (p. 342):

Archivo: Cuenta

NÚMERO DE CUENTA	TIPO DE CUENTA	SALDO	FECHA DE ÚLTIMO RETIRO	FECHA ÚLTIMO DEPÓSITO

Suponga que el archivo "Cuenta" es de tipo relativo, con su llave "Número de cuenta". Por su parte, la información que registra el cajero por cada transacción tiene el siguiente formato:

Archivo: Transacción

NÚMERO DE CUENTA	TIPO DE TRANSACCIÓN	MONTO	FECHA

Cuando el cajero ha atendido una transacción, esta se registra en el archivo del mismo nombre. A continuación, esta información se emplea para recuperar o actualizar datos en el archivo "Cuenta":

- Si "Tipo de transacción" es "Información" entonces, a través de "número de cuenta", se ubica el registro correspondiente en "Cuenta" y se extraen los datos requeridos. Por ejemplo, Saldo y Fecha de último retiro.
- Si "Tipo de transacción" es "Retiro" o "Depósito", entonces, a través de "Número de cuenta", se ubica el registro correspondiente en "Cuenta" y se actualizan los campos respectivos.

Note que no es necesario acceder a todos los registros de "Cuenta", sino directamente al objetivo.

Finalmente, entonces la principal ventaja que ofrece un archivo relativo es la habilidad para acceder a un registro determinado de forma individual y directa. Eso quiere decir que la recuperación, inserción, modificación o hasta borrado de un determinado registro no afecta en nada a los demás.

El programa 3.2 muestra cómo escribir y leer en un archivo de tipo relativo.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

void main()
{
   // Variables para leer registros
   char NombreEmp[20], ApellidoEmp[20];
   int IdEmp, EdadEmp;

   // Variables para el nuevo registro
   char NombreEmpNuevo[20], ApellidoEmp-
Nuevo[20];
   int IdEmpNuevo, EdadEmpNuevo;

   // Variables para el programa
   char cadena[80]; int count=0, Ulti-
moIdEmp;



   // Declarar objetos vinculados a los
archivos
   ifstream lectura("D:\RegistroEm-
pleadoRelativo.txt", ios::out|ios::in);
   ofstream auxiliar("D:\RegistroAuxiliar.
txt", ios::out|ios::app);
```

```
// Si el nuevo Id debe ser insertado
después del último registro conocido
                    else
                    {
                           count = 0;
                           while(!lectura.
eof())
                           {
                                  count++;
                                  lectura.
getline(cadena,80);
                                  auxilia-
r<<cadena;
                                  if
(count<=UltimoIdEmp)

auxiliar<<endl;
                           }



                           for (int
i=1;i<IdEmpNuevo-UltimoIdEmp;i++)
                                  auxilia-
r<<endl;


                           auxilia-
r<<IdEmpNuevo<<" "<<NombreEmpNue-
vo<<""<<ApellidoEmpNuevo <<" "<<EdadEmp-
Nuevo<<endl;


                    }
             }

lectura.close();
auxiliar.close();
remove("D:\RegistroEmpleadoRelativo.
txt");
```

```
// ----------------
  // Ingresar los datos de 5 personas
  for (int i=0; i<5; i++)
  {
      // Solicitar datos al usuario
      cout<<"   REGISTRO DE NUEVO EMPLEADO   "<<endl;
      cout<<"-----------------------------"<<endl<<endl;
      cout<<"Digite Id: ";cin>>IdEmpNuevo;
      cout<<"Digite Nombre: ";cin>>NombreEmpNuevo;
      cout<<"Digite Apellido: ";cin>>ApellidoEmpNuevo;
      cout<<"Digite Edad: ";cin>>EdadEmpNuevo;

      if (lectura.is_open())
      {
            lectura>>UltimoIdEmp;
            while(!lectura.eof())
            {
                  lectura>>NombreEmp>>ApellidoEmp>>EdadEmp;
                  count++;
                  lectura>>UltimoIdEmp;
            }
            lectura.close();
// ESCRIBIR ARCHIVO

            // Reabrir el archivo (para que se posicione nuevamente en el primer registro)
            lectura.open("D:\RegistroEmpleadoRelativo.txt",ios::out|ios::in);

            // Si el archivo aún no tiene registros
            if (count==0)
            {
                  // Agregar saltos de línea hasta el nro de fila correspondiente y a continuación el nuevo registro
                  for (int i=1;i<IdEmpNuevo;i++)
                        auxiliar<<endl;
                  auxiliar<<IdEmpNuevo<<" "<<NombreEmpNuevo<<" " <<ApellidoEmpNuevo<<" "<<EdadEmpNuevo<<endl;
            }
            // Si el archivo ya tiene registros
            else
            {
rename("D:\RegistroAuxiliar.txt","D:\RegistroEmpleadoRelativo.txt");
      }
      else
      {
            cout<<"Error: El archivo no se pudo abrir"<<endl;
      }
  }

  // IMPRIMIR REGISTROS
  ifstream lectura1("D:\RegistroEmpleadoRelativo.txt", ios::out|ios::in);

  lectura1>>IdEmp;
  while (!lectura1.eof())
  {
      lectura1>>NombreEmp>>ApellidoEmp>>EdadEmp;
      cout<<endl<<"Id Empleado: "<<IdEmp<<endl;
      cout<<"Nombre: "<<NombreEmp<<endl;
      cout<<"Apellido: "<<ApellidoEmp<<endl;
      cout<<"Edad: "<<EdadEmp<<endl;
      lectura1>>IdEmp;
  }
  lectura1.close();

  //LEER ARCHIVO (En esta ocasión el usuario decide que registro imprimir)
  //------------
  lectura1.open("D:\RegistroEmpleadoRelativo.txt", ios::out|ios::in);
  int IdEmpBuscado;
  bool encontrado=false;

  cout<<endl<<"Digite el código de empleado que desea visualizar: ";cin>>IdEmpBuscado;

  lectura1>>IdEmp;
  while(!lectura1.eof())
  {
      lectura1>>NombreEmp>>ApellidoEmp>>EdadEmp;
      if (IdEmp==IdEmpBuscado)
      {
            encontrado = true;
            cout<<endl<<"Id Empleado: "<<IdEmp<<endl;
```

```
                    // Si el nuevo Id es
menor al Id del último empleado
                    if (IdEmpNuevo < Ul-
timoIdEmp)
                    {
                           count = 0;
                           while(!lectura.
eof())
                           {
                                  count++;

                                  if
(count==IdEmpNuevo)

auxiliar<<IdEmpNuevo<<" "<<NombreEmpNue-
vo<<" "<<ApellidoEmpNuevo <<" "<<EdadEmp-
Nuevo;
                                  else
                                  {

lectura.getline(cadena,80);

auxiliar<<cadena<<endl;
                                  }
                           }
                           lectura.
close();
                    }
```

```
                    cout<<"Nombre: "<<Nombre-
Emp<<endl;
                    cout<<"Apellido: "<<Apelli-
doEmp<<endl;
                    cout<<"Edad:
"<<EdadEmp<<endl;
             }
             lectura1>>IdEmp;
      }

      if (!encontrado)
             cout<<"El Id ingresado no ex-
iste"<<endl;

      system("Pause");
}
```

Programa 3.2

Para probar este código, es necesario que el archivo "RegistroEmpleadoRelativo.txt" ya exista en la ruta indicada. Como habrá notado, en este caso el proceso de inserción de un nuevo registro es más complejo; esto debido a que ahora puede ubicarse en cualquier parte del archivo y ya no solamente al final como sucedía con los archivos secuenciales. Otra consideración a tomar en cuenta es que en este ejemplo se emplean dos archivos: "RegistroEmpleadoRelativo" y "RegistroAuxiliar"; el motivo es que con C++ (y los métodos mostrados) no es posible modificar el contenido intermedio de un archivo; por ejemplo, agregar el registro número 5 cuando ya existen 1 y 10. Lo que se hace es trasladar línea a línea el contenido de "RegistroEmpleadoRelativo" a "Registro auxiliar" y, cuando se llega al Id del nuevo registro, se procede a reemplazar su contenido con los nuevos datos. Finalmente, el archivo original es eliminado y "Registro auxiliar" es renombrado tomando el lugar de aquel. La figura siguiente representa de mejor manera lo descrito.

Figura 63. Proceso de inserción de un nuevo registro en una posición intermedia de un archivo relativo

Finalmente, con este tipo de acceso a archivos el usuario elige a qué registro desea acceder.

2.3. Archivo secuencial indexado

Para imaginar el funcionamiento de este tipo de archivos, considere un diccionario con un conjunto de lengüetas que proporcionan un índice al conjunto de palabras organizadas de forma secuencial. Para encontrar una palabra en particular, por ejemplo, "Programación", normalmente no recorre todo el diccionario desde la primera página. En lugar de eso, selecciona una lengüeta en particular, en este caso la "P", para dirigir la búsqueda a la sección más próxima a la palabra en el conjunto de datos. Ubicado en esta sección, nuevamente, la búsqueda no empieza desde la primera palabra con "P", sino que se busca en los encabezados que indican la primera y última palabra de cada página. Ya en la página respectiva, recién empieza una búsqueda secuencial hasta la palabra deseada.

Para el funcionamiento de este tipo de estructuras ya son necesarios dos archivos: un archivo índice y el archivo de datos propiamente. Observe la siguiente figura:

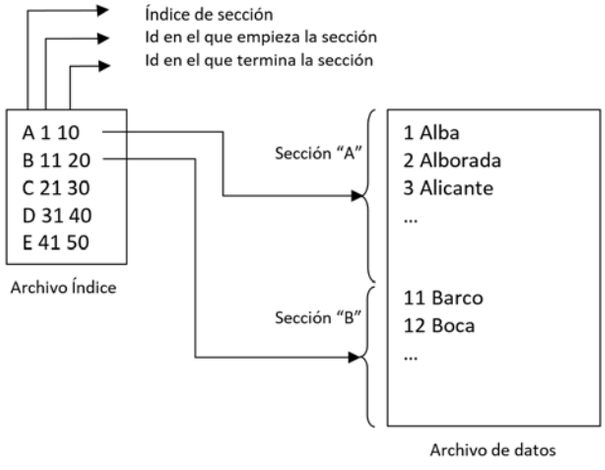


Figura 64. Ejemplo y partes de una estructura de archivo indexado
Fuente: Elaboración propia.

El programa 3.3 muestra cómo implementar archivos indexados a partir del ejemplo mostrado en la figura 64.

```
#include <iostream>
#include <fstream>
using namespace std;

void main()
{
   // Variables para el diccionario
   int IdPalabra;
   char palabra[20];

   // Variables para el indice
   char IdEntrada;
   int IdMin, IdMax;

   // Variables para el nuevo registro
   int IdNuevaPalabra;
   char PalabraNueva[20];
```

```
//Si el nuevo registro está después del
último registro existente
       else
       {
              count=0;
              while (!diccionario.eof())
              {
                     count++;
                     diccionario.getline(-
cadena,80);
                     dicAuxiliar<<cadena;
                     if (count<=ultimoId)
                            dicAuxilia-
r<<endl;
              }
```

```
   // Variables para el funcionamiento del
programa
   char IdEntradaB, cadena[100];
   int IdMinE=0, IdMaxE=0, count=0, ulti-
moId=0;
   bool empiezaConteo=false;


   ifstream indice ("D:\indice.txt",
ios::out|ios::in);
   ifstream diccionario ("D:\diccionario.
txt", ios::out|ios::in);
   ofstream dicAuxiliar ("D:\DicAuxiliar.
txt", ios::out|ios::out);


   cout<<"Digite la palabra a insertar:
";cin>>PalabraNueva;
   IdEntradaB = PalabraNueva[0]; // Cap-
turar el primer carácter (para posterior-
mente buscarlo en el archivo índice)



   // ESCRIBIR EN EL ARCHIVO
   // ----------------------


   // Determinar el rango (IdMinE e Id-
MaxE) donde se insertará la nueva palabra
   if (indice.is_open())
   {
        indice>>IdEntrada;
        while (!indice.eof())
        {
               indice>>IdMin>>IdMax;
               if (IdEntradaB == IdEntrada)
                     { IdMinE = IdMin; Id-
MaxE = IdMax; break; }
               indice>>IdEntrada;

}
        indice.close();
   }
   else
        cout<<"El archivo 'Indice.txt' no
se pudo abrir"<<endl;
```

```
               for (int i=1;i<IdMinE-ulti-
moId;i++)
                     dicAuxiliar<<endl;

               dicAuxiliar<<IdNuevaPal-
abra<<" "<<PalabraNueva<<endl;
        }
        diccionario.close();
        dicAuxiliar.close();
   }

   else // Si la sección ya existe
   {
        diccionario.open("D:\diccionario.
txt", ios::out|ios::in);

        count=0;
        while (!diccionario.eof())
        {
               count++;
               if (count==IdNuevaPalabra)
                     dicAuxiliar<<IdNue-
vaPalabra<<" "<<PalabraNueva;
               else
               {
                     diccionario.getline(-
cadena, 80);
                     dicAuxiliar<<cade-
na<<endl;
               }
        }
        diccionario.close();
        dicAuxiliar.close();
   }

   remove("D:\diccionario.txt");
   rename("D:\DicAuxiliar.txt", "D:\dic-
cionario.txt");

   // LEER EL ARCHIVO
   // ----------------------
   char palabraBuscada[20];
   cout<<endl<<"Digite la palabra a bus-
car: ";cin>>palabraBuscada;
```

```
en la sección que le corresponde
  if (diccionario.is_open())
  {
       count=0;
       diccionario>>IdPalabra;
       while (!diccionario.eof())
       {
              diccionario>>palabra;

              if (IdPalabra == IdMinE)
                     empiezaConteo=true;
              if (empiezaConteo)
                     if (IdPalabra<=Id-
MaxE)
                            count++;
                     else
                            empiezaConteo=-
false;

              diccionario>>IdPalabra;
       }
  }
  else
       cout<<"El archivo 'diccionario.txt'
no se pudo abrir"<<endl;
  diccionario.close();

  // Capturar Id de la nueva palabra
  IdNuevaPalabra = IdMinE + count;

  if (count==10) // Si la sección está
llena (En este ejemplo cada sección acep-
ta un máximo de 10 palabras)
       cout<<"La seccion está
llena"<<endl;

  // Insertar el nuevo dato

  else if (count==0) // Si la sección aún
no ha sido creada (No contiene palabra
alguna)
  {
       diccionario.open("D:\diccionario.
txt", ios::out|ios::in);
       // Determinar ultimo id registrado
       if (diccionario.is_open())
       {
              diccionario>>ultimoId;
              while(!diccionario.eof())
              {
```

```
  bool encontrado=false;
  int idencontrado=0;

  // Capturar el índice de la palabra
(primera letra)
  IdEntradaB = palabraBuscada[0];

  // Determinar el rango (IdMinE e Id-
MaxE) donde se buscará la palabra
  ifstream indice1("D:\indice.txt",
ios::out|ios::in);

  if (indice1.is_open())
  {
       indice1>>IdEntrada;
       while (!indice1.eof())
       {
              indice1>>IdMin>>IdMax;
              if (IdEntradaB == IdEntrada)
                     { IdMinE = IdMin; Id-
MaxE = IdMax; break; }
              indice1>>IdEntrada;
       }
       indice1.close();
  }
  else
       cout<<"El archivo 'Indice.txt' no
se pudo abrir"<<endl;

  // Buscar la palabra
  ifstream diccionario1 ("D:\diccionario.
txt", ios::out|ios::in);

  if (diccionario1.is_open())
  {
       diccionario1>>IdPalabra;
       while (!diccionario1.eof())
       {
              diccionario1>>palabra;
              // Buscar solo en la sección
correspondiente
              if(IdPalabra>=IdMinE && Id-
Palabra<=IdMaxE)
              {
                     if (strcmp(pal-
abra,palabraBuscada)==0)
                     {    idencontrado
=IdPalabra; encontrado=true;            }
              }
              diccionario1>>IdPalabra;
       }
```

```
// Determinar el ID de la nueva palabra
                        diccionario>>palabra;
                    count++;
                    diccionario>>ultimoId;
             }
       }
       diccionario.close();
// Si el nuevo registro está antes del último registro existente
       diccionario.open("D:\diccionario.txt", ios::out|ios::in);
       if (IdMinE<ultimoId)
       {
             count=0;
             while(!diccionario.eof())
             {
                    count++;

                    if (count==IdMinE)
                          dicAuxiliar<<IdNuevaPalabra<< " "<<PalabraNueva;
                    else
                    {
                          diccionario.getline(cadena,80);
                          dicAuxiliar<<cadena<<endl;
                    }
             }
       }
       if (encontrado)
       {   cout<<"Palabra encontrada en la posición ["<<idencontrado<<"]"<<endl;   }
       else
       {   cout<<"No se encontró ninguna coincidencia"<<endl;   }
   }
   else
       cout<<"No se pudo abrir el archivo 'Diccionario.txt'"<<endl;

   system("Pause");
}
```

Programa 3.3

Los métodos empleados para la inserción de nuevos registros son similares a los de los archivos secuenciales, con la excepción que primero se determina el rango (IdMinE e IdMaxE) donde le corresponde grabarse.

El código para recuperar un registro específico sí es totalmente diferente, ya que la búsqueda debe centrarse únicamente en la sección que le concierne. Si bien la instrucción while (!diccionario1.eof())recorre todo el archivo, solo compara aquellas palabras que se encuentran entre IdMinE e IdMaxE.

2.4. Archivo multi-llave

"Existen numerosas técnicas que han sido utilizadas para implantar archivos multillave. La mayoría de estos métodos están basados en la construcción de índices para proporcionar acceso directo mediante el valor de las llaves" (Loomis, 1995, p. 446). Gracias a esta ventaja, es posible elaborar listados de datos contenidos en el archivo de manera rápida.

Estructuralmente son similares a los archivos indexados, ya que trabajan con dos ficheros: uno que contiene los índices y un segundo para los datos propiamente dichos.

Los métodos que se verán a continuación son dos:

a) La organización mediante inversión

b) La organización multilista

2.4.1. **Organización de archivos mediante inversión**

En este caso, el "archivo índice" contiene en cada fila el valor de una llave y a continuación una lista del conjunto de claves donde se puede encontrar esa llave en el "archivo de datos". En términos gráficos, cada registro del "archivo índice" apunta a uno o varios registros del "archivo de datos". Observe la siguiente figura:

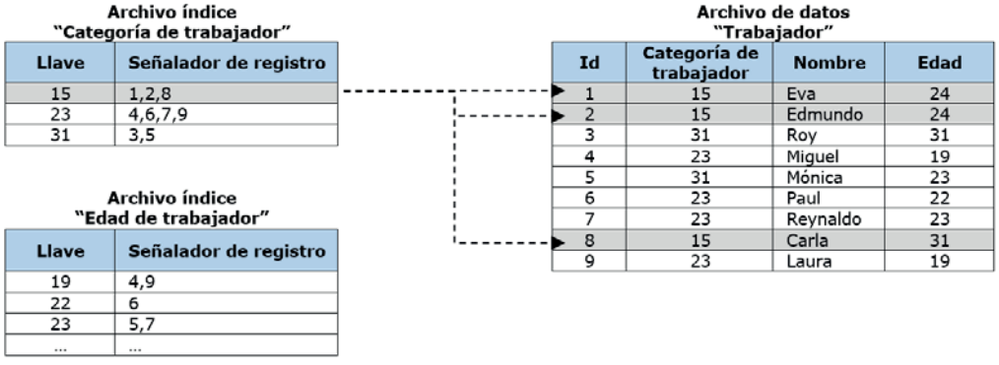


Figura 65. Ejemplo de una estructura de archivos mediante inversión.
Elaboración propia.

¡Ten en cuenta!

Cuando el "archivo índice" apunta a los registros del "archivo de datos" donde coincide la llave, se dice que el archivo de datos está "invertido sobre esa llave".

Puede existir un archivo índice por cada columna a la cual se desee tener acceso directamente. Es debido a esta cualidad que se le conoce a este tipo de organización como multillave.

Un algoritmo propuesto para la lectura de los registros se muestra en las siguientes líneas:

```
void listar_por_categoria(int categoría)
{
  cargar_archivo("índice");
  Ids_registros[] = recuperar_ids(categoría);

  cargar_archivo("datos");

  indice = 0;

  mientras(índice != NULL)
  {
     imprimir_registro_Nro(Ids_registros[índice]);
     índice = siguiente_indice(índice);
  }
}
```

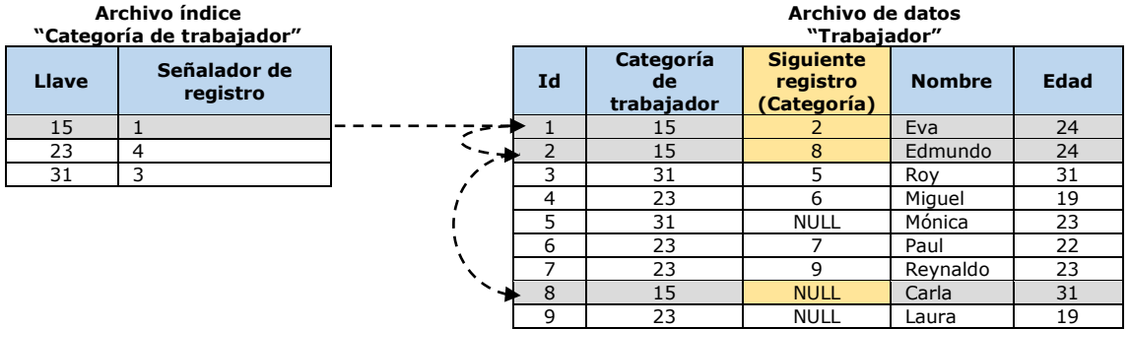
Respecto a su implementación en C++, como podrá notar, estimado estudiante, las técnicas son las mismas que las empleadas para los archivos indexados, por lo que lo invitamos a implementar el ejemplo de la figura 65 y a verificar su funcionamiento.

2.4.2. Organización de archivos multilista

Por su parte, en este tipo de organización, cada fila del "archivo índice" apunta solo al primer registro del "archivo de datos" donde aparece la llave; este tiene un puntador al segundo y así sucesivamente. En palabras de Loomis (1995), las diferencias saltan a la vista:

> La organización multilista difiere de la de inversión en que, mientras que la entrada en el índice de inversión para un valor de llave tiene un apuntador a cada registro de datos (…), la entrada en el índice multilista (…) tiene un solo apuntador al primer registro de datos.
>
> p. 455

La figura siguiente intenta esclarecer este último tipo de organización continuando con el ejemplo ya tratado.



Archivo índice "Categoría de trabajador"

Llave	Señalador de registro
15	1
23	4
31	3

Archivo de datos "Trabajador"

Id	Categoría de trabajador	Siguiente registro (Categoría)	Nombre	Edad
1	15	2	Eva	24
2	15	8	Edmundo	24
3	31	5	Roy	31
4	23	6	Miguel	19
5	31	NULL	Mónica	23
6	23	7	Paul	22
7	23	9	Reynaldo	23
8	15	NULL	Carla	31
9	23	NULL	Laura	19

Figura 66. Archivo índice y archivo de datos trabajador

Claro está que la organización multilista requiere de una columna adicional por cada tipo de llave que se desee emplear.

Un algoritmo que permite recuperar toda la información para una determinada llave es el siguiente:

```
void listar_por_categoría(int categoría)
{
   cargar_archivo("índice");
   Id_primer_registro = Primer_Id(categoría);

   cargar_archivo("datos");

   imprimir_registro_Nro(Id_primer_registro);
   leer_siguiente_registro_de(Id_primer_registro);
   mientras(siguiente_registro != NULL)
   {
      imprimir_registro_Nro(siguiente_registro);
      siguiente_registro = leer_siguiente_registro_de(siguiente_registro);
   }
}
```

Lectura seleccionada n.º 04

Sistemas de bases de datos frente a sistemas de archivos

Leer el apartado 1.2. "Sistemas de bases de datos frente a sistemas de archivos" (pp. 2-3).

Silberschatz, A., Korth, H. & Sudarshan S. (2002). *Fundamentos de bases de datos*. Madrid, España: McGRAW-HILL/Interamericana de España, S. A. U. Disponible en https://unefazuliasistemas.files.wordpress.com/2011/04/fundamentos-de-bases-de-datos-silberschatz-korth-sudarshan.pdf

Actividad N.º 4

Foro de discusión sobre tablas Hash

Instrucciones:

a) Implemente una aplicación que ejecute los tres métodos para evitar colisiones: exploración lineal, exploración cuadrática y dispersión doble.

b) Pruebe cada método con un arreglo relativamente grande (mayor a 100), llenándolo por lo menos a la mitad de su capacidad. Anote el número de colisiones presentadas.

c) Ingrese al foro y participe respondiendo las siguientes preguntas:

¿Cuál de los tres métodos distribuye de forma más uniforme los datos?

¿A qué se debe esta diferencia? Explique.

Actividad N.º 5

Foro de discusión sobre modelo relacional y gestión de archivos

Instrucciones:

a) Repase la lectura seleccionada n.º 1.

b) Ingrese al foro y participe respondiendo las siguientes preguntas:

Mencione 3 ventajas y 3 desventajas de emplear el modelo relacional como almacén de datos.

Mencione 3 ventajas y 3 desventajas de emplear la gestión de archivos como método de almacén de datos.

c) Mencione un caso real en el que sea más adecuado emplear el modelo relacional, y otro caso, en el que sea más adecuado emplear la gestión de archivos.

Glosario de la Unidad IV

C

Cardinalidad, razón de. Expresa el número de entidades a las que otra entidad puede estar asociada vía un conjunto de relaciones. (Silberschatz, Abraham & Korth, Henry, Sudarshan, S. Fundamentos de base de datos).

E

Entidad. Una entidad es una cosa u objeto en el mundo real que es distinguible de todos los demás objetos. (Silberschatz, Abraham & Korth, Henry, Sudarshan, S. Fundamentos de base de datos)

F

Función. Relación entre dos conjuntos que asigna a cada elemento del primero un elemento del segundo o ninguno. (Diccionario de la Real Academia Española)

I

Interactuar. Actuar recíprocamente. (Diccionario de la Real Academia Española)

M

Modelo. Esquema teórico, generalmente en forma matemática, de un sistema o de una realidad compleja, como la evolución económica de un país, que se elabora para facilitar su comprensión y el estudio de su comportamiento. (Diccionario de la Real Academia Española).

O

Organizar. Poner algo en orden. Diccionario de la Real Academia Española

R

Relación. Es una asociación entre diferentes entidades. (Silberschatz, Abraham & Korth, Henry, Sudarshan, S. Fundamentos de base de datos).

Bibliografía de la Unidad IV

Aho, A., Ullman, J. & Hopcroft, J. (1983). *Data structures and algorithms*. Nueva Delhi: Pearson Education.

Cairo, O. y Guardati, S. (2010). *Estructuras de datos* (3.ª ed.). México: Editorial McGraw Hill.

Charles, S. (2009). *Python para informáticos: Explorando la información*.

Cruz, D. (s.f.). *Apuntes de la asignatura: Estructura de datos*. México: Tecnológico de Estudios Superiores de Ecatepec. Disponible en http://myslide.es/documents/manual-estructura-de-datos.html

Frittelli, V., Steffolani, F., Harach, J., Serrano, D., Fernández, J., Scarafia, D., Teicher, R., Bett, G., Tartabini, M. & Strub, A. (s.f). *Archivos Hash: Implementación y aplicaciones*. Universidad Tecnológica Nacional, Facultad Regional Córdoba. Disponible en http://conaiisi.unsl.edu.ar/2013/121-493-1-DR.pdf

Joyanes, L. y Zahonero, I. (2003). *Programación en C. Metodología, algoritmos y estructura de datos*. Málaga, España: McGraw Hill.

Loomis, M. E. (1995). *Estructura de datos y organización de archivos*. México: Prentice-Hall Hispanoamericana.

López, J. C. (2002). *Tablas Hash*. Universidad del Valle. Colombia. Disponible en http://ocw.univalle.edu.co/ocw/ingenieria-de-sistemas-telematica-y-afines/fundamentos-de-analisis-y-diseno-de-algoritmos/material/hash.pdf

Mehta, D. P., & Sahni, S. (2005). *Handbook of Data Structures and Applications.* Florida, U.S.A.: Editorial Chapman & Hall/CRC. Disponibe en http://www.e-reading.club/bookreader.php/138822/Mehta_-_Handbook_of_Data_Structures_and_Applications.pdf

Pérez, J. (2004). *Matemáticas*. Instituto Nacional de Tecnologías Educativas y de Formación del Profesorado. España. Disponible en http://sauce.pntic.mec.es/~jpeo0002/Archivos/PDF/T06.pdf

Sahni, S. (2005). *Data Structures, Algorithms, and Applications in C++* (2.ª ed.) Hyderabad, India: Universities Press.

Silberschatz, A., Korth, H. & Sudarshan S. (2002). *Fundamentos de bases de datos*. Madrid, España: McGRAW-HILL/Interamericana de España, S. A. U. Disponible en https://unefazuliasistemas.files.wordpress.com/2011/04/fundamentos-de-bases-de-datos-silberschatz-korth-sudarshan.pdf

Weiss, M. (1995). *Estructura de datos y algoritmos*. EE. UU.: Addison-Wesley Iberoamericana.

Autoevaluación N.º 4

1. En una tabla hash, el tiempo de recuperación de la información es

   a. Constante
   b. Variable
   c. Inmediata
   d. Superior a cualquier otro método

2. Conjunto de operaciones matemáticas que permiten escribir/leer datos en una tabla Hash.

   a. Fórmula polinómica
   b. Función de dispersión
   c. Fórmula Hash
   d. Función de cálculo

3. No es una característica del método de dispersión abierta.

   a. Opera con un arreglo de punteros.
   b. Los elementos que colisionan forman una lista enlazada.
   c. Tiene límite máximo permitido de elementos.
   d. Para buscar un elemento se recorre la lista correspondiente.

4. El modelo entidad-relación:

   a. Es la estructura de una base de datos
   b. Es un repositorio de datos volátil
   c. Se diseña directamente desde la realidad
   d. Está conformado por tablas y relaciones

5. Es un ejemplo de entidad y atributo (en ese orden)

   a. Nombre – Alumno
   b. Vehículo–camión
   c. Teclado–Componente
   d. Persona – nombre

6. No es una cualidad de un atributo principal en una base de datos:

   a. Representa unívocamente a cada registro.

   b. Sus valores no se repiten.

   c. Sus valores no pueden ser nulos.

   d. Sus valores siempre son lógicos

7. El modelo físico de base de datos es

   a. Una consecuencia del modelo entidad-relación

   b. Un diagrama hecho a base de rombos, elipse y rectángulos

   c. Un conjunto de entidades

   d. Un medio de comunicación entre la aplicación y los datos

8. Son tipos de archivos según la función que cumplen en un sistema de información:

   i. Archivo de reporte

   ii. Archivo de trabajo

   iii. Archivo de programa

   iv. Archivo maestro

   a. i

   b. i y iii

   c. ii y iv

   d. i y iv

9. Correlacione cada tipo de archivo con su definición o características particulares:

A	Archivo secuencial		Se emplea cuando se necesita acceder individual y directamente a un registro.
B	Archivo relativo		Los registros se graban y recuperan uno a continuación de otro.
C	Archivo secuencial indexado		Permite acceder a los registros a través de diferentes campos.
D	Archivo multillave		Opera de forma similar a un diccionario de datos.

   a. BADC

   b. ABDC

   c. DBAC

   d. CABD

10. Es una desventaja de emplear archivos en vez de modelo relacional:

    a. La gran cantidad de espacio de ocupan en disco.

    b. No se puede ver la información contenida directamente y con facilidad.

    c. La integridad de datos.

    d. Es imposible eliminar un registro relacionado a un archivo principal.

ANEXO: SOLUCIONARIO DE LAS AUTOEVALUACIONES

Respuestas de la autoevaluación de la Unidad I

NÚMERO	RESPUESTA
1	B
2	A
3	c
4	b
5	b
6	c
7	a
8	b
9	c
10	C

Respuestas de la autoevaluación de la Unidad II

NÚMERO	RESPUESTA
1	D
2	C
3	B
4	A
5	A
6	A
7	A
8	B
9	D
10	C

Respuestas de la autoevaluación de la Unidad III

PREGUNTA	ALTERNATIVA
1	a
2	c
3	c
4	a
5	d
6	d
7	b
8	b
9	c
10	b

Respuestas de la autoevaluación de la Unidad IV

PREGUNTA	ALTERNATIVA
1	a
2	b
3	c
4	a
5	d
6	d
7	a
8	d
9	a
10	c

Universidad Continental

MANUAL AUTOFORMATIVO INTERACTIVO

Este manual autoformativo es el material didáctico más importante de la presente asignatura. Elaborado por el docente, orienta y facilita el auto aprendizaje de los contenidos y el desarrollo de las actividades propuestas en el sílabo.

Los demás recursos educativos del aula virtual complementan y se derivan del manual. Los contenidos multimedia ofrecidos utilizando videos, presentaciones, audios, clases interactivas, se corresponden a los contenidos del presente manual. La educación a distancia en entornos virtuales te permite estudiar desde el lugar donde te encuentres y a la hora que más te convenga. Basta conectarse al Internet, ingresar al campus virtual donde encontrarás todos tus servicios: aulas, videoclases, presentaciones animadas, biblioteca de recursos, muro y las tareas, siempre acompañado de tus docentes y amigos.

El modelo educativo de la Universidad Continental a distancia es innovador, interactivo e integral, conjugando el conocimiento, la investigación y la innovación. Su estructura, organización y funcionamiento están de acuerdo a los estándares internacionales. Es innovador, porque desarrolla las mejores prácticas del e-learning universitario global; interactivo, porque proporciona recursos para la comunicación y colaboración síncrona y asíncrona con docentes y estudiantes; e integral, pues articula contenidos, medios y recursos para el aprendizaje permanente y en espacios flexibles. Ahora podrás estar en la universidad en tiempo real sin ir a la universidad.