UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN
CENTRO DE EDUCACIÓN A DISTANCIA Y GESTIÓN DEL CONOCIMIENTO

APUNTE ELECTRÓNICO

Programación (Estructura de datos)

Licenciatura en Informática

Plan 2012 2016 actualizado

COLABORADORES

DIRECTOR DE LA FCA
Mtro. Tomás Humberto Rubio Pérez

SECRETARIO GENERAL
Dr. Armando Tomé González

– – – –

COORDINACIÓN GENERAL
Mtra. Gabriela Montero Montiel
Jefe del Centro de Educación a Distancia y
Gestión del conocimiento

COORDINACIÓN ACADÉMICA
Mtro. Francisco Hernández Mendoza
FCA-UNAM

– – –

AUTOR
Mtro. German Ignacio Cervantes González

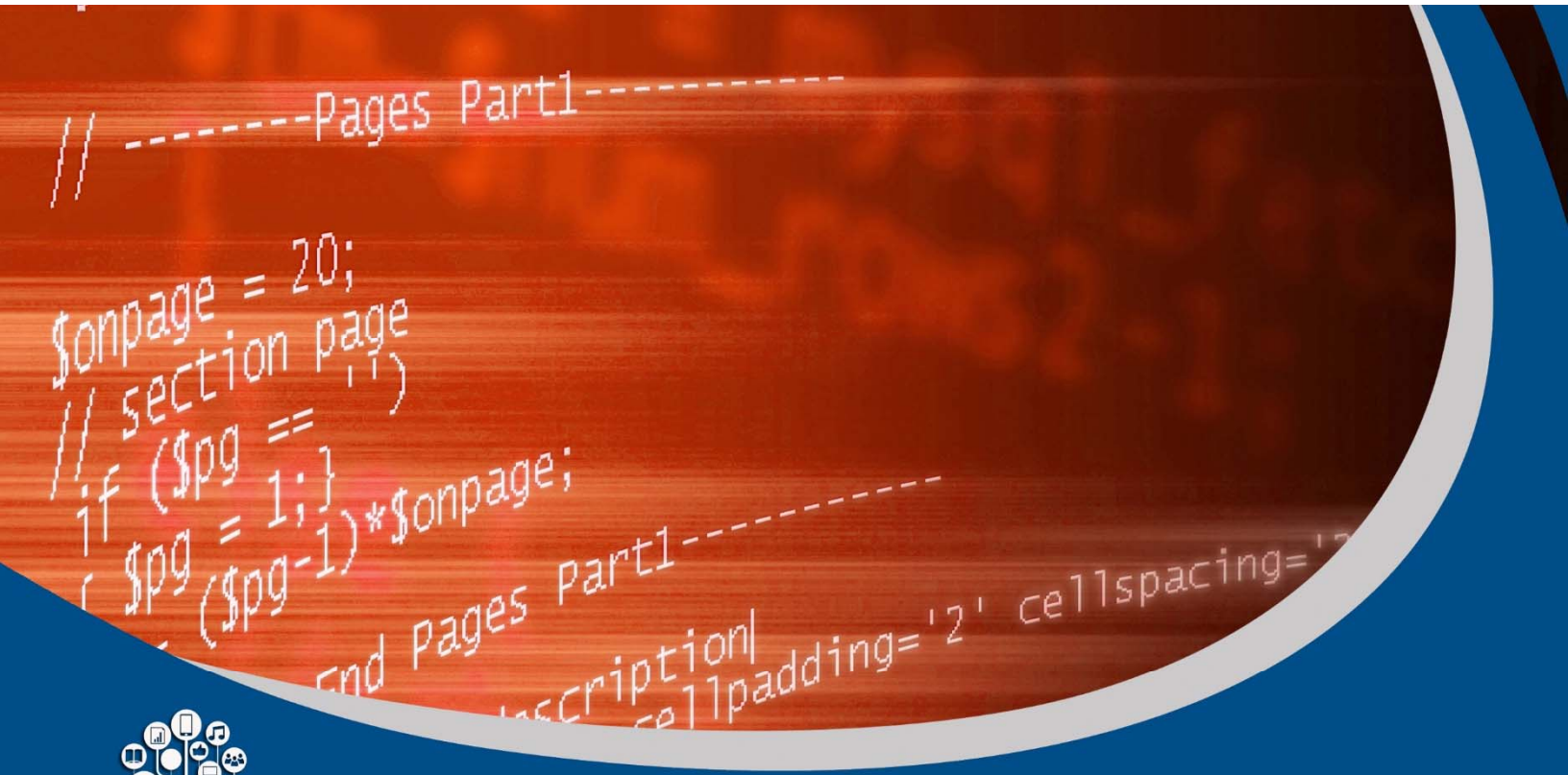
REVISIÓN PEDAGÓGICA
Lic. Melissa Michel Rogel

CORRECCIÓN DE ESTILO
Mtro. Carlos Rodolfo Rodríguez de Alba

DISEÑO DE PORTADAS
L.CG. Ricardo Alberto Báez Caballero

Unidad 1

Fundamentos de las estructuras de datos



UNAM FCA
CEDGEC
Centro de Educación a Distancia y Gestión del Conocimiento

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

Al terminar la unidad, el alumno conocerá las estructuras de datos, su relación con los tipos de datos y su importancia para la abstracción de datos.

TEMARIO DETALLADO
(8 horas)

1. Fundamentos de las estructuras de datos

1.1. Definición de estructura de datos

1.2. Tipos de datos

1.3. Tipos de datos abstractos

INTRODUCCIÓN

Al almacenar datos en una computadora, se pretende (entre otras cosas) realizar posteriormente operaciones con dichos datos, cuyo ejemplo podrían ser las operaciones matemáticas; sin embargo, existen datos y operaciones más complejos, v. gr., asociar unos caracteres con otros (operación concatenar), pues su vínculo definirá las operaciones que tendremos que realizar.

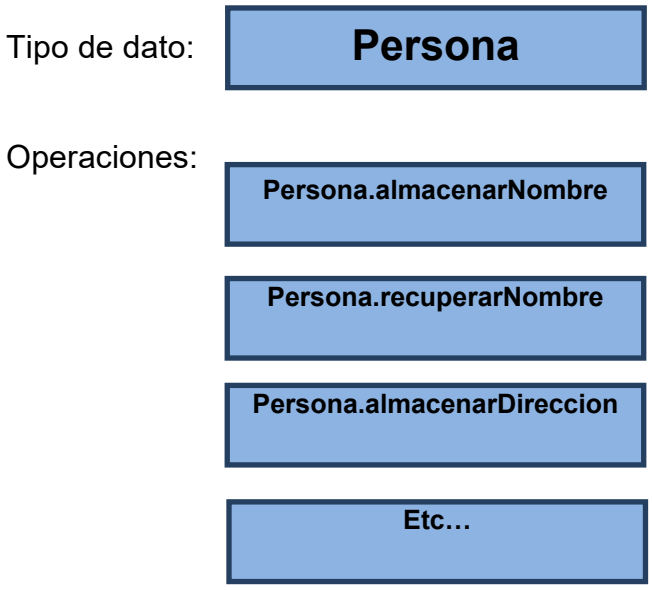
Conocer los tipos de datos más simples es de suma importancia, sobre todo para aquellos que se interesan en la programación, pues esto les permitirá identificar el tipo de operaciones que se realizarán y en que aplicaciones podrán llevarlas a cabo. Por otro lado, identificar este tipo de datos, te permitirá conocer la forma en que se pueden combinar para formar datos compuestos y, de esta manera, elegir la operación que más se adecue a los problemas que queremos resolver.

En esta unidad estudiaremos los diferentes tipos de datos, especialmente los abstractos y las estructuras más simples alojadas en la memoria principal de la computadora, lo anterior es importante por dos razones fundamentales: la primera, porque de ellas se forman otras estructuras más complejas y, la segunda, porque varios lenguajes de programación incluyen los tipos de datos estándar.

1.1. Definición de estructuras de datos

Imaginemos que vamos a crear una agenda de contactos; para ello necesitaremos algunos datos personalizados que organizaremos y ocuparemos de acuerdo a la estructura de datos que construyamos y necesitemos. Comencemos con un dato sencillo, le asignaremos el nombre de **persona**. Dentro de **persona** tendremos la oportunidad de almacenar más información, tal como nombre completo, dirección, correo electrónico, teléfono, etcétera. Posterior al almacenamiento de dichos datos, se realizarán operaciones sobre ellos tales como: almacenar el nombre, almacenar la dirección, recuperar el nombre o recuperar la dirección.

Figura 1.1. Ejemplo de tipo de dato y sus operaciones



Fuente: autoría propia, (2018)

Las estructuras de datos, en palabras de Joyanes (1996), se precisan así: “una colección de datos que pueden ser caracterizados por su organización y las operaciones que se definen de ella”

Dentro de la programación, la organización de la información en la memoria interna de la computadora (Random Access Memory, RAM) se representa con estructuras de datos; por otro lado, el contenido en la memoria RAM se representa con tipos de datos simples vía los cuales se puede acceder, manejar y almacenar la información (operaciones).

Los tipos de datos son objetos que representan tipos de información determinada, almacenada y manejada en la memoria interna de la computadora y que se utilizarán para una aplicación.

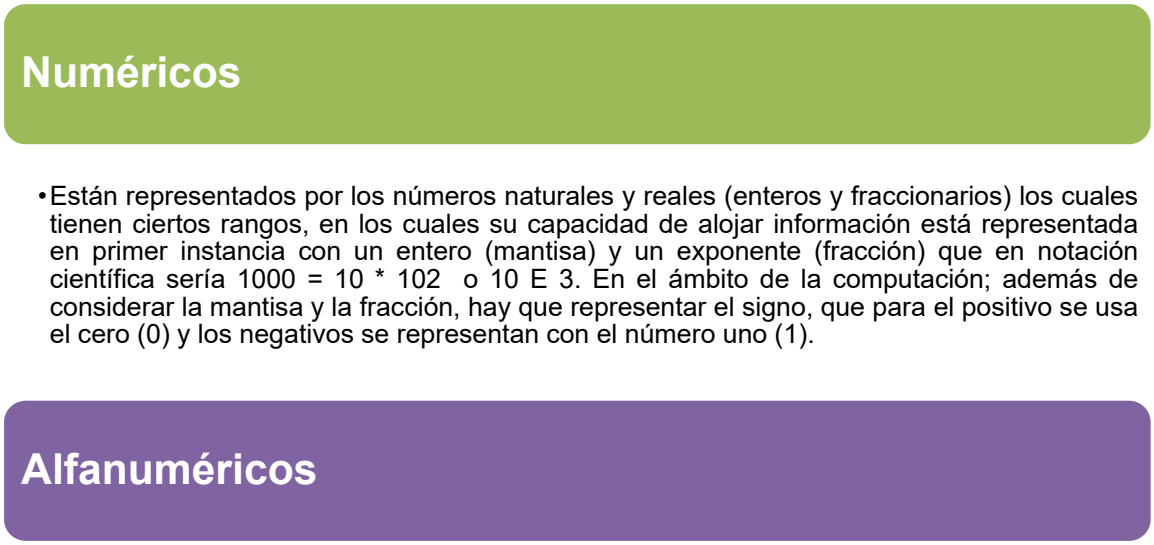
La organización interna de los datos es muy importante en virtud de que la memoria interna de la computadora es un recurso limitado y que se comparte con otros procesos en fracciones de segundo, por lo que, si no se sabe administrar dicho espacio, corremos el riesgo de desperdiciar y desaprovechar la capacidad y potencial de la computadora para las diferentes aplicaciones que se van a utilizar. Por ejemplo, si en una localidad de memoria para almacenar el número 1 con formato real o *float,* cuando *a priori* sabemos que no ocuparemos las decimales, implicará un desperdicio para esa localidad; en cambio, si necesitamos calcular el promedio de las estaturas de los alumnos y definimos la variable *PROM* como *integer*, el compilador truncará las decimales resultantes del cálculo, o, en el mejor de los casos, redondeará el resultado. Los tipos de datos ofrecidos por defecto en los compiladores son llamados **estándar** o **primitivos,** los cuales serán abordados más adelante.

A continuación, estudiaremos los datos simples en su forma conceptual y su forma de representación en la memoria interna de la computadora y la manera de concatenar

los ítems (elementos) para integrar estructuras más complejas con las cuales realizar alguna aplicación en particular. Cabe mencionar que las estructuras simples no pueden contener otras estructuras de información.

Los tipos de datos manejados por la computadora básicamente son dos, obsérvalos en la siguiente figura:

Figura 1.2. Tipos de datos en computadora



Fuente: autoría propia (2018)

Elaborado por: Michel, M. (2018)

Teniendo en cuenta lo explicado anteriormente, podemos considerar las estructuras de datos como representaciones de la forma en que se organiza la información en la memoria interna de la computadora para su manipulación posterior. Son modelos teóricos de cómo se agrupan los datos en la memoria interna (memoria RAM) de la computadora para después ir construyendo estructuras más amplias.

En conclusión, los tipos de datos están íntimamente relacionados con las estructuras de datos, ya que éstos son ofrecidos por los compiladores actuales, posteriormente se les asignarán nombres a variables [1], por lo que las estructuras de datos son organizaciones de información que conforman estructuras con propiedades y formatos englobados bajo identificadores asignados por el usuario (nombres de variables), para emplearse posteriormente en estructuras de programación y así ofrecer mayor versatilidad, impuesta por las aplicaciones actuales del mundo real.

1.2. Tipos de datos

Al hablar de tipos de datos en programación, hacemos alusión a los atributos y/o características de dichos datos, que permiten al programador (o a la computadora) distinguir qué valores pueden tomar, y determinar las operaciones que se pueden procesar. Lo anterior deja ver que existe una estrecha relación entre los lenguajes de programación y los tipos de datos, pues una de las formas de instruir y proporcionar información a una computadora, es precisamente por medio del lenguaje de programación.

En general, los tipos de datos que se pueden encontrar más comúnmente en los lenguajes de programación son los denominados enteros, los números de coma flotante o decimales, cadenas alfanuméricas, etc., no descartando la posibilidad de que algún lenguaje pudiese manejar terminología diferente.

En este tema revisaremos cómo se relacionan los actuales y distintos lenguajes de

[1] Equivalentes a las variables en matemáticas.

programación a través del manejo de las estructuras de datos.

Comenzaremos con los datos de tipo **carácter**, los cuales se pueden unir, formar una cadena y después otro tipo de estructuras llamadas arreglos.

Los de tipo **entero** pueden ser convertidos en números de punto flotante o float; es decir, con punto decimal.

Los tipos de datos se dividen en: simples estándar o primitivos y son necesarios para formar estructuras, así como aquellos datos que son definidos por el programador (llamados también incorporados). A continuación se revisarán las características de cada uno en la siguiente tabla:

Tabla 1.1. Características de los tipos de datos

Tipos de datos estándar No presentan una estructura, son unitarios y nos permiten almacenar un solo dato.	
Entero	El tipo entero es un subconjunto de los números enteros, que dependiendo del lenguaje de programación que estemos usando, podrá ser mayor o menor que 16 bits (2^{16}=32768); es decir, se pueden representar desde el -32768 hasta el 32767. Para representar un número entero fuera de este rango se tendría que usar un dato de tipo real.
Real	El tipo real define un conjunto de números que puede ser representado con la notación de punto-flotante, por lo que nos permite representar datos muy grandes o muy específicos.
Carácter	Cualquier signo tipográfico. Puede ser una letra, un número, un signo de puntuación o un espacio. Generalmente, este tipo de dato está definido por el conjunto ASCII.

Lógicos	Este tipo de dato, también llamado *booleano*, permite almacenar valores de lógica booleana o binaria; es decir, representaciones de verdadero o falso. Nota: La definición del tipo lógico no es conocida en todos los lenguajes de programación como es el caso de C y PHP, los cuales no manejan variables de tipo booleano como tal.
Float	Este tipo de datos se emplean en aquellos datos fraccionarios que requieren de cifras a la derecha del punto decimal o de entero con varios decimales. De estos se desprende la necesidad de crear el formato de representación científica (10 E +12).

Fuente: autoría propia, (2018)

Por otro lado, existen datos que son definidos por el programador y son aquellos que nos ayudan a delimitar los datos que podemos manejar dentro de un programa y nos sirve como una manera segura de validar la entrada/salida de los mismos. En la siguiente tabla podrás revisar las características de este tipo de datos:

Tabla 1.2. Datos definidos por el programador

Definidos por el programador	
Subrango	En este tipo de datos se delimita un rango de valores, definiendo un valor menor como el límite inferior y un valor mayor como el límite superior, de los posibles valores que puede tomar una variable, de esta manera podemos asegurar que la entrada o salida de nuestro programa estén controladas. Ejemplos: type Tipo1 = 1… 30 type Tipo2 = 1… 10
Enumerativo	Los tipos de dato enumerativo pueden tomar valores dentro de un rango en el que se especifica ordenadamente cada uno de dichos valores, al igual que el tipo de subrango, nos permite restringir los valores de una variable.

Fuente: autoría propia, (2018)

Como vimos en la primera tabla; es decir, la de tipos de datos estándar, los datos de

tipo lógico también son conocidos como booleanos. La definición del tipo booleano o lógico no es soportada o conocida en todos los lenguajes de programación y representa valores de lógica binaria, es decir, dos valores que se expresan, en este caso, en falso o verdadero. Una constante de este tipo de datos, acepta el valor falso o verdadero (*false* o *true*) representándolo con 0 y 1, respectivamente, en muchos lenguajes de programación. El valor de una constante booleana no cambia durante la ejecución del programa. En el caso del manejo de variables booleanas o lógicas también acepta solamente uno de dos valores: verdadero (*true*) o falso (*false*), pero el valor de la variable puede cambiar durante la ejecución del programa.

Otro caso en el que se manejan valores booleanos en diversos lenguajes de programación es para generar un dato o valor lógico a partir de otros tipos de datos, típicamente se emplean los operadores relacionales (u operadores de relación), por ejemplo: 0 es igual a falso y 1 es igual a verdadero. Por ejemplo:

¿Es (3>2)? -> 1 = verdadero

¿Es (7>9)? -> 0 = falso

En el Lenguaje C, no hay ninguna implementación primitiva del tipo booleano, por lo que para declarar variables de este estilo, hace falta definirlo previamente. La manera de hacerlo es añadir entre las declaraciones de tipos la siguiente instrucción, en la cual podemos observar que se define el tipo booleano y asigna a sus elementos *FALSE* y *TRUE*, los valores 0 y 1, respectivamente:

Figura 1.3. Declaración del tipo booleano en lenguaje C

```
typedef enum {FALSE=0, TRUE=1} booleano;
```

Fuente: autoría propia, (2018)

En aplicaciones específicas, como *Microsoft Access*, que es un manejador de bases de datos que podemos encontrar en *Microsoft Office*, los tipos de datos ofrecidos son aún mayores, como los de tipo *MEMO* (un pequeño campo de texto que puede contener hasta 65.536 caracteres por campo), o los de tipo *PICTURE* u *OBJECT*, los cuales pueden almacenar imágenes de cierto tamaño.

1.3. Tipos de datos abstractos

El concepto de Tipo de Dato Abstracto (TDA, Abstract Data Type), surge en 1974 por John Guttag, pero fue en 1975 que, por primera vez, Bárbara Liskov, lo propuso para el lenguaje de programación llamado CLU[2]. Posteriormente fue implementado por lenguajes modulares como Turbo Pascal y Ada, y en la actualidad son estructuras muy utilizadas en el paradigma orientado a objetos.

Los Tipos de Datos Abstractos son modelos con los cuales se representan estructuras con propiedades relativas a un tipo de dato del cual se pueden crear objetos[3], los cuales tienen atributos, y todos ellos servirán para desarrollar una aplicación en particular. A diferencia de los tipos de datos comunes o las estructuras de datos

[2] Para mayor información puedes remitirte al siguiente artículo de Liskov, B. (1992) *A history of CLU* Recuperado *de* http://bit.ly/2HdrpIJ Consultado el 28 de enero de 2018.
[3] Un objeto es el equivalente de un TDA a una variable de tipo estándar.

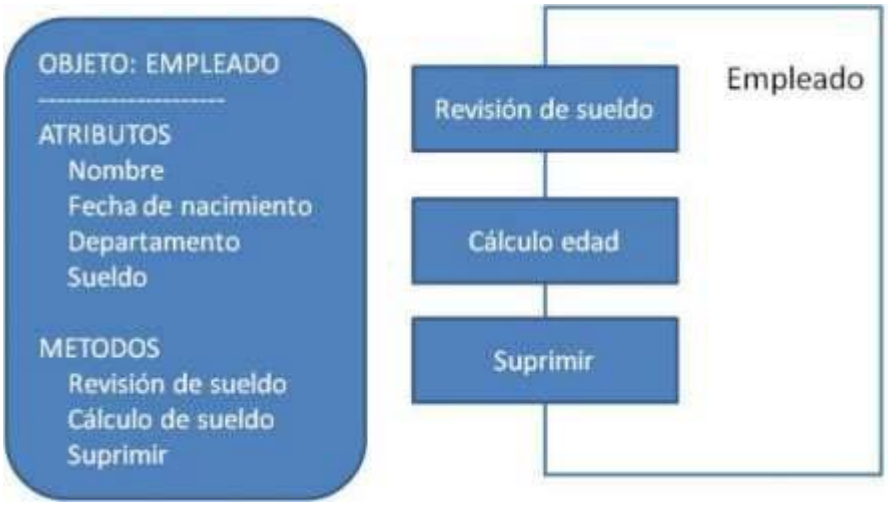
tradicionales, aquí tenemos dos características importantes que son la modularidad y el encapsulamiento.

→ **Encapsulamiento**: Todas las operaciones y las propiedades del tipo de dato abstracto, están definidas dentro de él mismo, nada se define por separado; de esa manera el estado del objeto sólo puede cambiar con las operaciones definidas para él mismo, eso nos da una mejor organización ya que para acceder a las operaciones o propiedades de un objeto de ese tipo, hay que hacer forzosamente referencia a él mismo. Otra característica importante del encapsulamiento, es que los detalles de implementación del tipo de dato abstracto están escondidos para el programador, por lo que sólo tiene que preocuparse de las operaciones y como éstas cambian el estado del objeto por medio de sus mismas propiedades, el cómo funcionan o de qué manera se crearon no le atañe, lo que hace más práctico el poder utilizar los objetos de cierto tipo de TDA.

→ **Modularidad**: Cada objeto de un cierto tipo de TDA es único, y en él residen todas las características del mismo; así como las posibilidades de cambiar su estado; ese elemento puede ser tomado como ente independiente para formar objetos más complejos; es decir, se toman varios objetos de diferentes tipos, para formar objetos más complejos o para agregar nuevas operaciones y propiedades, de esa manera se hace una variante del objeto anterior.

En la siguiente figura podemos observar un ejemplo de un TDA llamado “EMPLEADO”, en él puedes identificar cómo se representan las propiedades o atributos del objeto, así como sus operaciones o métodos.

Figura 1.4. Ejemplo de la definición genérica de un TDA



Fuente: autoría propia, (2018)

En la imagen anterior puedes ver también el ejemplo de encapsulamiento, ya que todo lo que se puede hacer con el empleado (operaciones) está definido dentro de él mismo, también para aclarar la modularidad a través de éste ejemplo, piensa en que al combinar el dato empleado con otro TDA que se llame “Puesto”, podrás tener un objeto compuesto llamado “Administrativo”, que es un empleado que combina las características de su puesto para presentar cierta información combinada de ambos objetos como el tiempo que trabaja al día o las funciones que tiene.

Al comenzar el diseño de un TDA (tipo abstracto), es necesario tener una representación del mismo sobre el cual se quiere trabajar, sin pensar en ninguna estructura de datos concreta en su implementación y el tipo de datos estándar del lenguaje de programación seleccionado para definirlo. Esto va a permitir expresar las condiciones, relaciones y operaciones de los elementos modelados, sin restringirse a una representación interna concreta. En este orden, lo primero que se hace es dar nombre y estructura a los elementos a través de los cuales se puede modelar el estado interno de un objeto, utilizando algún formalismo matemático o gráfico.

Siempre hay que tomar en cuenta que un TDA es un modelo abstracto para resolver un problema y después ya se piensa en cómo implementarlo en programación. La finalidad de crear un TDA es aplicarse de forma general a los problemas semejantes, al originalmente planteado, por lo que resulta independiente de la programación.

Un TDA se define con un nombre, un formalismo para expresar un objeto; es decir, un invariante [4] y un conjunto de operaciones que se pueden realizar sobre este objeto.

Veamos una representación genérica de cómo definir un TDA para un lenguaje de programación genérico:

Figura 1.5. Cómo definir un TDA

[4] Es parte de un sistema que no admite las variaciones que afectan a otras partes del mismo.

```
TDA <nombre>
<Objeto abstracto1>
  <Invariante del TDA>
  <Operaciones>
<Objeto abstracto2>
  <Invariante del TDA>
  <Operaciones>
```

Fuente: autoría propia, (2018)

Ahora veamos cómo definiríamos el TDA de un Auto, de acuerdo al ejemplo anterior:

Figura 1.6. Pseudocódigo de un TDA AUTO

```
TDA AUTO
<Object Tamaño>
        Altura. Centímetros
        Anchura.
        Largo.
<Object Motor>
        Capacidad.
END AUTO
```

Fuente: autoría propia, (2018)

Si se definen más TDA, entonces se necesitarán más renglones para cada propiedad.

RESUMEN

En esta unidad estudiamos el concepto de lo que es una estructura de datos, considerando los diferentes tipos que se procesan así como las estructuras de datos como modelos teóricos que muestran la forma en que la computadora maneja la información en la memoria interna; dichas estructuras son organizaciones de datos que conforman estructuras con propiedades y formatos englobados bajo identificadores asignados por el usuario para emplearse en programas simples o complejos y así ofrecer una mayor versatilidad de manejo de información a las aplicaciones informáticas actuales.

Establecimos que la forma en que se transmite la información o datos a la computadora es por medio de un lenguaje de programación, que soporte y sea capaz de representarla aun cuando sea abundante y compleja, de acuerdo con los avances de la actualidad.

De igual modo, conceptualizamos los tipos de datos como un conjunto de valores que se pueden tomar durante la ejecución de un programa determinado.

Asimismo, se concibe a un TDA (Tipo de dato abstracto) como modelo compuesto por una colección de propiedades y operaciones encapsuladas, para su aplicación en la solución de problemas del mismo tipo; diferrentes TDA se pueden combinar para crear otros TDA (modularidad) y resolver problemas más complejos.

BIBLIOGRAFÍA

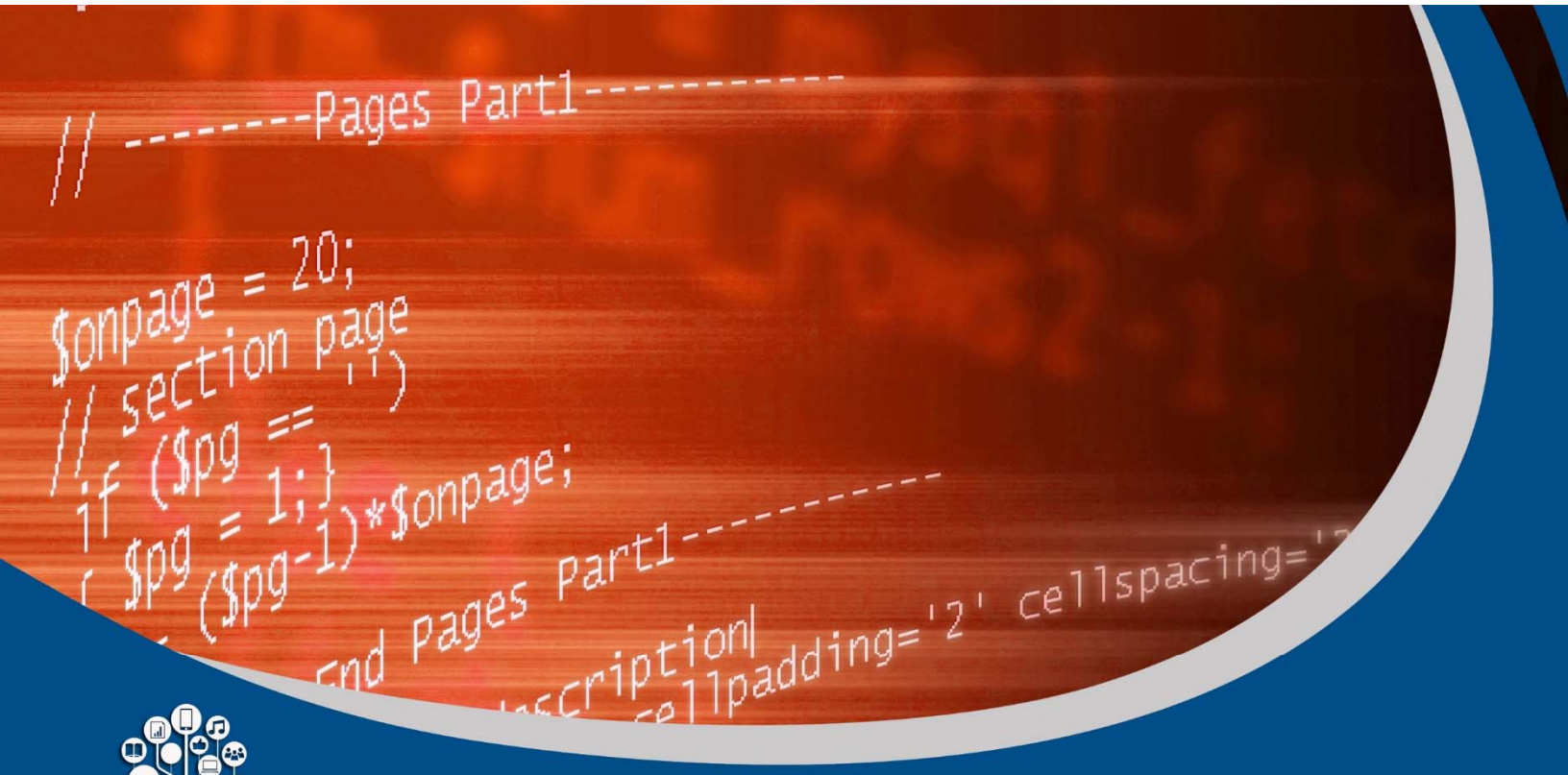
SUGERIDA

Autor	Capítulo	Páginas
Joyanes (1996)	7. Estructuras de datos I	274-283

Joyanes Aguilar, Luis. (1996). *Fundamentos de programación: Algoritmos y estructura de datos*. Madrid: Mc Graw Hill

Unidad 2

Estructuras de datos fundamentales



UNAM FCA
CEDIGEC
Centro de Educación a Distancia y Gestión del Conocimiento

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

El alumno conceptualizará los tipos de datos complejos, su construcción a partir de datos simples y sus características principales para su aplicación en la solución de problemas específicos.

TEMARIO DETALLADO
(16 horas)

2. Estructuras de datos fundamentales

- 2.1. Introducción a los tipos de datos abstractos
- 2.2. Arreglos
  - 2.2.1. Unidimensionales
  - 2.2.2. Multidimensionales
  - 2.2.3. Operaciones con arreglos
- 2.3. Listas
  - 2.3.1 Definición del tipo de dato abstracto lista
  - 2.3.2 Definición de las operaciones sobre listas
  - 2.3.3 Implantación de una lista
- 2.4. Pilas
  - 2.4.1. Definición del tipo de dato abstracto pila
  - 2.4.2. Definición de las operaciones sobre pilas
  - 2.4.3. Implantación de una pila
- 2.5. Colas
  - 2.5.1. Definición del tipo de dato abstracto cola

2.5.2. Definición de las operaciones sobre colas

2.5.3. Bicolas

2.5.4. Implantación de una cola

2.6. Tablas de dispersión, funciones hash

INTRODUCCIÓN

El manejo de los datos complejos se integra de varios tipos de datos simples, dicho manejo puede realizarse ya sea procesando los datos simples, es decir, del mismo tipo o distintos, o bien, de acuerdo a sus necesidades, para lo cual se dividen en dinámicos y estáticos. Los tipos de datos simples ocupan solo una casilla de memoria en los que podemos mencionar en lenguaje C a los de tipo *int, byte, short, long, doublé, float, char y boolean*. Tenemos también a los tipos de datos estructurados que hacen referencia a un grupo de casillas de memoria como los Arreglos o vectores, archivos, árboles, registros, etc. En esta unidad explicaremos las estructuras ya mencionadas.

2.1. Introducción a los tipos de datos abstractos

Imagina un estéreo de auto, éste, al igual que otros aparatos electrónicos, se compone de una serie de circuitos que le permiten funcionar adecuadamente; tú desconoces los detalles de la implementación de dichos circuitos, sin embargo, sí conoces las operaciones (o funciones) que puedes realizar sobre él, como la de sintonizar la estación de radio, subir el volumen o regular el ecualizador (entre otras) ¿cómo llevas a cabo dichas operaciones? A través de los botones del estéreo que, para el contexto de la programación, podemos llamar interfaz.

Tomando como base el ejemplo anterior:

Un Tipo de Dato Abstracto (TDA) está definido por un conjunto valores y las operaciones que se pueden realizar sobre ellos, cuyos detalles de implementación están ocultos y sólo se muestra lo que nos permite realizar las operaciones sobre él (a esto se le conoce también como abstracción). En el ejemplo del estéreo se ignoran los detalles que no son significativos como los circuitos que lo componen o el cableado y se destacan los aspectos que permiten utilizar el estéreo como son el control de volumen, los botones para cambiar de canción, etc.

Las operaciones son los elementos que nos permiten interactuar con el tipo de dato y en su conjunto son conocidas como interfaz. En el ejemplo del estéreo la interfaz sería la carátula y las operaciones son cada uno de los elementos que la componen (control del volumen, display del estéreo, sintonizador de estaciones de radio; etc.).

2.2. Arreglos

Partiendo de un modelo conceptual, podemos decir que, en programación, los arreglos son:

Estructuras de datos compuestos en las que se utilizan uno o más subíndices para identificar los elementos individuales almacenados, a los que es posible tener acceso en cualquier orden.

Fuente: Joyanes, 1996: 276

La siguiente imagen representa un ejemplo de un tipo de arreglo:

Figura 2.1 Arreglo de una sola dimensión (vector)

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
A	**5**	**B**	**k**	**2**	**90**	**54**	**k**	**2**	**100**

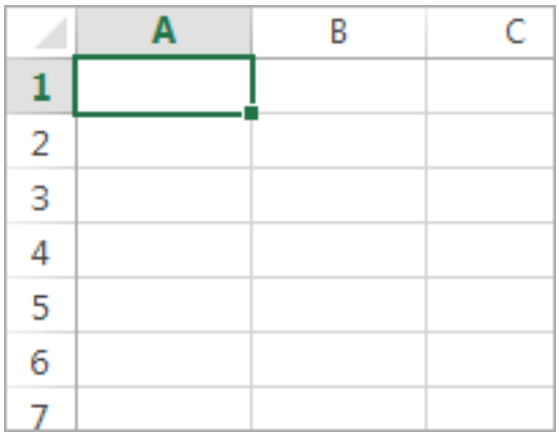
Fuente: autoría propia, (2017)

En el contexto de los sistemas de cómputo, un arreglo (vector o matriz), es una estructura de datos que hace referencia a un grupo de casillas de memoria que se puede ver como una colección finita [5], homogénea [6] y ordenada [7] de elementos. Una forma de pensar en un arreglo es como si fuera una hoja de Excel en donde las columnas y/o reglones son el índice que representa la posición en la que se

[5] **Finita:** indica el número máximo de elementos.
[6] **Homogénea:** del mismo tipo de dato (sea entero, real, carácter, etc.) Esta característica es fundamental en los arreglos ya que en esta estructura de datos no se permite mezclar diferentes tipos.
[7] **Ordenada:** con una secuencia consecutiva a través de un índice.

localiza la casilla que contiene el dato o los datos. [8]

Figura 2.2. Hoja de datos de Excel



Fuente: autoría propia, (2017)

En los lenguajes de programación de bajo nivel, o mejor conocidos como **lenguaje máquina** (ejemplo el Lenguaje C), un arreglo puede tener todos sus elementos, ya sea de tipo entero o de tipo carácter, pero no de ambos al mismo tiempo; sin embargo, en los lenguajes de alto nivel, también conocidos como **de hoy en día**, puede haber arreglos con tipos de datos mixtos.

Las dos operaciones básicas que se pueden realizar en un arreglo son:

ALIMENTACIÓN

- Acepta un acceso a una posición del arreglo con la ayuda de un dato de tipo índice, ya sea ordinario o entero, inicializándolo desde el 0 o 1 dependiendo del lenguaje.

EXTRACCIÓN

- Hace uso del índice a fin de llegar al elemento deseado para después eliminarlo.

[8] Recuerda que en un arreglo, todos los datos deben ser del mismo tipo.

El elemento de menor valor, en un arreglo del tipo índice, es su límite inferior y el de mayor valor es su límite superior. Observa la siguiente imagen, en ella, **0** es el elemento de menor valor y **9** el de mayor valor.

Figura 2.3 Arreglo de una sola dimensión (vector)

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
A	**5**	**B**	**k**	**2**	**90**	**54**	**k**	**2**	**100**

Fuente: autoría propia, (2017)

2.2.1. Unidimensionales

Un arreglo unidimensional, es una estructura que:

Almacena datos de forma secuencial, está definida por una sola dimensión y sus nodos son leídos hacia una sola dirección.

Desde el punto de vista de las matemáticas, a una matriz (o conjunto de números o valores), de una dimensión se le llama **vector** y está definida por la siguiente notación:

V = [0, 1, 2, 3, 4, 5]

Pero, ¿qué relación tiene un vector con los arreglos unidimensionales? Pues un vector es justamente un arreglo de este tipo que sólo utiliza un índice para referenciar a cada uno de sus elementos. La sintaxis de su declaración en la mayoría de los lenguajes de programación es la siguiente:

```
tipo_dato nombre_arreglo [tamaño];
```

La siguiente imagen es un ejemplo de programación de un arreglo, usando Lenguaje C:

Figura 2.4 Programación de arreglo en C

```
/*Declaración de la biblioteca con funciones para enviar y
recibir cadenas en pantalla */
#include <stdio.h>
/*Programa Principal*/
main() /* Rellenamos el arreglo del 0 al 9 */
{
/*Declaración del arreglo de tipo Entero, con un tamaño de 10
elementos, en éste caso el primer índice será 0*/
int vector[10];
int i;
        for (i=0;i<10;i++) vector[i]=i;
        for (i=0;i<10;i++) printf(" %d",vector[i]);
}
```

Fuente: autoría propia, (2017)

Las características de este arreglo ejemplo son las siguientes:

- ✓ Definimos el arreglo con el nombre de vector
- ✓ Su tamaño es de 10 números enteros
- ✓ Incluye la librería *stdio.h* para manejo de valores de entrada y salida
- ✓ Utiliza dos instrucciones de ciclo: *for*, una para grabar el arreglo con el incremento del índice y *printf* de Lenguaje C para mostrar los valores

2.2.2. Multidimensionales

Un arreglo también puede ser de dos o más dimensiones por lo que desde el punto de vista de las matemáticas una matriz es un arreglo multidimensional. Por ejemplo, en Lenguaje C, se definen igual que los vectores con la excepción de que en este tipo de arreglos se requiere un índice por cada dimensión. Su sintaxis es la siguiente:

```
tipo_dato nombre_arreglo [tamaño 1][tamaño 2]...;
```

En el caso de la matriz bidimensional, ésta se representará gráficamente como una tabla con filas y columnas. Por ejemplo, una matriz de 2X3 (2 filas por 3 columnas), se inicializa de este modo usando el Lenguaje C/C++:

Figura 2.5 Ejemplo de matriz bi-dimensional en Lenguaje C

```
int matriz[2][3] = {
                    { 20,50,30 },
                    { 4,15,166 }
                   };
```

Fuente: autoría propia, (2017)

Otro ejemplo, es el de una matriz de 3X4 (3 filas por 4 columnas); usando los mismos lenguajes, se inicializa de este modo:

```
int numeros[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

Donde quedarían asignados de la siguiente manera:

```
numeros[0][0]=1  numeros[0][1]=2   numeros[0][2]=3   numeros[0][3]=4
numeros[1][0]=5  numeros[1][1]=6   numeros[1][2]=7   numeros[1][3]=8
numeros[2][0]=9  numeros[2][1]=10  numeros[2][2]=11  numeros[2][3]=12
```

Una forma más de llenar el arreglo es mediante una instrucción *FOR* anidada, como se muestra en el siguiente código:

Figura 2.6 Manipulación de una matriz bidimensional en Lenguaje C

```
/* Ejemplo de matriz bidimensional. */
#include <stdio.h>
main() /* Rellenamos una matriz de dos dimensiones */
{
int x,i,numeros[3][4];/* rellenamos la matriz */
printf("Dime  los valores de matriz 3X4\n");
for (x=0;x<3;x++)
 for (i=0;i<4;i++)
        scanf("%d",&numeros[x][i]);
 /* visualizamos la matriz */
  for (x=0;x<3;x++)
   for (i=0;i<4;i++)
     printf("%d",numeros[x][i]);
}
```

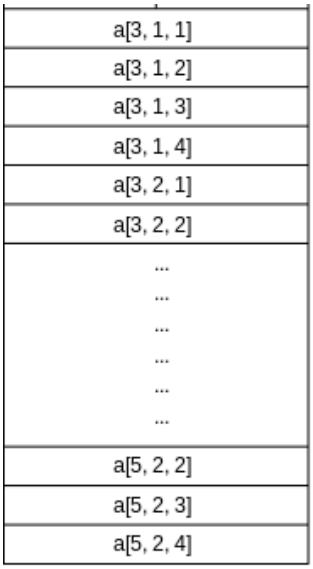
Fuente: autoría propia, (2017)

Como se representa en la imagen anterior, leemos los valores del arreglo con una instrucción *scanf* y los mostramos con una instrucción *printf*. Nótese que en ambos casos utilizamos una instrucción *FOR* anidada.

Por otro lado, un **arreglo de dos dimensiones** ilustra claramente las diferencias lógica y física de un dato pues es una estructura de datos lógicos, útil en programación y en la solución de problemas. Sin embargo, aunque los elementos de dicho arreglo están organizados en un diagrama de dos dimensiones, el hardware de la mayoría de las computadoras no da este tipo de facilidad. El arreglo debe ser almacenado en la memoria de la computadora, la cual usualmente es lineal, es decir, cuentan con una arquitectura peculiar en la que el procesador ingresa la información secuencialmente hasta que una página de memoria se completa para continuar con la siguiente.

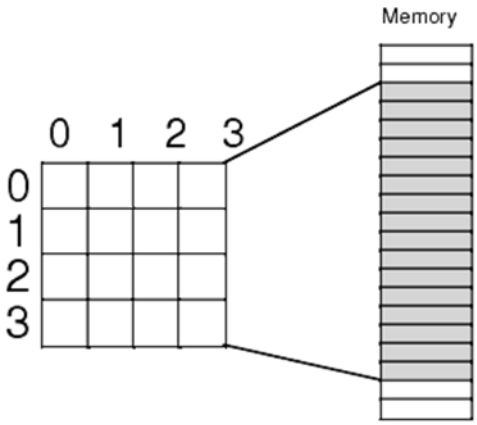
Un método para mostrar un arreglo de dos dimensiones en memoria es la representación fila mayor. Bajo esta representación, la primera fila del arreglo ocupa el primer conjunto de posiciones en memoria reservada para el arreglo; la segunda fila el segundo y así sucesivamente. Observa las siguientes imágenes, te mostrarán de manera gráfica lo anteriormente mencionado.

Figura 2.7 Representación gráfica de un arreglo de memoria



Fuente: Tevfik, A. (2014) obtenido de http://bit.ly/2qGreSE

Figura 2.8 Representación de un arreglo de memoria



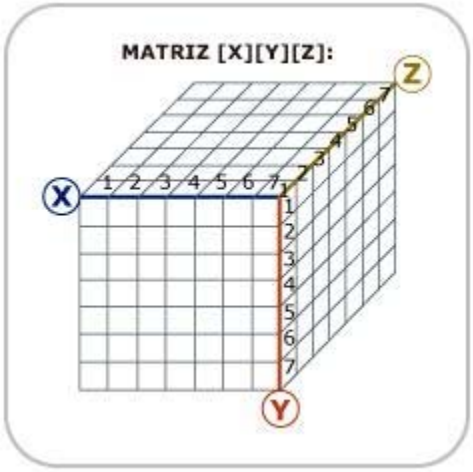
Fuente: Wikispaces (s/a), obtenido de http://bit.ly/2rglnCa

Cabe explicar también el caso de un arreglo con estructura tridimensional, el cual es útil cuando se determina un valor mediante tres entradas y se especifica por medio de tres subíndices:

- ✓ El primer índice precisa el número del **plano**
- ✓ El segundo índice señala el número de la **fila**
- ✓ El tercer índice indica el número de la **columna**.

Observa la siguiente imagen:

Figura 2.9 Descripción gráfica de un arreglo de 3 dimensiones



Fuente: autoría propia, (2017)

Elaborado por: Padilla, V. (2017)

En relación a la imagen anterior, el índice de **Z** no se incrementa, sino hasta que todas las combinaciones posibles de los índices **X** y **Y** hayan sido completadas.

2.2.3. Operaciones con arreglos

Las operaciones básicas que se emplean en los arreglos son las siguientes:

Lectura/Escritura

- El proceso de **lectura** de un arreglo consiste en leer y asignar un valor a cada uno de sus componentes. Similar a la lectura, se debe **escribir** el valor de cada uno de los componentes.

Asignación

- En este caso no es posible asignar directamente un valor a todo el arreglo; sino que se debe asignar el valor deseado a cada componente.

Actualización

- Este proceso se puede dar a través de tres operaciones básicas: **inserción o adición** de un nuevo elemento al arreglo, **eliminación o borrado** de un elemento del arreglo y **modificación o reasignación** de un elemento del arreglo.

Ordenación: inserción, eliminación, modificación y ordenación.

- Es el proceso de organizar los elementos de un vector en algún orden dado que puede ser ascendente o descendente. Existen diferentes formas o métodos para hacer este proceso: método de burbuja, método de burbuja mejorado, ordenación por selección, inserción o método de la baraja, método *Shell*, *Binsort* o por urnas, ordenación por montículos o *HeapSort,* por mezcla o *MergeSort*, método de la sacudida o *Shackersort, Rapid Sort* o *Quick Sort*, por Árboles, etc . (En la unidad 4 abordaremos a detalle estos métodos).

Búsqueda

- Consiste en encontrar un determinado valor dentro del conjunto de datos del arreglo para recuperar alguna información asociada con el valor buscado. Existen diferentes formas para hacer esta operación: búsqueda secuencial o lineal, búsqueda binaria, búsqueda *HASH*, árboles de búsqueda. (En la unidad 5 abordaremos estos métodos a detalle).

Fuente: autoría propia, (2017)

2.3 Pilas

Tal vez la palabra pila te haga pensar en ese pequeño utensilio de metal y químicos que insertas en ciertos aparatos portátiles y que te permite abastecerlos de energía eléctrica para poder utilizar dichos aparatos; tal vez el término pila te remita al concepto que se acopla exactamente al tema que vamos a tocar; es decir, pensarías en lo que obtienes como resultado de apilar un conjunto de elementos.

Cuando se realiza una pila de algo, los elementos que se van apilando se ponen uno encima del otro, lo que quiere decir que no hay otra forma de agregar un elemento a la pila más que por encima, ni por abajo, ni en medio y curiosamente, el único elemento que se pude tomar es el último que se puso en la pila; es decir, el de encima.

Para diferenciar lo que revisaremos en el tema, piensa lo siguiente: ¿qué pasaría si intentaras apilar un conjunto de elementos perecederos como las naranjas? Ahora, ¿qué pasaría si al tomar alguna, eligieras siempre la que agregaste al final? Llegará un momento en el que las primeras naranjas que pusiste en la pila se pudrirán. A diferencia de lo anterior, cuando hablamos de pilas desde el punto de vista de un TDA, tenemos que pensar los escenarios en que podríamos utilizar este tipo de dato, ya que puede sernos útil en ciertas circunstancias, pero en otras no, como el ejemplo anterior.

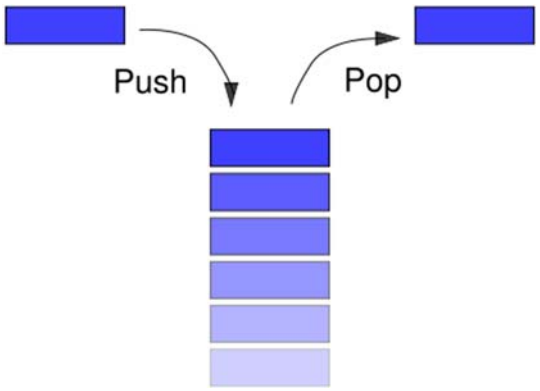
2.3.1. Definición del tipo de dato abstracto pila

Una pila o *stack* es un tipo especial de lista en la que la inserción y borrado de nuevos elementos se realiza sólo por un extremo que se denomina cima o tope

también conocido como *top* (Informática II, 2012).

Una pila es una colección ordenada de elementos en la cual, en un extremo, pueden insertarse o retirarse otros elementos, colocados por la parte superior de la pila. Una pila permite la inserción y eliminación de elementos, por lo que realmente es un objeto dinámico que cambia constantemente.

Figura 2.10. Representación abstracta de una Pila



Fuente: Boivie, (2012) obtenido de recuperado de http://bit.ly/2qLvNGo

Es importante notar que los elementos que se agregan por un extremo de la lista o arreglo (pensando en un arreglo, como una lista de elementos), se sacan por el mismo extremo; es decir, el último elemento que se agrega a la lista, es el primer elemento que se puede sacar y ningún otro más, por eso a este tipo de estructuras, se les conoce como LIFO (*last-in*, *first-out* o última entrada - primera salida)

Un ejemplo de pila o *stack* se puede observar en el mismo procesador de un sistema de cómputo; es decir, cada vez que en los programas aparece una llamada a una función, el microprocesador guarda el estado de ciertos registros en un segmento de memoria, conocido como *Stack Segment*, mismos que serán recuperados al regreso de la función.

2.3.2. Definición de las operaciones sobre pilas

Las operaciones que debe tener una pila son:

push() (agrega un elemento a la pila)
pop() (saca un elemento de la pila)

Otras operaciones que son menos relevantes, pero que pueden utilizarse en una pila son:

isEmpty() (Evalúa si la pila está vacía o no)
top() (Muestra el último elemento de la pila)
size() (Muestra el tamaño actual de la pila)

2.3.3. Implementación de una pila

A continuación, explicaremos por medio de imágenes y código de programación en lenguaje C, la forma de implementar una pila con una lista enlazada simple.

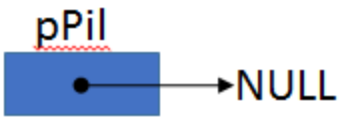
Figura 2.11. Declaración de una pila en Lenguaje C

```c
#include <stdio.h>

//Definición del nodo que representa un elemento de la lista en donde se implementará la pila
typedef struct ElementoLista {
    int dato;
    struct ElementoLista *sig;
}Elemento;


//Declaración de los elementos iniciales de la pila
Elemento *pPila;
int tamano;

//Inicialización de la pila
void init(){
  pPila = NULL;
  tamano = 0;
}
```

Fuente: autoría propia, (2017)

En relación a código anterior, es importante tomar en cuenta que el inicio de la pila debe ser declarado con un apuntador a nulo, como se ve representado en la siguiente imagen.

Figura 2.12. Representación de un apuntador a nulo (creación de una pila)



Fuente: autoría propia, (2017)

Por ejemplo, si quisiéramos agregar a la pila el elemento 6 y después el elemento 100, lo declararíamos como se puede apreciar en la siguiente imagen.

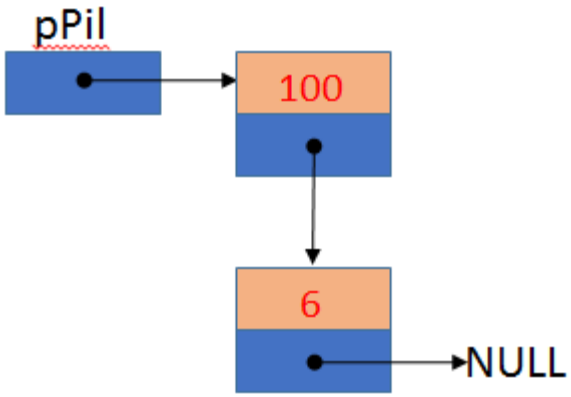
Figura 2.13. Declaración para agregar el elemento 6 y el elemento 100 a una pila en Lenguaje C

```
int main(){
    init();
    push(6);
    push(100);

    system("pause");

    exit(0);
}
```

Fuente: autoría propia, (2017)

La representación de la declaración anterior es la siguiente.

Figura 2.14. Representación de agregar el elemento 6 y después el 100 a una pila



Fuente: autoría propia, (2017)

A continuación y partiendo de los elementos que hemos agregado a la pila, realizamos la operación pop(), como se aprecia en la siguiente imagen donde se declara el código en Lenguaje C.

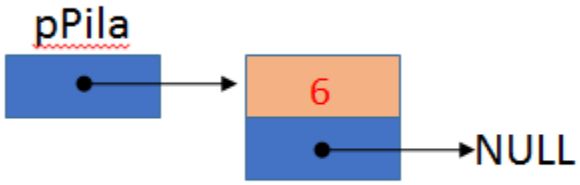
Figura 2.15 Declaración de la operación pop() en Lenguaje C

```
int main(){
	init();
	push(6);
	push(100);
	pop();

	system("pause");

	exit(0);
}
```

Fuente: autoría propia, (2017)

A continuación se muestra la representación del declaración anterior, del resultado de aplicar la operación pop() a la pila.

Figura 2.16. Representación de la pila después de aplicar la operación pop()



Fuente: autoría propia, (2017)

Por último, se muestra en la siguiente imagen la declaración completa del código en Lenguaje C para implementar las operaciones push() y pop().

Figura 2.17. Declaración de las operaciones *push()* y *pop()* para una pila en Lenguaje C

```
//Agregar un elemento a la pila
int push(int entero){
  Elemento *nodo;
  if ((nodo = (Elemento *) malloc (sizeof (Elemento))) != NULL){
    nodo->dato = entero;
    nodo->sig = pPila;
    pPila = nodo;
    tamano++;
    return 0;
  }else{
        printf("\nNo se puede agregar el elmento\n");
        return -1;
  }
}
//Quitar un elemento de la pila
int pop(){
  Elemento *nodo;
  int i;

  if (!isEmpty()){
  for(i=0;tamano == i; i++){
    nodo=nodo->sig;
  }
  nodo = pPila;
  pPila = pPila->sig;
  free (nodo);
  tamano--;
  return 0;
  }else{
        printf("\nLa pila está vacía\n");
        return -1;
  }
}
```

Fuente: autoría propia, (2017)

2.4. Colas

En la vida real existen muchos tipos de colas, por ejemplo, la cola para comprar boletos del metro, para pagar los productos que compramos en el supermercado o la más conocida, la que hacemos para comprar las tortillas. Si ponemos atención, nos daremos cuenta de que una de las características de las colas es la justicia en cuanto al momento en que se le despacha o atiende a la persona que está formada, ya que se les va atendiendo como van llegando, no antes, ni después, sino respetando el hecho de haber llegado antes que alguien más, que no será atendido hasta que la persona que llegó antes sea atendida.

2.4.1. Definición del tipo de dato abstracto cola

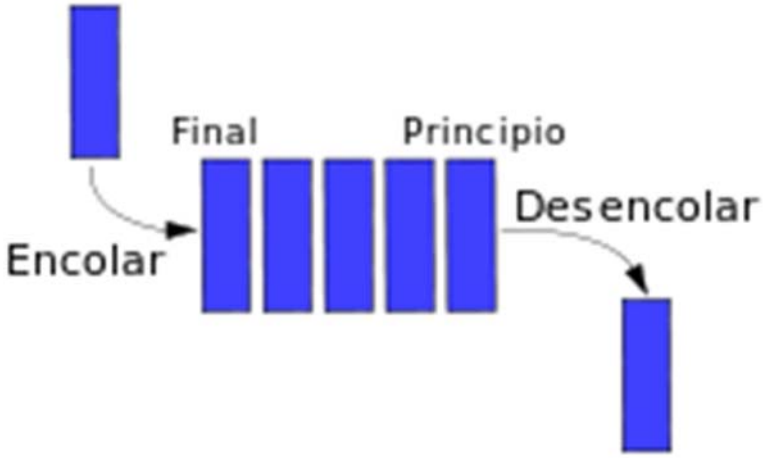
En la vida cotidiana es muy común ver las colas como una forma para agilizar los servicios de una empresa u organización. En informática, una cola representa una estructura de datos en la cual sólo se pueden insertar nodos en uno de los extremos de la lista y sólo se pueden eliminar nodos en el otro extremo. También se le llama estructura FIFO (*first-in*, *first-out* o primera entrada, primera salida), debido a que el primer elemento en entrar será también el primero en salir. Al igual que las pilas, las escrituras de los datos son inserciones de nodos y las lecturas siempre eliminan el nodo leído.

Las colas se utilizan en sistemas informáticos, bancos, empresas, servicios, transportes y operaciones de investigación (entre otros), donde los objetos, transacciones, personas o eventos, son tomados como datos que se almacenan y se guardan mediante colas para su posterior procesamiento, tal cual como una fila

(cola) en el banco, el proceso implica que la primer persona que entra, es la primera que sale.

En el siguiente ejemplo vemos una representación gráfica de una cola.

Figura 2.18. Representación abstracta de una cola



Fuente: Vegpuff, (2012) obtenido de http://bit.ly/2syMBlk

2.4.2. Definición de las operaciones sobre colas

Las operaciones que debe tener una cola son:

enqueue() (Se añade un elemento al final de la cola)

dequeue() (Se elimina un elemento del frente de la cola; es decir, el primero que entró)

Otras operaciones que son menos relevantes, pero que pueden utilizarse en una cola son:

isEmpty() (Evalúa si la pila está vacía o no)

getFront() (Devuelve el primer elemento que entró a la cola)

size() (Muestra el tamaño actual de la pila)

2.4.3. Bicolas

La doble cola o bicola, es una variante de las colas simples. Esta es una cola de dos dimensiones en la que las inserciones y eliminaciones pueden realizarse en cualquiera de los dos extremos de la lista, pero no por la mitad (Informática II, 2012). A diferencia de una cola simple, en donde solo se necesitan un método para leer y otro para escribir componentes en la lista, en una doble cola debe haber:

- ✓ 2 métodos para leer: uno para hacerlo por el frente y otro para leer por atrás
- ✓ 2 métodos para escribir: uno para hacerlo por el frente y otro para escribir por atrás

Por otro lado, existen dos variantes de las bicolas:

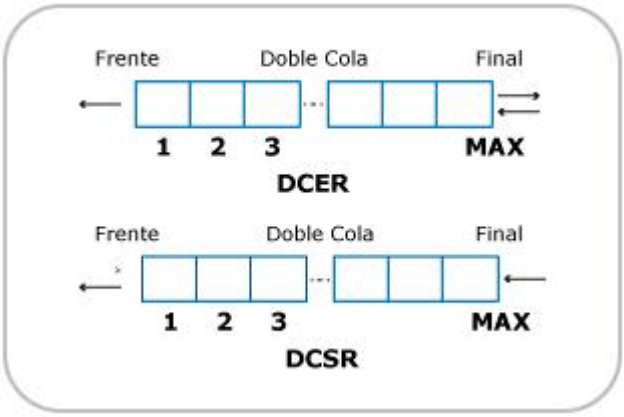
DOBLE COLA CON ENTRADA RESTRINGIDA (DCER)
En ellas se permite hacer eliminaciones por cualquiera de los extremos mientras que las inserciones se realizan solo por el final de la cola.

LA DOBLE COLA CON SALIDA RESTRINGIDA (DCSR)
Aquí las inserciones se realizan por cualquiera de los dos extremos, mientras que las eliminaciones solo por el frente de la cola.

A continuación, veremos de manera gráfica cómo se representan las operaciones

en las bicolas DCER y DCSR, observa en la imagen cómo por medio de flechas se muestran las inserciones y eliminaciones que se realizan sobre ellas.

Figura 2.19 Representación gráfica de las bicolas DCER y DSCR



Fuente: autoría propia, (2017)

Elaborado por: Padilla, V. (2017)

2.4.4. Implementación de una cola

A continuación, explicaremos por medio de imágenes y código de programación en Lenguaje C, la forma de implementar una cola con una lista enlazada simple.

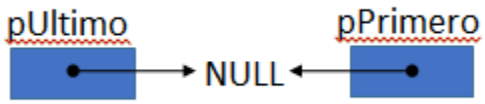
Figura 2.20 Declaración de una cola en Lenguaje C

```c
#include <stdio.h>

//Definición del nodo que representa un elemento de la lista en donde se implementará la cola
typedef struct ElementoLista {
    int dato;
    struct ElementoLista *sig;
}Elemento;


//Declaración de los elementos iniciales de la cola
Elemento *pPrimero,*pUltimo;
int tamano;

//Inicialización de la cola
void init(){
  pPrimero = NULL;
  pUltimo = NULL;
  tamano = 0;
}
```

Fuente: autoría propia, (2017)

En relación al código anterior, es importante tomar en cuenta que el inicio de la cola debe ser declarada con dos apuntadores a nulo, uno que representa el inicio (pPrimero) y otro el final (pUltimo), como se observa en la siguiente imagen.

Figura 2.21 Representación de un apuntador a nulo (creación de una cola)



Fuente: autoría propia, (2017)

Por ejemplo, si quisiéramos agregar a la pila el elemento 6 y después el elemento 100, lo declararíamos como se puede apreciar en la siguiente imagen.

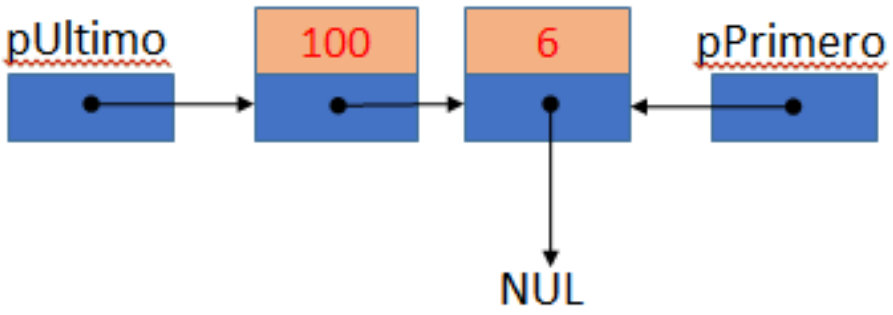
Figura 2.22 Declaración para agregar el elemento 6 y el elemento 100 a una cola en Lenguaje C

```
int main(){
    init();
    enqueue(6);
    enqueue(100);

    system("pause");

    exit(0);
}
```

Fuente: autoría propia, (2017)

La representación de la declaración anterior es la siguiente.

Figura. 2.23 Representación de agregar el elemento 6 y después el 100 a una cola



Fuente: autoría propia, (2017)

A continuación, y partiendo de los elementos que hemos agregado a la pila, realizamos la operación enqueue(), como se aprecia en la siguiente imagen donde se declara el código en Lenguaje C:

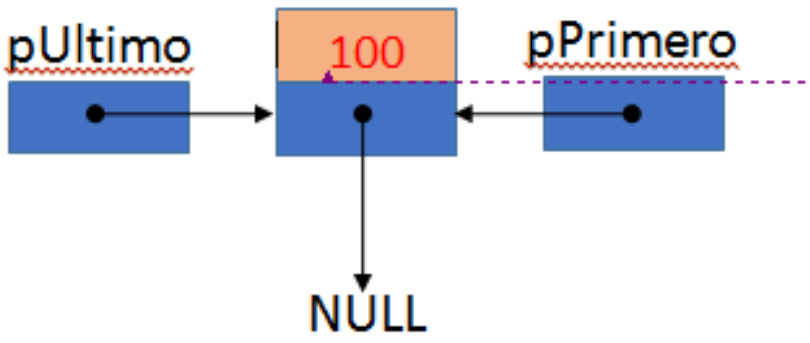
Figura 2.24 Declaración de la operación enqueue() en Lenguaje C

```
int main(){
      init();
      enqueue(6);
      enqueue(100);
      dequeue();

      system("pause");

      exit(0);
}
```

Fuente: autoría propia, (2017)

En seguida se muestra la representación del declaración anterior, del resultado de aplicar la operación enqueue(), a la cola.

Figura 2.25 Representación de la cola después de aplicar la operación enqueue()



Fuente: autoría propia, (2017)

Por último, se muestra en la siguientes imágenes la declaración completa del código en Lenguaje C para implementar las operaciones enqueue() y dequeue().

Figura 2.26. Declaración de las operaciones enqueue() y dequeue() para una pila en Lenguaje C

```
//Agregar un elemento a la cola
int enqueue(int entero){
  Elemento *nodo;

    if ((nodo = (Elemento *) malloc (sizeof (Elemento))) != NULL){
        if (isEmpty()){
           pPrimero=nodo;
        }
       nodo->dato = entero;
       //printf("El elemtno es %d\n", nodo->dato);
       nodo->sig = pUltimo;
       pUltimo=nodo;
       tamano++;
       return 0;
    }else{
         printf("\nNo se puede agregar el elmento\n");
         return -1;
    }
}
```

Fuente: autoría propia, (2017)

Figura 2.27. Continuación de la declaración de las operaciones enqueue() y dequeue() para una pila en Lenguaje C

```
//Quitar un elemento de la cola
int dequeue(){
  Elemento *nodo, *temp;
  int i;

  if (!isEmpty()){
    nodo = pUltimo;
    //Si no apuntan al mismo nodo es que hay más de un nodo
    if(nodo != pPrimero){
       //En esta forma de desplazamiento por la lista, se utiliza un ciclo while en vez de un for
       while(nodo->sig != pPrimero) nodo=nodo->sig;
       temp=pPrimero;
       pPrimero=nodo;
    }else{ //En caso de que haya un sólo nodo
       pPrimero = NULL;
       pUltimo  = NULL;
    }
       free(temp);
       tamano--;
       return 0;

  }else{
        printf("\nLa cola está vacía\n");
        return -1;
  }
}
```

Fuente: autoría propia, (2017)

2.5. Listas

En la vida cotidiana utilizamos las listas para organizar una serie de actividades o pasos que son consecutivos el uno del otro y de esa manera podemos organizar también nuestras ideas, abordando los problemas de manera secuencial y sin olvidar ningún detalle. Un ejemplo de ello pueden ser los pasos que tenemos que seguir para cocinar un pastel, las actividades que tenemos que realizar durante el día o, incluso, sólo pueden ser los elementos de un conjunto, sin algún orden particular, como la lista de compras del súper. Dos detalles importantes a tomar en cuenta en una lista, es que en ella estarán todos los datos que vamos a necesitar para resolver un problema en cierto contexto y que a un dato de la lista le sigue otro, pudiendo ubicar cada uno de esos datos por su posición en la lista como: el primero, el último, el siguiente o el anterior.

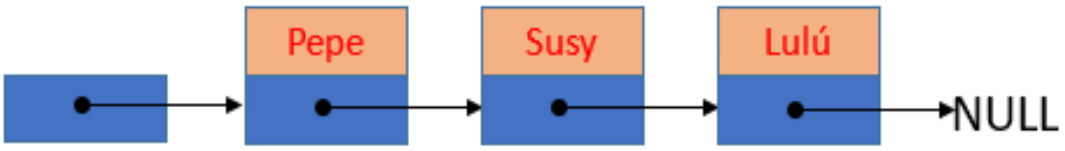
2.5.1. Definición del tipo de dato abstracto lista

Una lista **lineal** es un conjunto de elementos de un tipo dado que se encuentran ordenados, aunque pueden variar en número. Los elementos de una lista se almacenan normalmente de manera contigua, es decir, un elemento detrás de otro en posiciones de la memoria (Informática, 1998).

Una lista **enlazada** es una estructura de datos fundamental que se utiliza para implementar otras estructuras de datos como fue el caso de las pilas, las colas simples y las bicolas. Éstas cuentan con una secuencia de nodos, en los que se guardan campos de datos arbitrarios y una o dos referencias, enlaces o apuntadores al nodo anterior o posterior.

Delante mostramos la forma en que se puede representar una lista enlazada unidireccional, observa la siguiente imagen.

Figura 2.28 Representación de una lista enlazada unidireccional



Fuente: autoría propia, (2017)

2.5.2. Definición de las operaciones sobre listas

Las operaciones que se tienen en una lista son:

Insert() (Agregar un elemento a la lista)
delete() (Quitar un elemento de la lista)

Otras operaciones que son menos relevantes, pero que pueden utilizarse en una lista son:

isEmpty() (Evalúa si la pila está vacía o no)
display() (Muestra los elementos de la pila, desde el inicio al final –NULL-))
size() (Muestra el tamaño actual de la pila)

2.5.3. Implantación de una lista

En seguida explicaremos, por medio de imágenes y código de programación en lenguaje C, la forma de implementar una lista enlazada simple.

Figura 2.30 Declaración de una lista enlazada simple en Lenguaje C

```
#include <stdio.h>

//Definición del nodo que representa un elemento de la lista
typedef struct Nodo {
    char *nombre;
    struct Nodo *sig;
}Lista;


//Declaración de los elementos iniciales de la cola
Lista *inicio;
int tamano;

//Inicialización de la cola
void init(){
  inicio = NULL;
  tamano = 0;
}
```

Fuente: autoría propia, (2017)

En relación al código anterior, es importante tomar en cuenta que el inicio de la lista enlazada simple debe ser declarado con un apuntador a nulo, como se ve representado en la siguiente imagen.

Figura 2.31 Representación de un apuntador a nulo (creación de una lista)

Fuente: autoría propia, (2017)

Por ejemplo, si quisiéramos agregar a la lista 3 nombres de personas, de manera consecutiva, lo declararíamos como se puede apreciar en la siguiente imagen.

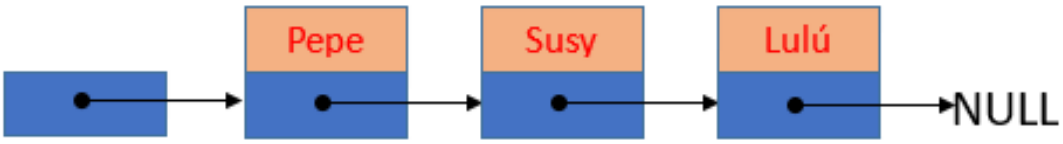
Figura 2.32 Declaración para agregar tres nombres de manera consecutiva a una lista en Lenguaje C

```
int main(){
    init();
    insert("Pepe");
    insert("Susy");
    insert("Lulu");
    display();
    delete();
    display();
    system("pause");

    exit(0);
}
```

Fuente: autoría propia, (2017)

La representación de la declaración anterior es la siguiente.

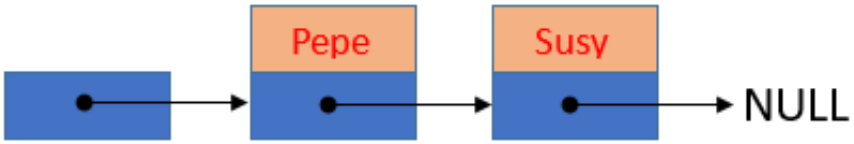
Figura 2.33 Representación de agregar tres nombres de manera consecutiva a una lista



Fuente: autoría propia, (2017)

En caso de aplicar la operación delete() para eliminar un elemento de la lista, la representación sería la siguiente.

Figura 2.34 Representación de aplicar la operación delete() a una lista



Fuente: autoría propia, (2017)

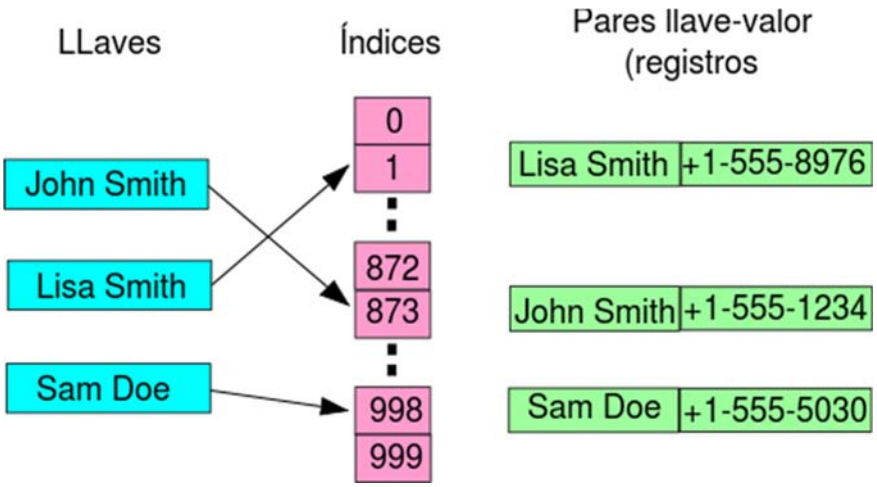
2.6. Tablas de dispersión, funciones hash

Las tablas hash, arreglos hash o mapa hash, son un tipo especial de estructura de datos que realiza una asociación entre llaves o claves con valores y son utilizadas para almacenar grandes cantidades de información.

Lo interesante con esta estructura es que almacena la información en localidades pseudo-aleatorias, por lo que, si se quiere acceder a los valores de manera ordenada, el proceso es muy lento. Se puede decir que las operaciones de inserción, eliminación y búsqueda que se realizan sobre éste tipo de arreglos cuentan con un promedio de tiempo constante; siendo la operación más importante la de búsqueda.

El ejemplo más claro es el de un directorio que cuenta con datos de contacto; en este ejemplo, el nombre del contacto es la llave y los datos del contacto como dirección, teléfono y correo electrónico, son los valores asociados a esa llave.

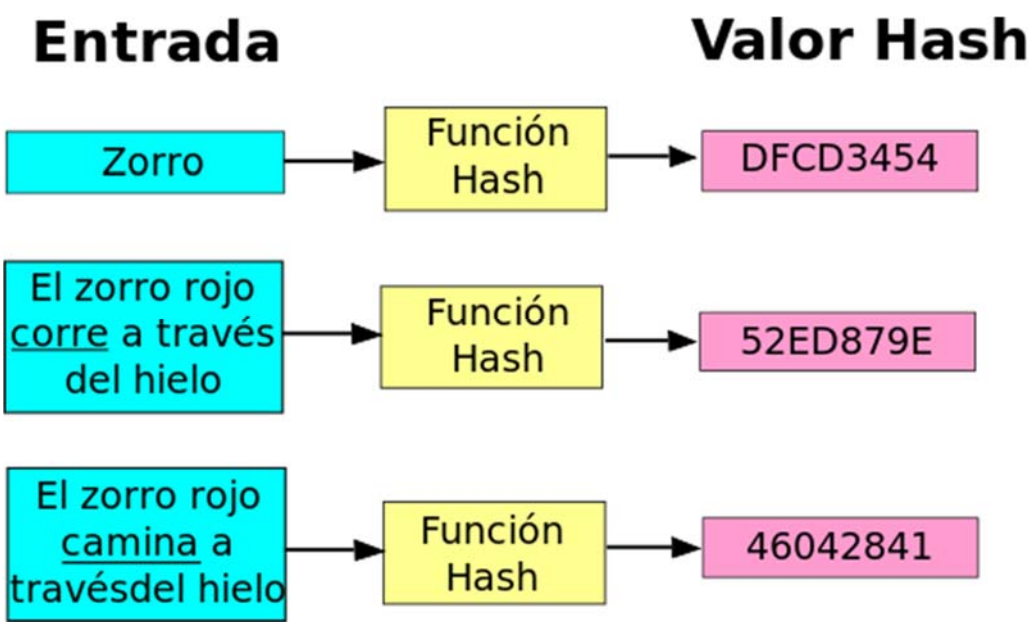
Figura 2.38 Ejemplo de Tabla Hash



Fuente: Wikimedia (s/a) obtenido de http://bit.ly/2rM5JyS

Se pueden implementar como una lista enlazada simple o como una combinación de listas enlazadas con varias llaves.

Figura 2.39 Representación de una tabla hash con listas enlazadas



Fuente: Fercufer, (2011) obtenido de http://bit.ly/2r6Nn7P

Uno de los problemas principales en la implementación de tablas hash es la búsqueda de llaves, por lo cual se deben contar con algoritmos eficientes que permitan llevar a cabo dicha búsqueda [9].

Básicamente estos algoritmos funcionan de la siguiente manera:

1. Se utiliza una llave como índice para buscar un valor.
2. A dicha llave se le aplica la función hash.
3. La función hash debe garantizar que para una entrada dada se pueda obtener una sola salida (valor) de tamaño fijo y que siempre sea constante para esa entrada y sólo para esa entrada.
4. La salida que se obtiene del hash es la llave que sirve de valor único para localizar la información de la tabla.

Como se observa en la imagen anterior, tenemos la palabra Zorro, la cual se

[9] El algoritmo más utilizado es el de *Hash de división.*

convierte, por la función hash, a un código, que es la llave y que nos permite realizar la búsqueda sobre la tabla.

RESUMEN

En esta unidad estudiamos las estructuras de datos fundamentales, ya que el manejo de los datos complejos se integra a partir de datos simples, los cuales, a su vez se dividen en estáticos y dinámicos con la finalidad de optimizar el uso de memoria a través del procesamiento de estructuras de datos adecuadas.

Por un lado, las estructuras de datos estáticas se identifican como aquellas que, desde la compilación, reservan un espacio fijo de elementos en memoria como son los arreglos, vectores de una dimensión y matriz de *n* dimensiones. Por otro lado, las estructuras de datos dinámicas son las que, en tiempo de ejecución, varía el número de elementos y uso de memoria a lo largo del programa; entre ellas están las lineales que son estructuras de datos básicas [10] y lineales[11], uno luego del otro. Comprendiendo las estructuras mencionadas, podemos visualizar que cada nodo puede estar formado por uno o varios elementos (pueden pertenecer a cualquier tipo de dato, regularmente básicos), dichos elementos se reúnen en estructuras de datos como las revisadas en el tema, las cuales se caracterizan por:

Listas simples enlazadas: siempre tienen un enlace o relación por nodo, este enlace apunta al siguiente o al valor NULL (indicador de que la lista está vacía, si es el último nodo).

Lista doblemente enlazada es estructura de datos en forma de lista enlazada de dos vías donde cada nodo tiene dos enlaces: uno apunta al nodo anterior o al valor NULL, si es el primer nodo; y otro que apunta al nodo siguiente o al valor NULL, si es el último nodo.

Lista enlazada circular: el primer y el último nodo están enlazados en forma de anillo o círculo.

[10] Se caracterizan en que todos sus elementos están en secuencia y relacionadas.
[11] Sus elementos se encuentran uno luego del otro.

Lo anterior se puede hacer tanto para listas enlazadas simples como para las doblemente enlazadas.

Estos conjuntos de estructuras de datos se caracterizan por ser estructuras de datos fundamentales que puede ser usadas para implementar otras estructuras más complejas, que, por su programación, dependen de las estructuras fundamentales como son las pilas y las colas. Una de las estructuras lineales más práctica es la pila en la que se pueden agregar o quitar elementos únicamente por uno de los extremos y se eliminan en el orden inverso al que se insertaron, debido a esta característica se le conoce también como últimas entradas, primeras salidas, en inglés LIFO (last input, first output). En cambio, en las colas los elementos se insertan por un sitio y se sacan por el otro, en el caso de la cola simple se insertan por el final y se sacan por el principio, también son llamadas como primeras entradas, primeras salidas, en inglés FIFO (first in first out). Podemos mencionar dos variantes de estructuras de datos colas como son colas circulares y de prioridades.

En las colas circulares se considera que después del último elemento se accede de nuevo al primero. De esta forma se reutilizan las posiciones extraídas, el final de la cola es a su vez el principio, creándose un círculo o circuito cerrado. Las colas con prioridad se implementan mediante listas o arreglos ordenados. Puede darse el caso que existan varios elementos con la misma prioridad, en este caso saldrá primero aquel que primero llegó (FIFO).

Si bien la implantación estática a través del manejo de arreglos es muy natural, para estas estructuras de datos, recomendamos una implantación dinámica para pilas, colas y listas enlazadas, considerando que es la forma más eficiente y apropiada para la optimización del uso de memoria principal.

BIBLIOGRAFÍA

SUGERIDA

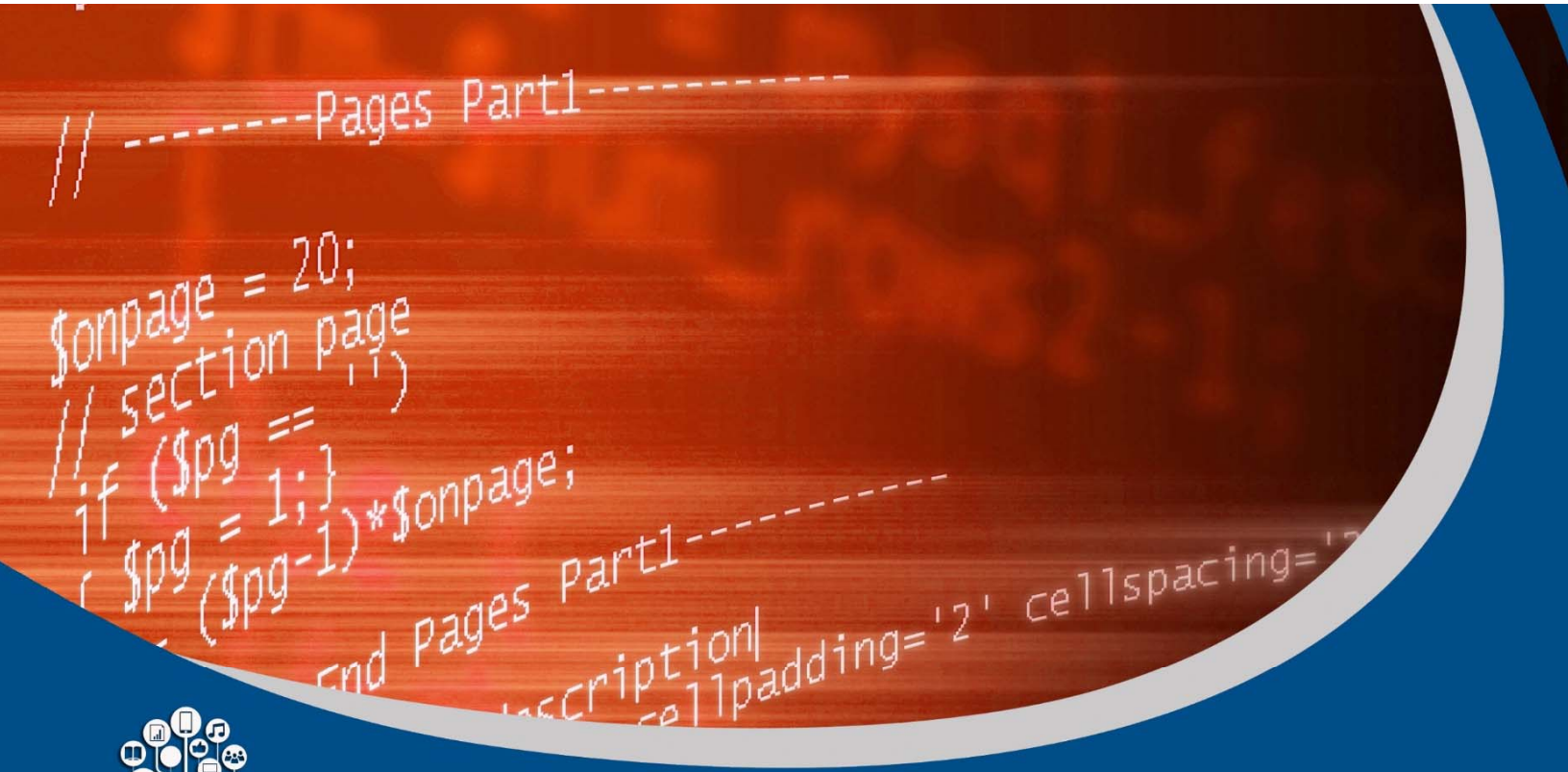
Autor	Capítulo	Páginas
Joyanes (1996)	7. Estructuras de datos I	274-283
Informática II (2012)	Capítulo II	37-63

Joyanes Aguilar, Luis (1996). *Fundamentos de programación: Algoritmos y estructura de datos* (2ª ed.). México: McGraw-Hill [Vista previa de la 3ª ed.]

Informática II. (2012). México: UNAM / Facultad de Contaduría y Administración.

Unidad 3

Estructuras de datos avanzadas



UNAM FCA
CEDIGEC
Centro de Educación a Distancia y Gestión del Conocimiento

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

Al finalizar la unidad, el alumno conocerá las estructuras de datos avanzadas y sus principales aplicaciones en la solución de problemas específicos mediante el uso dinámico de la memoria.

TEMARIO DETALLADO

(16 horas)

3. Estructuras de datos avanzadas

3.1 Árboles

- 3.1.1. Definición del tipo de dato abstracto árbol binario
- 3.1.2. Definición de las operaciones sobre árboles binarios
- 3.1.3. Implantación de un árbol binario

3.2 Grafos

- 3.2.1 Definición del tipo de dato abstracto grafo
- 3.2.2 Operaciones sobre un grafo
- 3.2.3 Implantación de un grafo

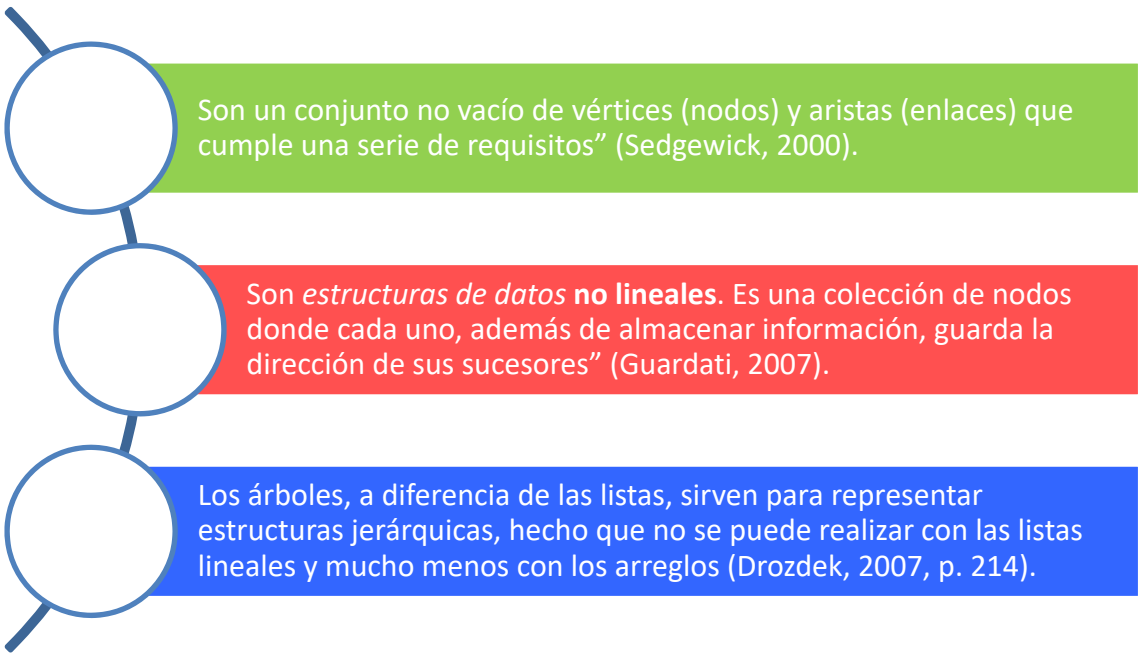
INTRODUCCIÓN

Existen estructuras de datos avanzadas, que se consideran así porque pueden tener una mayor cantidad de operaciones que las estructuras simples y manejan una organización en memoria más compleja. En su manejo pueden implicar también crear variantes de los tipos de datos simples.

3.1. Árboles

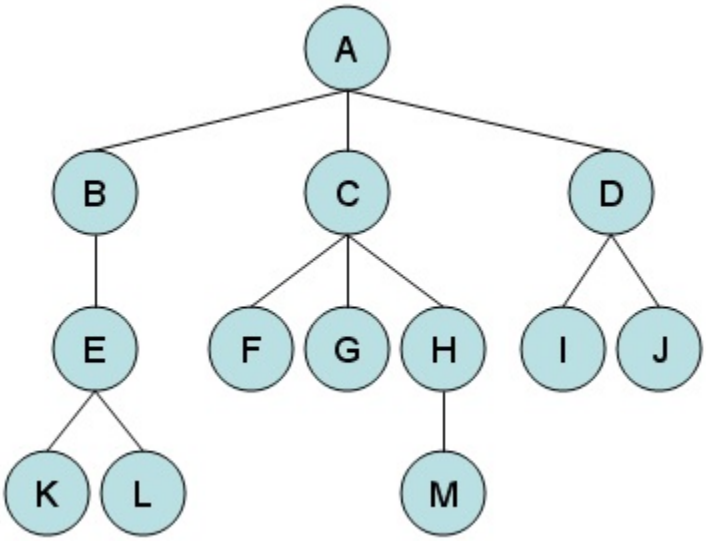
Cuando pensamos en un árbol, como objeto de la vida real, pensamos en un tronco con una serie de ramas y hojas. Podemos pensar que tanto las ramas como las hojas parten del tronco del árbol como una extensión o producto del mismo, en una relación dependiente o jerárquica. Existen tipos de datos que manejan esta relación jerárquica o dependencia entre los elementos para expresar significados particulares de acuerdo al contexto donde se aplican. Para aclarar el concepto de árbol, hablando de la estructura de datos, a continuación revisaremos la definición de acuerdo a diferentes autores.

Figura 3.1. Definición de árboles



Elaborado por: Michel, M. (2018)

Figura 3.2. Representación genérica de un árbol



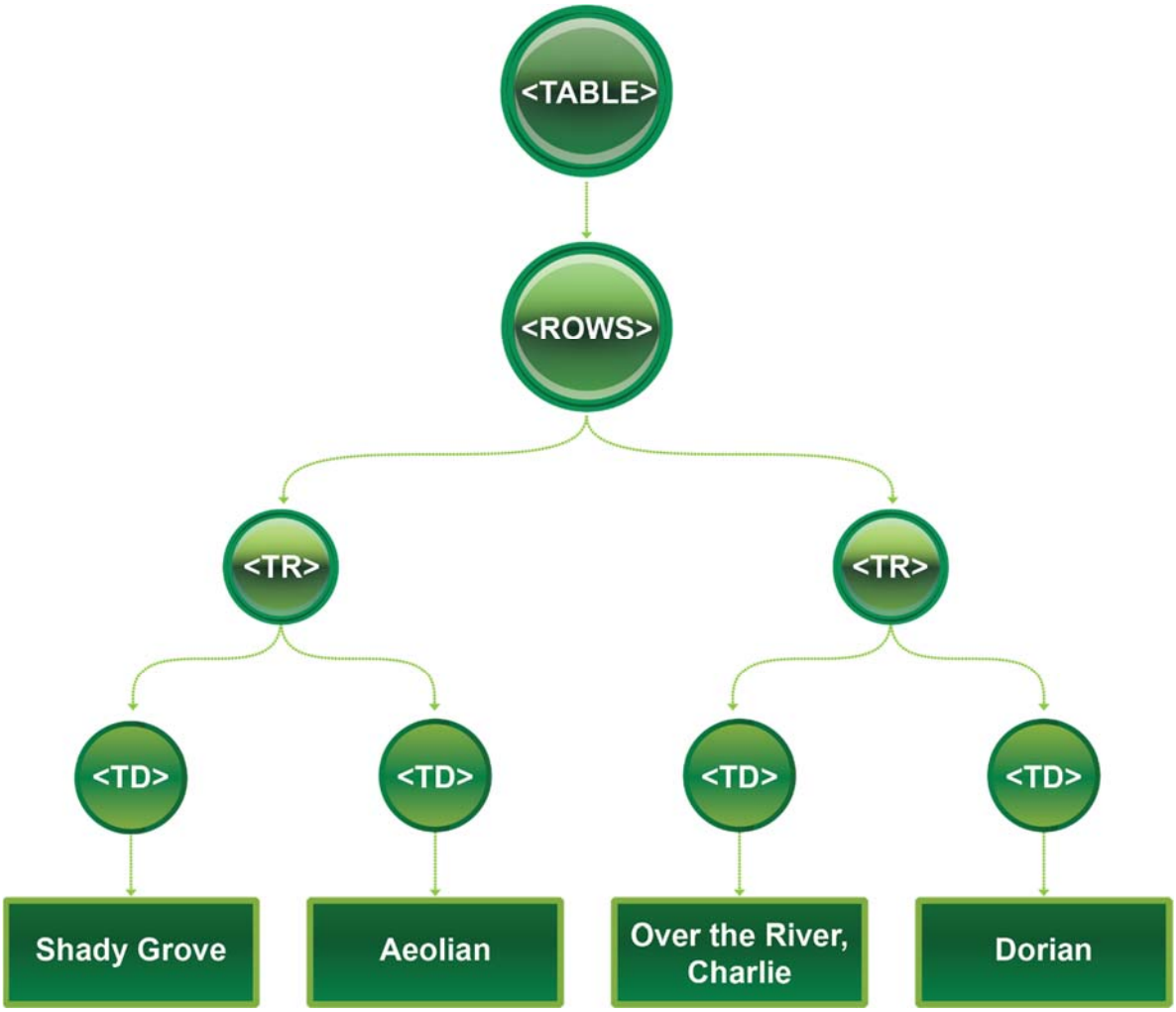
Fuente: Wikimedia (2016) recuperado de https://goo.gl/hrCQdJ

A continuación, revisaremos algunas de las aplicaciones que tienen los árboles:

- ✓ En algoritmos, los árboles son comúnmente utilizados en métodos de clasificación y búsqueda de datos.
- ✓ En áreas como el desarrollo de compiladores los árboles se utilizan para representar expresiones de sintaxis de un lenguaje de programación.
- ✓ En inteligencia artificial se utilizan mucho los árboles para programas que implican la toma de decisiones (árboles de decisión), árboles de juegos, árboles de resolución, etc.
- ✓ Entre las representaciones que se pueden hacer con un árbol tenemos a los árboles genealógicos o las tablas representadas con árboles.

Observa la siguiente imagen, en ella podrás identificar la representación de un árbol en código HTML, el cual te permitirá expresar estructuras jerárquicas de las etiquetas que utilices.

Figura 3.3. Representación de código HTML como un árbol



Fuente: autoría propia, (2017)

Elaborado por: Padilla, V. (2018)

Por otro lado, para comprender la estructura y funciones de los árboles, es importante identificar los elementos que los conforman, los cuales puedes revisar en la siguiente figura:

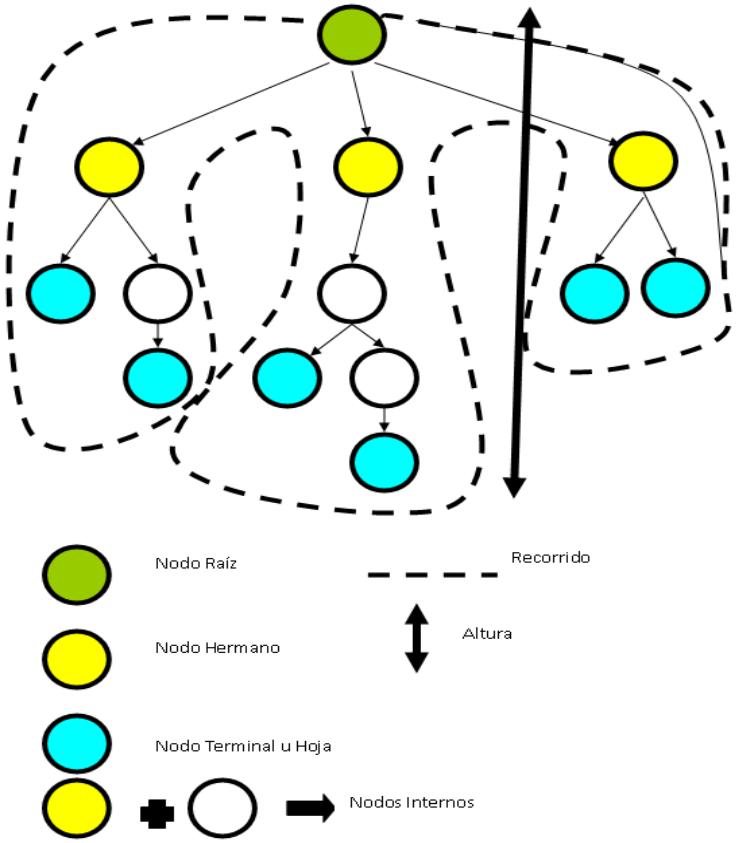
Figura 3.4. Elementos de un árbol

Nodo Raíz (Root)	•Es el elemento principal del árbol y de él parten el resto de los nodos; es el único nodo sin nodo padre.
Nodo (node)	•Son cada uno de los elementos que forman el árbol y se enlistan a contuinuación: •**Nodo hijo (child):** Nodo que desciende de otro nodo. •**Nodo padre (parent):** Nodo que antecede a otro nodo. Todos los nodos, tienen un solo padre, excepto el nodo raíz. •**Nodo terminal u hoja (leaf):** Nodo que no tiene hijos (se encuentra al final del camino). •**Nodo interior (internal node):** Nodos que no son hojas ni raíz**.** •**Hermanos (siblings):** Un grupo de nodos con el mismo padre.
Arista (edge)	•Conexión entre dos nodos
Camino o recorrido (path)	•La secuencia de nodos y aristas que conectan a un nodo con un descendiente.
Nivel (level)	•Número de conexiones que se tiene desde la raíz a un nodo específico.
Rama (branch)	•Aristas entre nodos que termina en una hoja.
Altura o profundidad (depth)	•Número máximo de nodos en una rama.

Fuente: autoría propia, (2018)
Elaborado por: Michel, M. (2018)

En la siguiente imagen podrás observar la ubicación de los elementos anteriormente revisados ya dentro de la estructura de un árbol:

Figura 3.5. Representación gráfica de los elementos de un árbol



Elaborado por: autoría propia, (2018)

3.1.1. Definición del tipo de dato abstracto árbol binario

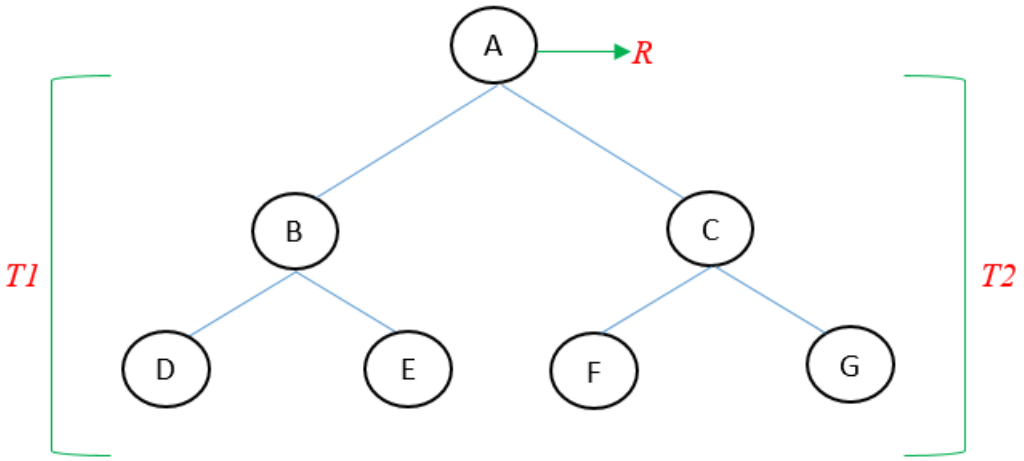
Un árbol binario, que podríamos representar con la letra T, se define como un conjunto finito de elementos llamados nodos, de forma que:

- Si T está vacío se llama árbol nulo o árbol vacío.
- T contiene un nodo distinguido, que llamaremos R, y que es la raíz de T.
- Los restantes nodos de T forman un par ordenado de árboles binarios separados T1 y T2.
- Si T contiene una raíz R, los dos árboles T1 y T2 se llaman, respectivamente, sub-árboles izquierdo y derecho de la raíz R.
- Si T1 no está vacío, entonces su raíz se llama sucesor izquierdo de R; y análogamente, si T2 no está vacío, su raíz se llama sucesor derecho de R.

La siguiente imagen te muestra cómo se representa un árbol binario en la estructura de datos, en la que puedes observar que:

- ✓ B es un sucesor izquierdo y C un sucesor derecho del nodo A.
- ✓ El subárbol izquierdo de la raíz A consiste en los nodos B, D y E, y el subárbol derecho de A consiste en los nodos C, F y G.

Figura 3.6 Representación del árbol binario T



Fuente: autoría propia, (2018)

Elaborado por: Padilla, V. (2018)

Continuando con nuestra descripción del árbol T, tenemos que:

T es recursiva, ya que T se define en términos de los sub-árboles binarios T1 (Nodos B,D y E) y T2 (Nodos C,F yG). Esto significa, en particular, que cada nodo que llamaremos N de T contiene un subárbol izquierdo y uno derecho. Además, si N es un nodo terminal, ambos árboles están vacíos.

Dos árboles binarios T y T' se dicen que son similares si tienen la misma estructura o, en otras palabras, si tienen la misma forma. Los árboles se dice que son copias si son similares y tienen los mismos contenidos en sus correspondientes nodos.

En la memoria de un sistema de cómputo, existen dos formas, que son las más utilizadas, de representar un árbol binario, a saber:

- ✓ Por medio de datos tipo puntero (apuntadores), también conocidos como variables dinámicas
- ✓ Por medio de arreglos

Las dos representaciones anteriores se pueden ver como un registro de datos, donde se tiene como mínimo tres campos:

1. Uno en el que se almacenará la información del nodo.
2. Otro que señala el árbol **izquierdo** del nodo actual.
3. Un tercero que indica al árbol **derecho** del nodo actual.

Figura 3.7. Campos del registro del nodo N

IZQ	INFO	DER

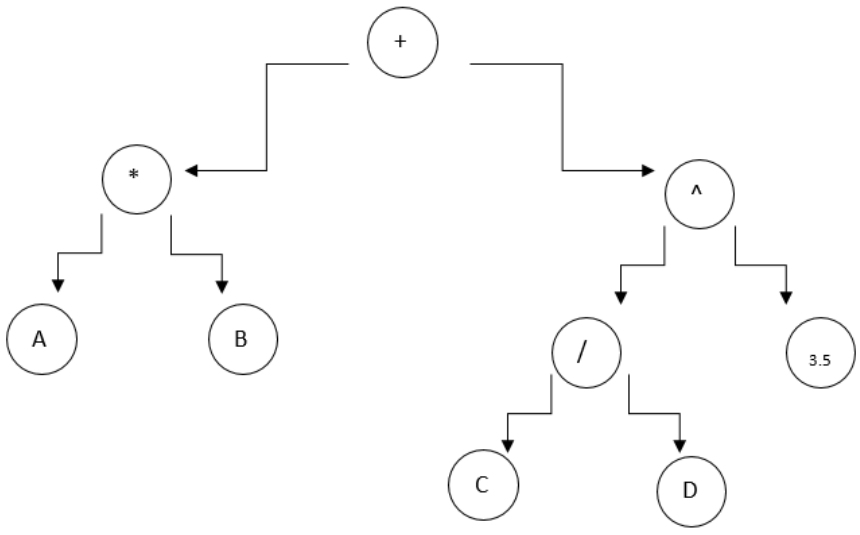
En la imagen anterior vemos los 3 campos que se han mencionado en donde:

- **IZQ**: es el campo donde se almacenará la dirección del subárbol izquierdo del nodo T.
- **INFO**: representa el campo donde se almacena la información del nodo. En este campo se pueden almacenar tipos de datos simples o complejos.
- **DER**: es el campo donde se almacena la dirección del subárbol derecho del nodo T.

A continuación, observarás un ejemplo de representación de un árbol binario:

Considérese un árbol binario que representa la expresión matemática (A * B) + ((C / D) ^ 3.5). Su representación en memoria es la siguiente (Cairó y Guardati, 2006: 196):

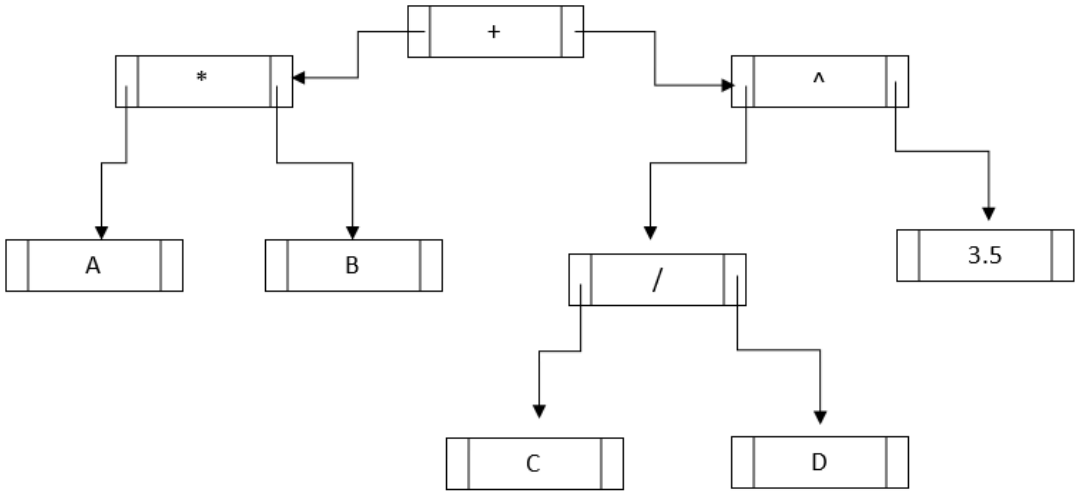
Figura 3.8. Representación de expresión matemática con un árbol binario



Fuente y elaboración: autoría propia, (2018)

Considerando las representaciones de un árbol en memoria que se mencionaron anteriormente, como un registro con 3 campos, ahora vamos a ver cómo se haría la representación de la misma expresión matemática (A * B) + ((C / D) ^ 3.5), con dichos registros.

Figura 3.9. Representación de expresión matemática en memoria con un árbol binario como un registro de 3 campos



Fuente y elaboración: autoría propia, (2018)

3.1.2. Definición de las operaciones con árboles binarios

Las operaciones de acuerdo con su empleo generarán estructuras de árboles; su recorrido comienza partiendo del Nodo Raíz, para ir por las ramas, nodos internos, hasta llegar a los nodos terminales.

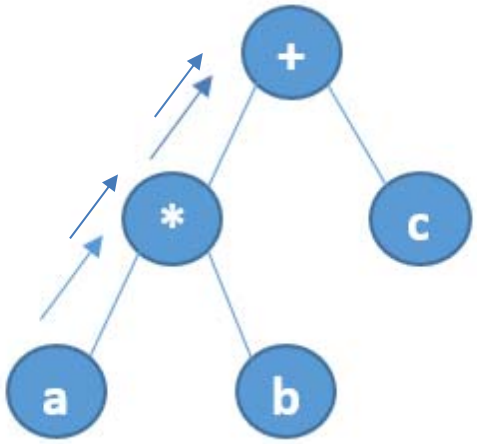
Las operaciones que se realizan sobre un árbol, permiten principalmente agregar o eliminar nodos, pero también hay otras operaciones que permiten hacer

recorridos sobre ellos. A continuación, se muestran las operaciones que se pueden realizar sobre los árboles:

- ✓ insert(): Insertar un elemento en un árbol
- ✓ delete(): Eliminar un elemento de un árbol
- ✓ isEmpty(): Comprobar si un árbol está vacío
- ✓ size(): Contar el número de nodos
- ✓ depth(): Calcular la altura de un árbol

Otro punto importante a revisar son los recorridos del árbol, en los cuales se puede visitar cada uno de los nodos del árbol, para almacenar o leer datos.

Figura 3.10. Árbol binario de una expresión matemática



Fuente y elaboración: autoría propia, (2018)

Con base en la imagen anterior, veamos los tipos de recorridos:

- ✓ printInOrder(): Recorrer el subárbol izquierdo en in-orden, visitar el nodo raíz, recorrer el subárbol derecho en in-orden.

  In-Orden a * b + c

- ✓ printPreOrder(): Visitar el nodo raíz, recorrer el árbol izquierdo en pre-orden, recorrer el subárbol derecho en pre-orden.

  Pre-Orden + * a b c

- ✓ printPostOrder(): Recorrer el subárbol izquierdo en post-orden, recorrer el subárbol derecho en post-orden, visitar el nodo raíz.

  Post-Orden a b * c +

3.1.3 Implementación de un árbol binario

A continuación, explicaremos por medio de imágenes y código de programación en lenguaje C, la forma de implementar algunas operaciones de un árbol.

Figura 3.11. Declaración de un árbol de enteros.

```
/*
Author: Germán Ignacio Cervantes González
Todos los Derechos Reservados FCA-UNAM (c)

Programa que representa un árbol de enteros.
Los valores menores van del lado izquierdo y los mayores del derecho.
*/
#include <stdio.h>

//Definición de un nodo
typedef struct Nodo
{
    int dato;
struct Nodo *izq;
struct Nodo *der;
}Arbol;

//inicializar el árbol con el valor de la raíz
//Declaración de la raíz del árbol
Arbol *raiz=NULL;
```

Fuente: autoría propia, (2018)

A continuación, revisaremos el código para estructurar una inserción en un árbol.

Figura 3.12. Inserción de un árbol

```
void inserta(int entero){
     Arbol *hoja;
     Arbol *ptr=NULL;
     int bandera=1;

     hoja=(Arbol *)malloc(sizeof(Arbol));
     hoja->izq=NULL;
     hoja->der=NULL;
     hoja->dato=entero;

     if(!isEmpty()){
        ptr=raiz;
        while(bandera){
          if(ptr->dato >= entero){
             if(ptr->izq == NULL){
                 ptr->izq=hoja;
                 bandera=0;
             }else ptr=ptr->izq;
          }else{//Si el dato del nodo es menor que el entero
              if(ptr->der == NULL){
                 ptr->der=hoja;
                 bandera=0;
             }else ptr=ptr->der;
          }
        }//While
     }//if(raiz !isEmpty())
     else raiz=hoja;
}


int isEmpty(){
   if(raiz==NULL) return 1;
   else return 0;
}
```

Fuente: autoría propia, (2018)

3.2. Grafos

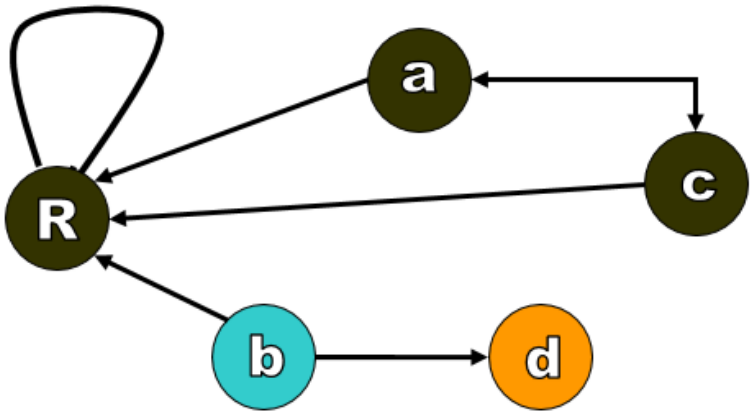
Imagina que tienes una serie de destinos a los cuales quieres llegar, como diferentes estados de la República mexicana, y quieres sabes cuál es la ruta más corta y más práctica para visitar todos los destinos en el menor tiempo y utilizando la menor cantidad de gasolina; éste tipo de problemas se pueden representar con grafos, en donde cada uno de los destinos son nodos y las rutas entre ellos son vértices en donde puedes manejar datos como, la distancia o la cantidad de gasolina que gastas de un punto a otro.

3.2.1 Definición del tipo de dato abstracto grafo

En palabras de Guardati (2007), los grafos son

> **Estructuras de datos no lineales, en las cuales cada elemento puede tener cero o más sucesores, así como cero o más predecesores. Dichas estructuras están formadas por nodos o vértices (representan información) y por arcos o aristas (relaciones entre la información).**

Figura 3.13. Representación genérica de un grafo



Fuente: autoría propia, (2018)

Para poner un ejemplo de lo anterior, consideremos la representación de una red de carreteras entre diferentes ciudades, cada una de las ciudades representaría un nodo (vértice) del grafo y las carreteras los arcos (aristas). A cada arco se asociaría información como la distancia entre ciudades. Los grafos ayudan a esquematizar dicha información, como la que se mencionó anteriormente y que se representa como la imagen 3.13.

Por otro lado, existen diferentes tipos de grafos, los cuales se clasifican de acuerdo a lo que podrás observar en la siguiente imagen:

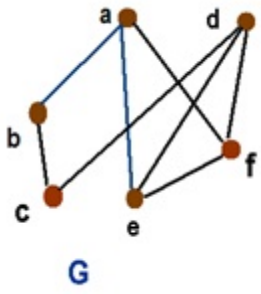
Figura 3.14. Clasificación de los tipos de grafos

Fuente: autoría propia, (2018)

Elaborado por: Michel, M. (2018)

Un grafo puede tener señalada una dirección o no. Cuando el grafo no señala ninguna dirección se llama grafo no dirigido, tal como se muestra en la siguiente imagen:

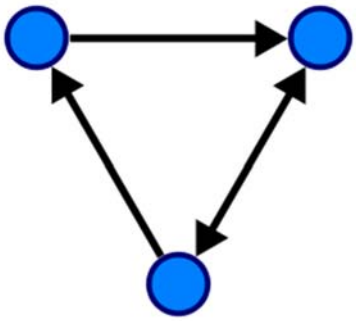
Figura 3.15. Grafo no dirigido



Fuente: Skriom (2012) recuperado de https://bit.ly/2INFkXR

En un grafo dirigido o también conocido como **dígrafo**, el conjunto de sus aristas tiene una dirección.

Figura 3.16. Grafo dirigido



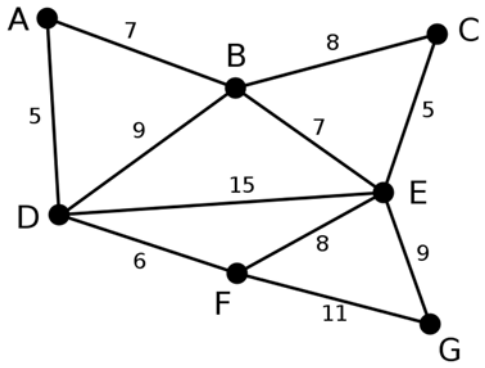
Fuente: Wikimedia (s/a) (2017) recuperado de: https://bit.ly/2G3K5e0

Los grafos ponderados[12] son muy útiles, por ejemplo, para medir las distancias en un mapa. En el caso de un repartidor de muebles que tiene que recorrer varias ciudades de México conectadas entre sí por las carreteras que hay entre la Ciudad de México, Guadalajara y Monterrey, su misión es optimizar la distancia recorrida (minimizar el tiempo, prever tráfico y atascos). El grafo correspondiente tendrá como vértices estas ciudades y como aristas la red de carreteras y la valoración, peso o ponderación será la distancia que hay entre ellas.

Para tal efecto, es preciso atribuir a cada arista un número específico, ponderación, peso o coste según el contexto, y se obtiene así un grafo valuado o ponderado.

[12] Se les llama ponderados porque las aristas tienen un valor.

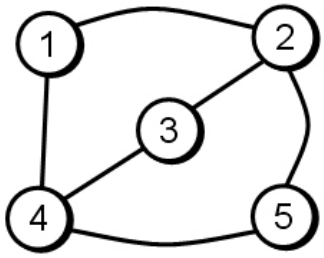
Figura 3.17. Grafo ponderado



Fuente: Drichel, A. (2008) recuperado de: https://bit.ly/2DQxCbn

Una de las formas de representar un grafo de cualquier tipo es a través de una matriz de adyacencias, ésta matriz puede representarse por medio de un arreglo de la siguiente forma.

Figura 3.18. Grafo representado como una matriz de adyacencias



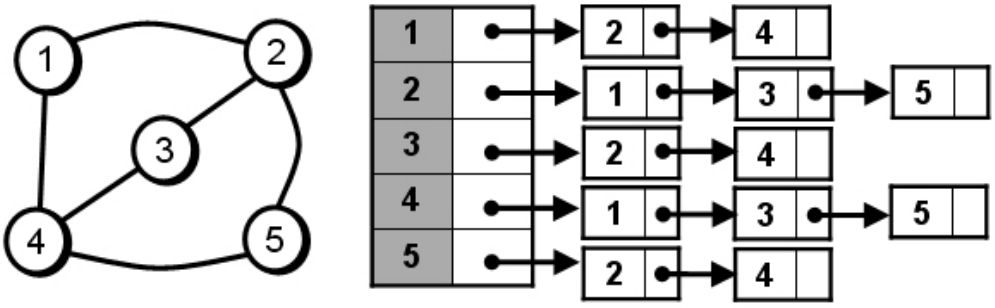
M	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	1
3	0	1	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

Fuente: Hebeb, (2015) recuperado de: https://bit.ly/2DQwXXD

En la imagen anterior se observa que cada uno de los índices del arreglo representa uno de los nodos numerados del 1 al 5; en la matriz se representa la conexión entre nodos con un número 1 y con 0, aquéllos que no tienen conexión.

Otra forma de representar un grafo de cualquier tipo es a través de una lista de adyacencias de la siguiente forma; en donde cada lista representa los nodos adyacentes a un nodo en particular.

Figura 3.19. Grafo representado como una lista de adyacencias



Fuente: Hebeb (2008) recuperado de: https://bit.ly/2DQwXXD

:

En la imagen anterior se representan las conexiones de cada uno de los nodos con una lista; por ejemplo, el nodo 4 tiene conexiones con los nodos 1, 3 y 5, por eso se forma una lista con dichas adyacencias y de la misma forma para el resto de los nodos.

3.2.2 Operaciones sobre un grafo

Las operaciones más básicas que debe tener un gafo [13], permiten agregar nodos y borrarlos, así como agregar aristas y borrarlas:

[13] Considera que las imágenes y códigos ejemplificados anteriormente te permitirán comprender estas operaciones.

- ✓ Init(): Iniciar un grafo.
- ✓ addNodo(): Agregar nodo o vértice
- ✓ addEdge(): Agregar arista
- ✓ removeNode(): Borrar nodo o vértice
- ✓ removeEdge(): Borrar arista

3.2.3 Implementación de un grafo

Con base en la representación que ya vimos de un grafo como una matriz de adyacencias, revisaremos cómo expresar esa matriz en código de programación con Lenguaje C y las operaciones para la generación de grafos; en este caso sólo veremos la operación init()

Figura 3.20. Declaración de un grafo no dirigido en Lenguaje C, por medio de una matriz de adyacencias

```c
#include <stdio.h>

//Definición de la matriz
int grafo[50][50];


init(){
      int n,i,j;
      char res,enter;
      printf("Cuántos nodos tiene tu grafo: ");
      scanf("%d", &n);

      for ( i = 1 ; i <= n ; i++ ){
        for ( j = 1 ; j <= n ; j++ ){
            if ( i == j ){
                grafo[i][j] = 0;

            }

            printf("\n Los nodos %d y %d son adyacentes ? (Y/N) :",i,j);
            scanf("%c", &res);
            if ( res == 'y' || res == 'Y' )
                grafo[i][j] = 1;
            else
                grafo[i][j] = 0;
            scanf("%c", &enter);

        }
      }
}
```

Fuente: autoría propia, (2018)

RESUMEN

Como vimos en este tema, existen estructuras de datos avanzadas que nos permiten realizar organizaciones más complejas de los datos y, por lo tanto, implementar algoritmos que se adapten a este tipo de organizaciones de datos, como en el caso de elementos dependientes entre sí o la definición de rutas para decidir entre diferentes opciones, este tipo de algoritmos se pueden utilizar para la solución de diferentes problemas, también este tipo de estructuras de datos son muy utilizadas en inteligencia artificial y para representar pensamientos complejos.

BIBLIOGRAFÍA

SUGERIDA

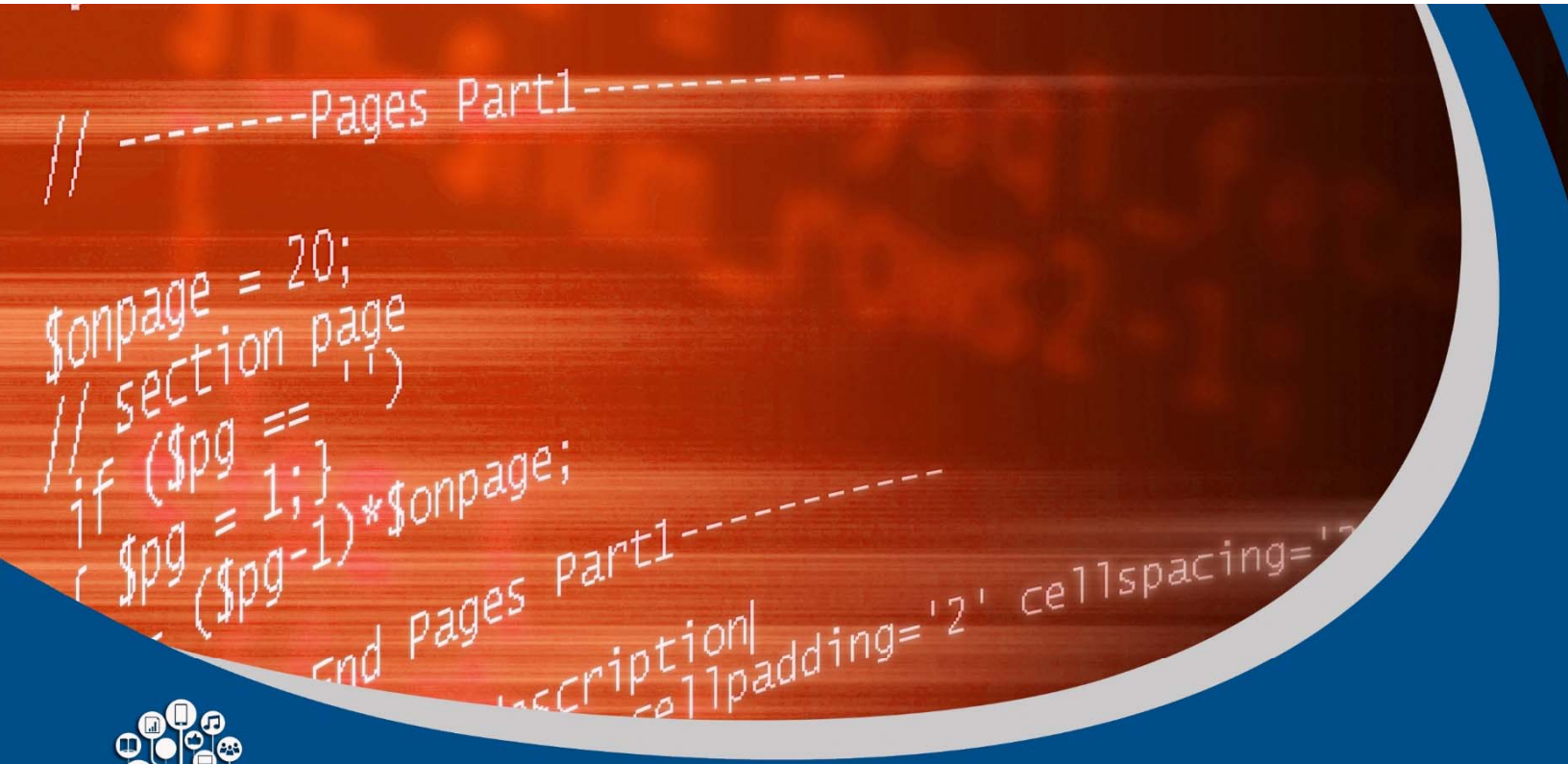
Autor	Capítulo	Páginas
Guardati, S. (2007)	7	-----
Lipschutz, S. (1998)	8	337 y ss.

Guardati B, Silvia (2007). *Estructura de datos orientada a objetos*. México: Pearson.

Lipschutz, Seymour (1988). *Estructura de datos*. México: McGraw-Hill.

Unidad 4

Métodos de ordenamiento



UNAM FCA
CED:GEC
Centro de Educación a Distancia y Gestión del Conocimiento

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

Al finalizar la unidad, el alumno identificará los diferentes métodos para la clasificación de datos, identificará características y los criterios para seleccionar el más adecuado a un conjunto de datos determinado.

TEMARIO DETALLADO

(12 horas)

4. Métodos de ordenamiento

4.1 Ordenamiento por Intercambio (Bubblesort)

4.2 Ordenamiento por inserción directa

4.3 Ordenamiento por selección

4.4 Método Shell

4.5 Ordenamiento rápido (Quick sort)

4.6 Criterios de selección del método de ordenamiento

INTRODUCCIÓN

Una de las operaciones más importantes en el manejo de datos es el ordenamiento. La ordenación (o sort en inglés), es reagrupar u organizar los elementos de un grupo de datos en una secuencia específica de orden (relación de orden), ya sea de mayor a menor, menor a mayor o, en caso de tratarse de otro tipo de datos como caracteres, ordenar por orden alfabético de manera descendente o ascendente. Muchas de estas operaciones las vemos en las hojas de cálculo como MS Excel, en páginas de internet y bases de datos. Nos centraremos en los métodos o algoritmos más populares, analizando la cantidad de comparaciones que suceden, el tiempo que demora y ejemplificando con pseudocódigo o código en lenguaje de programación; ya que estas características son las que nos permiten tomar la decisión de qué algoritmos usar; es decir, dependiendo de los recursos de cómputo que tengamos o el tiempo para realizar el ordenamiento.

Muchos lenguajes de programación ya incluyen funciones de ordenamiento optimizadas, por lo cual tal vez no sea necesario que llegues a programarlas en la práctica, pero el hecho de que se sepa cómo implementar diferentes métodos de ordenamiento, es una práctica indispensable para desarrollar habilidades en cualquier programador.

4.1. Ordenamiento por intercambio (*Bubble sort*)

Imagina que tenemos un conjunto de datos o elementos dentro de una lista o arreglo, de los cuales tomas dos de ellos y los introduces o aíslas en una burbuja inteligente, como si fueran los únicos en el mundo; al ser inteligente, dicha burbuja descubre cuál de los dos elementos es mayor y los intercambia [14], si es necesario, de manera que deja al menor del lado izquierdo y al mayor del lado derecho.

Con la micro operación explicada anteriormente, nuestra burbuja inteligente sólo ordenó dos elementos y de menor a mayor, sin embargo, dicha operación puede aplicarse varias veces (sobre un arreglo o lista de elementos) hasta dejar todos los elementos ordenados.

Hablando de un ordenamiento ascendente; el intercambio deja al menor elemento del lado izquierdo y va aumentando el valor hacia el lado derecho, de manera contraria, existe el ordenamiento descendente en donde el elemento mayor se coloca del lado izquierdo y va disminuyendo el valor hacia el lado derecho.

El ordenamiento por intercambio o *bubble sort (ordenamiento de burbuja en español)*, funciona de la siguiente manera:

[14] Este es el proceso más común de ordenamiento por intercambio, más adelante se explican otros.

Figura 4.1 Ordenamiento por *bubble sort*

Fuente: autoría propia, (2018)

Elaborado por: Michel, M. (2018)

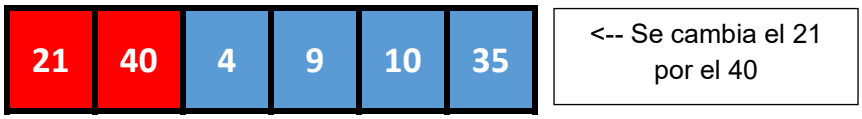
En relación a lo anterior, podemos decir que el algoritmo Bubble Sort consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados; recordemos que dichos elementos a ordenar estarán en una lista o arreglo, antes de realizar el ordenamiento. Este método también es conocido como **intercambio directo**.

Observa el siguiente ejemplo en el que se detalla el proceso:

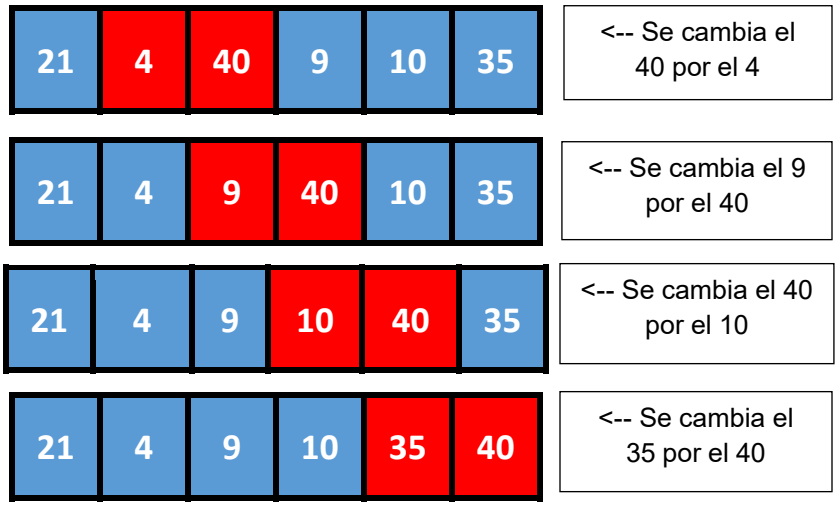
Tenemos un arreglo de 6 números, cada uno representa un empleado:

40	21	4	9	10	35

Llevamos a cabo la **primer pasada** y, para comenzar, se compara el 40 [15] con el 21, que son los dos primeros elementos; como el 21 es menor, debe quedar del lado izquierdo, entonces se intercambian.



Después comparamos el 40 con el 4 que son los dos siguientes elementos; debido a que el 4 es menor, debe quedar del lado izquierdo, entonces se intercambian también. Debido a que 40 es el número mayor de la lista, siempre habrá intercambio; hasta que quede al final de la lista (del lado derecho), como vemos en la secuencia ejemplificada en las siguientes imágenes.



Como puedes ver, el elemento mayor siempre es llevado hasta el final del arreglo en cada pasada.

Continuamos con la segunda pasada, llevando el mismo proceso que revisamos anteriormente. En esta pasada partiremos del número 21 tomándolo como el mayor pues el 40 ya está ordenado,

[15] Por ser el mayor.

4 21 9 10 35 40	<-- Se cambia el 21 por el 4
4 9 21 10 35 40	<-- Se cambia el 9 por el 21
4 9 10 21 35 40	<-- Se cambia el 21 por el 10
4 9 10 21 35 40	<-- Al comparar 21 con 35, vemos que el 21 es menor, por eso aquí no hay intercambio.
4 9 10 21 35 40	<-- Al comparar 35 con 40, vemos que el 35 es menor, por eso aquí tampoco hay intercambio.
4 9 10 21 35 40	<-- Éste es el resultado final de la segunda pasada.

Si te das cuenta, el arreglo ya está ordenado, pero para comprobarlo habría que hacer una tercera pasada, en la que notarás que no hay intercambios. Intenta hacerla para que lo compruebes.

Para la implementación de éste método vamos a definir que:

A = arreglo

N = tamaño

Observa la siguiente imagen:

Figura 4.2. Declaración de la función de ordenamiento por Bubble Sort

```
//La función recibe un arreglo de N elementos y se pasa el valor de N,
//que es el tamaño del arreglo.
int bubblesort(int A[],int N){

    int i,j,AUX;
    //El siguiente ciclo for va a dar N-1 pasadas sobre todo el arreglo;
    //es decir, el número de pasadas es igual al número de elementos menos 1,
    //para asegurar que el arreglo quede ordenado.
    for(i=0; i<(N-1);i++){
        //El siguiente ciclo for representa una pasada sobre el arreglo.
        //Sólo se intercambian los primeros N-i-1 elementos, porque sabemos
        //que los elementos del final, van quedando ordenados en cada pasada.
        for(j=0 ;j<(N-i-1);j++){ }

            if(A[j-1] > A[j]){ //si el elemento j-1 > j intercambio
                AUX = A[j-1];
                A[j-1] = A[j];
                A[j]=AUX;

            }
        }
    }
    return 1;
}
```

Fuente: autoría propia, (2018)

En la declaración de código anterior de la función *bubblesort()* utilizamos dos ciclos *for* anidados con los índices i y j para ir comparando y ordenando los elementos durante el recorrido del arreglo.

Como puedes observar en la implementación, aquí dimos N vueltas al arreglo, pero la implementación se puede mejorar si cada vez que das una vuelta, compruebas si el arreglo ya está ordenado; en ese caso ya no se darán más vueltas y se da por terminado el ordenamiento.

Este ordenamiento es el más fácil de implementar; sin embargo, si cuenta con un mayor volumen de datos se vuelve el ineficiente; ese es un rasgo que te permitirá saber si será el correcto a utilizar o no.

Así tenemos que, partiendo de un número (n) de elementos, en el método *Bubble Sort* tendremos comparaciones consecutivas entre pares de números de tal manera que:

- ✓ En la primera pasada tendremos (n-1) comparaciones
- ✓ En la segunda (n-2)
- ✓ Así sucesivamente hasta llegar a 2 y 1 comparaciones entre elementos dependiendo de tamaño del arreglo.

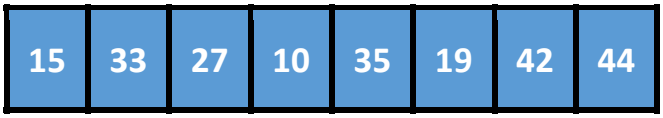
Por último, para identificar los casos en los que utilizaremos éste algoritmo, es necesario considerar que éste no es recomendado cuando se van a ordenar una gran cantidad de elementos, pues en el peor de los casos su complejidad es $O(n^2)$ (un lista en completo desorden, de mayor a menor), cuando se busca ordenar ascendentemente, y en el mejor de los casos es O(n) (la lista ya está ordenada).

4.2. Ordenamiento por inserción directa

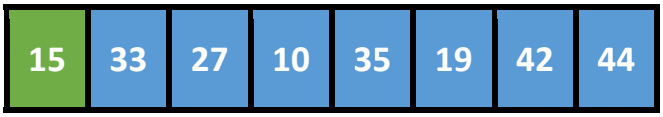
Imagina que tienes una baraja dividida en dos partes; una mitad está ordenada y la otra está desordenada; después tomas las cartas de la mitad desordenada y las insertas una por una de manera ordenada en la otra mitad, de manera que dicha mitad no pierda la característica de ser la ordenada. Ésta es la forma, a grandes rasgos de cómo funciona el método de ordenamiento por inserción directa.

Pero veámoslo, paso por paso, en un arreglo; es importante tomar en cuenta que, al inicio del ordenamiento, la parte izquierda ordenada no tiene ningún elemento, así que lo que se hace es ir formando la parte ordenada con los primeros elementos del arreglo, observa:

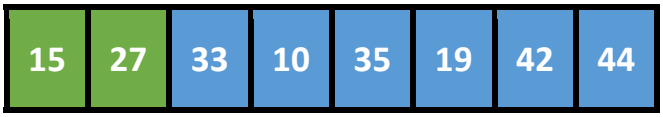
Tenemos el siguiente arreglo desordenado de elementos.



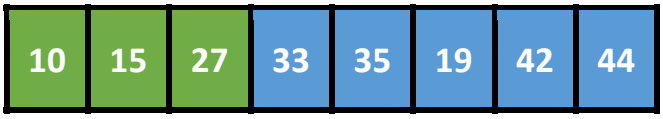
Primero comparamos los dos primeros elementos (15 y 33) y como se puede observar ya están ordenados de menor a mayor. Hasta aquí nuestra lista ordenada está compuesta sólo por el primer elemento (15).



Comparamos 33 y 27, y vemos que no están ordenados; entonces hacemos el intercambio. Después se compara el 27 con el 15; es decir, con la parte ordenada y vemos que no es necesario hacer nada, así se vuelve parte de la lista ordenada.



Ahora comparamos 33 y 10, se intercambian y después comparamos el 10 con nuestra lista ordenada, para que pueda formar parte de ella.



Este procedimiento se repite hasta que toda la lista esté ordenada.

En la siguiente imagen, vemos la implementación de éste método de ordenamiento en lenguaje C.

Figura 4.3. Declaración en Lenguaje C de la función de ordenamiento por inserción directa

```
void insertionSort() {

    int valorAInsertar;
    int posicionActual;
    int i;

    // Ciclo para operar sobre cada elemento del Arreglo.
    for(i = 1; i < MAX; i++) {

        // Asignación del valor que se va a comparar y a
        //insertar en la parte ordenada.
        valorAInsertar = intArreglo[i];

        // Posición actual sobre la que se hará la comparación
        posicionActual = i;

        // Con el siguiente ciclo se va moviendo el valor a ordenar
        while (posicionActual > 0 && intArreglo[posicionActual-1] > valorAInsertar) {
            intArreglo[posicionActual] = intArreglo[posicionActual-1];
            posicionActual--;
        }

        //En caso de que hubieramos hecho intercambios del elemento,
        //por último lo dejamos en la posición actual
        if(posicionActual != i) {
            intArreglo[posicionActual] = valorAInsertar;
        }

        printf("Interacción %d#:",i);
    } //for
}
```

Fuente: autoría propia, (2018)

Por último, para identificar los casos en los que utilizaremos éste algoritmo, es necesario considerar que éste no es recomendado cuando se van a ordenar una gran cantidad de elementos, pues en el peor de los casos su complejidad es $O(n^2)$ (una lista en completo desorden, de mayor a menor).

4.3 Ordenamiento por selección

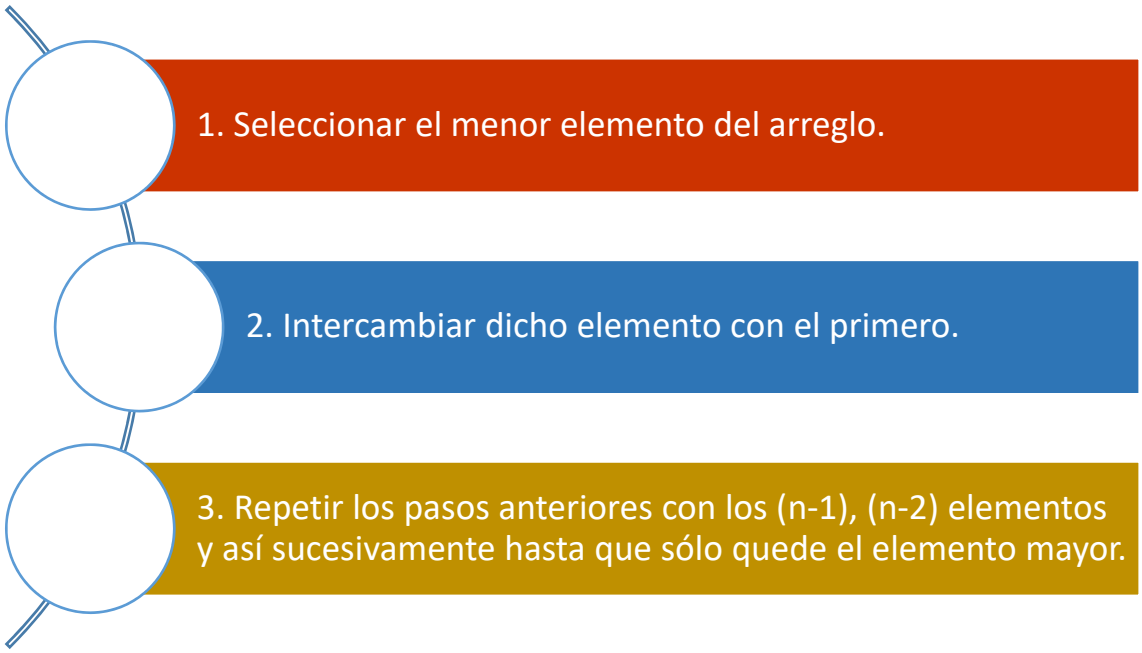
Imagina que tenemos una baraja desordenada, vamos a utilizar otra estrategia para lograr nuestro objetivo de ordenarla. Lo que hacemos es dividir el juego en dos partes, intentemos colocar la parte ordenada del lado izquierdo y la desordenada del lado derecho, por lógica al estar toda la baraja en desorden, el lado izquierdo estará vacío.

.

Ahora, nos vamos al juego de cartas y seleccionamos el elemento de menor valor de una sola figura, es decir; el 2 (recuerda que en una baraja no hay 1 ni 0), ese elemento se pone del lado izquierdo en la parte ordenada y así sucesivamente hasta que tengas toda la baraja ordenada del lado izquierdo.

En el contexto de un arreglo, el proceso de este algoritmo consiste en buscar el menor elemento en el arreglo e intercambiarlo por el elemento en la primera posición. Luego se busca el segundo elemento más pequeño del arreglo y se intercambia con el elemento de la segunda posición. El proceso continúa hasta que todos los elementos del arreglo hayan sido ordenados. Dicho método se basa en los principios que puedes observar en la siguiente figura:

Figura 4.4 Principios del ordenamiento por selección



Fuente: autoría propia, (2018)

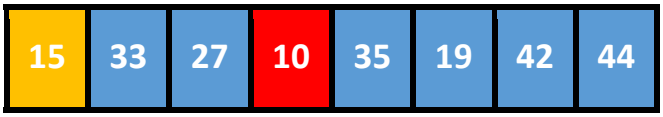
Elaborado por: Michel, M. (2018)

Lee con atención el siguiente ejemplo, en él podrás identificar de manera detallada los pasos que se deben seguir para utilizar éste método de ordenamiento:

Contamos con los siguientes elementos desordenados:

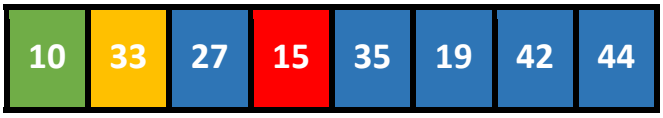
15	33	27	10	35	19	42	44

Seleccionamos la primera posición del arreglo, que es donde comenzaremos la lista ordenada; en el ejemplo el valor en esa posición es el 15. Recorremos toda la lista y nos damos cuenta que el elemento menor es el 10, como lo vemos en la siguiente imagen.

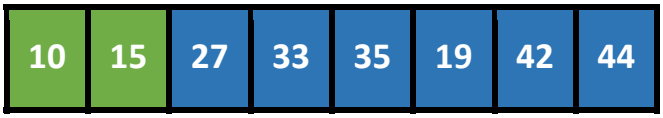


Una vez ubicados la primera posición del arreglo y el elemento menor de la lista, los intercambiamos de lugar, tal como verás en la siguiente imagen. Lo anterior da como resultado que el elemento más bajo quede en la primera posición de la lista.

El siguiente paso es ubicar la segunda posición del arreglo (color amarillo), así como el siguiente elemento menor (color rojo); y los intercambiamos como lo hicimos en el ordenamiento anterior.



Como resultado de los pasos anteriores, hemos logrado tener hasta el momento del proceso, 2 elementos ordenados en la lista; como se observa la siguiente imagen:



Este procedimiento se repite hasta que toda la lista esté ordenada.

En la siguiente imagen, vemos la implementación de éste método de ordenamiento en lenguaje C.

Figura 4.5. Declaración en Lenguaje C de la función para realizar el ordenamiento por selección.

```
void selectionSort() {

   int indiceMin, i, j;
   int temp;

   // Ciclo para operar sobre cada elemento del Arreglo.
   for(i = 0; i < MAX; i++) {

      //El elemento actual es el mínimo.
      indiceMin = i;

      //Buscar el elemento menor en todo el arreglo.
      for(j=i+1;j<MAX;j++){
         if(intArreglo[j] < intArreglo[indiceMin]){
            indiceMin=j;
         }
      }

      // El siguiente bloque, realiza el intercambio de los valores.
      if(indiceMin != i){
         temp = intArreglo[indiceMin];
         intArreglo[indiceMin] = intArreglo[i];
         intArreglo[i]=temp;
      }

   } //for
}
```

Fuente: autoría propia, (2018)

Por último, para identificar los casos en los que utilizaremos éste algoritmo, es necesario considerar que éste no es recomendado cuando se van a ordenar una gran cantidad de elementos, pues en el peor de los casos su complejidad es $O(n^2)$ (una lista en completo desorden, de mayor a menor).

4.4. Método *Shell*

Este algoritmo es parecido al de inserción directa y en términos generales, el funcionamiento del algoritmo del método *Shell* [16] consiste en dividir el arreglo (o la lista de elementos), en intervalos (o bloques) de varios elementos para organizarlos después por medio del ordenamiento de inserción directa; el proceso se repetirá pero con intervalos cada vez más pequeños, de tal manera que al final el ordenamiento se haga en un intervalo de una sola posición, similar al ordenamiento por inserción directa, la diferencia entre ambos es qué, para el final, en el método *Shell* los elementos ya están casi ordenados.

Existen varias formas de calcular el intervalo y eso puede mejorar la efectividad del algoritmo. A continuación explicaremos la secuencia original propuesta por Shell n/2, n/4 …, n/n; es decir 1 (dividir entre dos, hasta que el último intervalo sea 1), donde n es el tamaño del arreglo.

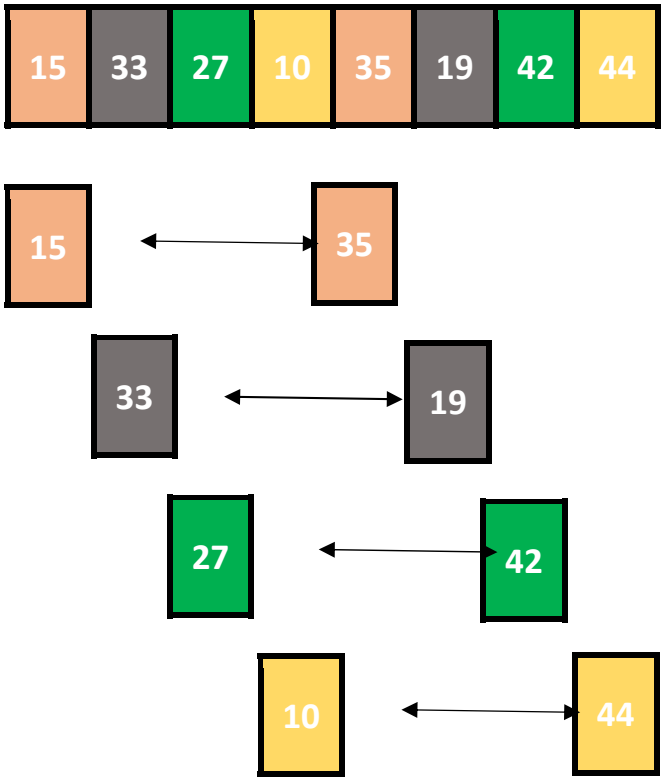
Observa el siguiente ejemplo, donde se aclara a detalle lo planteado anteriormente; en él podrás observar los pasos para llevar a cabo este tipo de ordenamiento sobre el arreglo desordenado de la siguiente imagen:

15	33	27	10	35	19	42	44

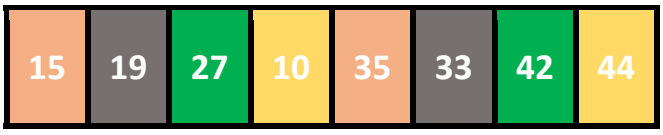
[16] Su nombre proviene de su creador ***Donald Shell*** y no tiene que ver en la forma como funciona el algoritmo.

En la imagen de abajo vemos cómo se realiza la ordenación; primero tomando como valor inicial de intervalo 4; es decir, como el número de elementos n es 8, tenemos que n/2=4.

Podemos decir también que en ésta primera interacción dividimos el arreglo en 4 subarreglos: {15,35}, {33,19}, {27,42} y {10,44}, los cuales ordenaremos por inserción directa.



En la siguiente imagen, vemos la forma en que quedaría el arreglo, después de ordenar los subarreglos.



Como se muestra en la imagen de abajo, ahora tomaremos un intervalo de 2; ya que en la siguiente interacción de la secuencia propuesta por Shell n/4=2, lo que genera dos subarreglos: {15, 27, 35, 42}, {19, 10, 33, 44}.

15	19	27	10	35	33	42	44

Al ordenar cada subarreglo por el método de inserción directa, obtenemos el siguiente resultado; en donde los únicos elementos que se intercambiaron por dicho método son el 10 por el 19, como se muestra en rojo en la siguiente imagen.

15	10	27	19	35	33	42	44

En la siguiente interacción, tomamos como intervalo 1 ya que n/8=1, lo que quiere decir que es la última interacción, en donde ordenaremos todo el arreglo por el método de inserción directa, obteniendo como resultado lo que se muestra en la siguiente imagen.

10	15	19	27	33	35	42	44

En la siguiente imagen, vemos la implementación de éste método de ordenamiento en lenguaje C.

.

Figura 4.6. Declaración en Lenguaje C de la función para el método de ordenamiento Shell.

```
void shellSort(){

   int intervalo, i, valorAInsertar, posicionActual;

    //Mientras el intervalo sea 1 o mayor a 1
    for (intervalo=MAX/2; (intervalo < MAX && intervalo > 0);
         intervalo=intervalo/2){

    //Se comprobará valor por valor, tomando en cuenta el
    //intervalo.
    for (i = intervalo; i < MAX; i++){
        valorAInsertar = intArreglo[i];
        posicionActual=i;

     // Con el siguiente ciclo comparamos los valores entre
     //intervalos y si es necesario ponemos valores mayores
     //a la derecha.
     while(posicionActual > 0 &&
           (intArreglo[posicionActual - intervalo] > valorAInsertar)
          ){
        intArreglo[posicionActual] = intArreglo[posicionActual - intervalo];
        posicionActual=posicionActual - intervalo;
     }
     //En caso de se haya hecho algún intercambio, ponemos el
     //valor a insertar en la posición actual.
     if(posicionActual != i) {
        intArreglo[posicionActual] = valorAInsertar;
     }

    }//for
   }//for

}
```

Fuente: autoría propia, (2018)

Por último, para identificar los casos en los que utilizaremos éste algoritmo, es necesario considerar que éste no es recomendado cuando se van a ordenar una gran cantidad de elementos, pues en el peor de los casos su complejidad es O(n) (una lista en completo desorden, de mayor a menor).

4.5 Ordenamiento rápido (*Quick sort)*

.

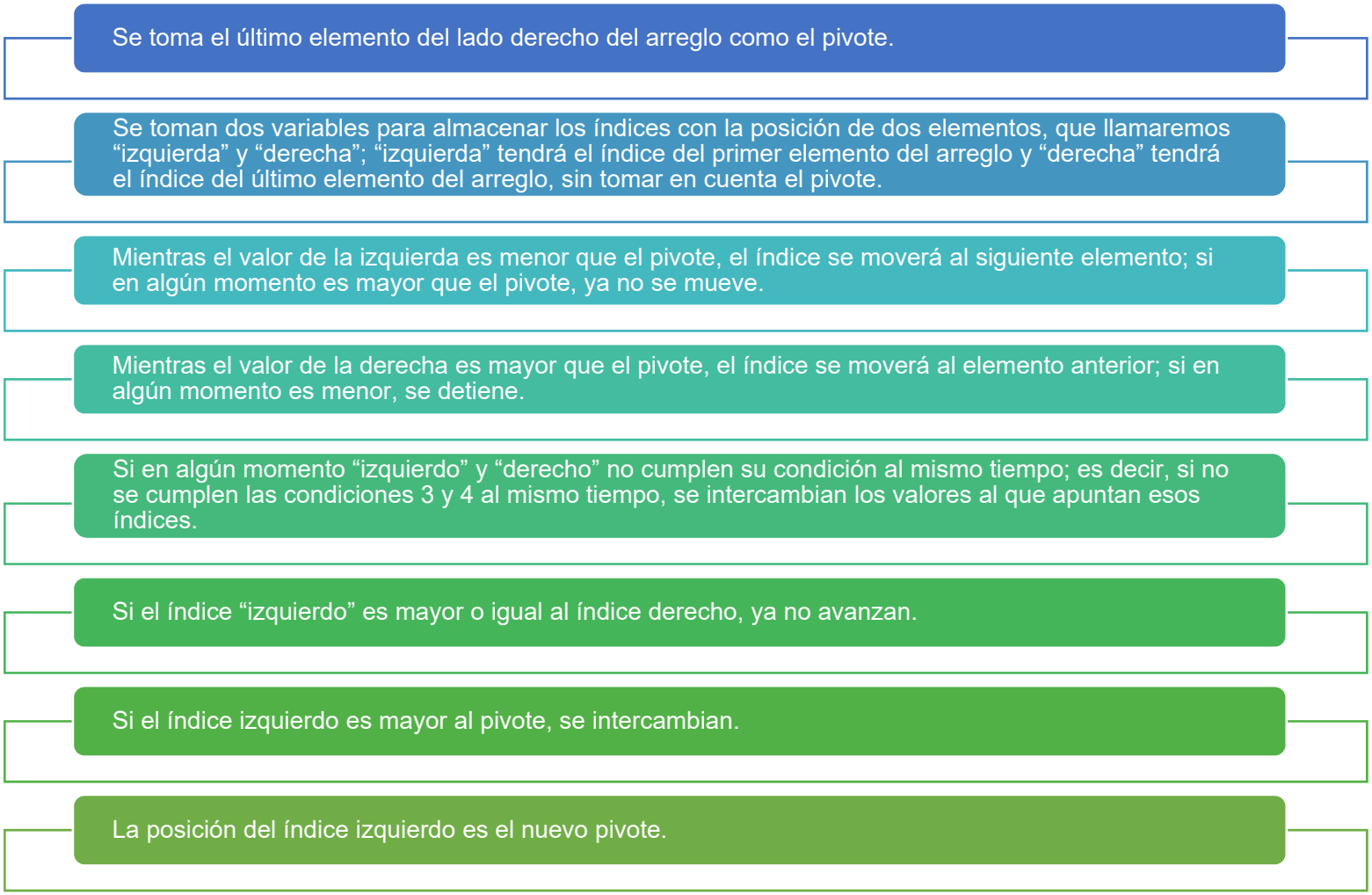
El algoritmo *Quick sort* o de ordenación rápida, es muy eficiente y se basa en dividir el arreglo (o lista) en arreglos más pequeños; para realizar ésta tarea se puede utilizar el método recursivo de programación, en donde una función se llama a sí misma para repetir un grupo de instrucciones determinado, de esta manera un problema complejo, se divide en subtareas más pequeñas pero que son idénticas entre sí, que al final dan como resultado la solución del problema complejo original.

En términos generales este algoritmo utiliza un valor del arreglo llamado pivote[17] para dividirlo en dos partes, una de las cuales tiene valores menores al pivote (parte izquierda) y la otra parte guarda valores mayores al pivote (parte derecha).

A continuación se describen los pasos para ordenar un arreglo o subarreglo de manera parcial; en la parte izquierda quedarán todos los valores menores al pivote y en la parte derecha quedarán todos los valores mayores al pivote; ésta es la función más simple que se aplicará de manera recursiva; a esta función o procedimiento lo llamaremos **Quicksort()**:

[17] Funciona como referencia para dividir el arreglo en 2 partes, de un lado quedarán los elementos menores a éste y del otro lado los mayores.

Figura 4.7. Proceso del Quicksort()



Fuente: autoría propia, (2018) **Elaborado por:** Michel, M. (2018)

La función anterior garantiza que todos los elementos menores quedan en la parte izquierda del nuevo pivote y todos los valores mayores quedan en la parte derecha del nuevo pivote.

A continuación se describen los pasos para el proceso que llama por primera vez a la función **Quicksort(),** de manera recursiva:

1. Aplicar **Quicksort**() al arreglo desordenado para dividirlo en 2 partes; la parte izquierda con los elementos menores al pivote y la parte derecha con los elementos mayores al pivote.
2. **Quicksort()** a la parte izquierda recursivamente.
3. **Quicksort()** a la parte derecha recursivamente.

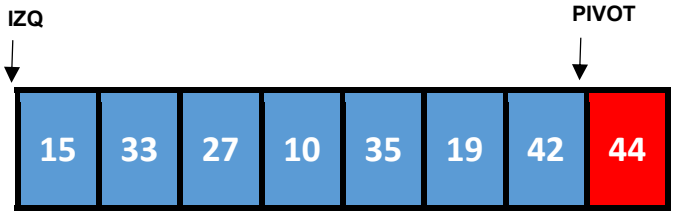
Así vemos que la función **Quicksort(),** que se describió paso por paso, se repite para cada una de las partes del arreglo; es decir, la parte izquierda del pivote y la parte derecha del pivote; esto quedará más claro cuando veas el código del Lenguaje C que se mostrará posteriormente.

Sería muy detallado ilustrar paso por paso el funcionamiento de éste algoritmo. A continuación, se muestran algunas interacciones, por ejemplo:

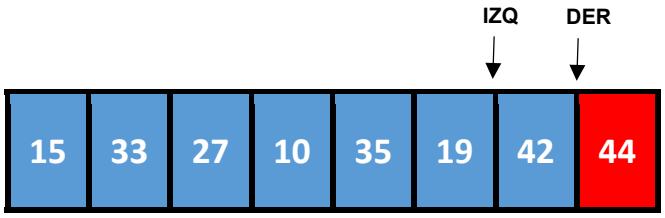
Tenemos el siguiente arreglo desordenado:

15	33	27	10	35	19	42	44

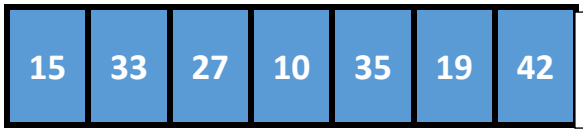
Como vemos en la imagen de abajo, primero tomamos como pivote el último elemento de la derecha (**paso 1)**; es decir, 44.



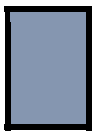
En la siguiente imagen vemos como queda el arreglo después de la primera interacción paso 2 al 6. Como todos los elementos son menores que el pivote no hay intercambios.



Como veremos en la siguiente imagen, para la siguiente interacción el arreglo se divide en 2 y se aplica el **Quicksort()** a ambos lados, en donde los elementos son menores que el pivote y donde los elementos son mayores que el pivote.

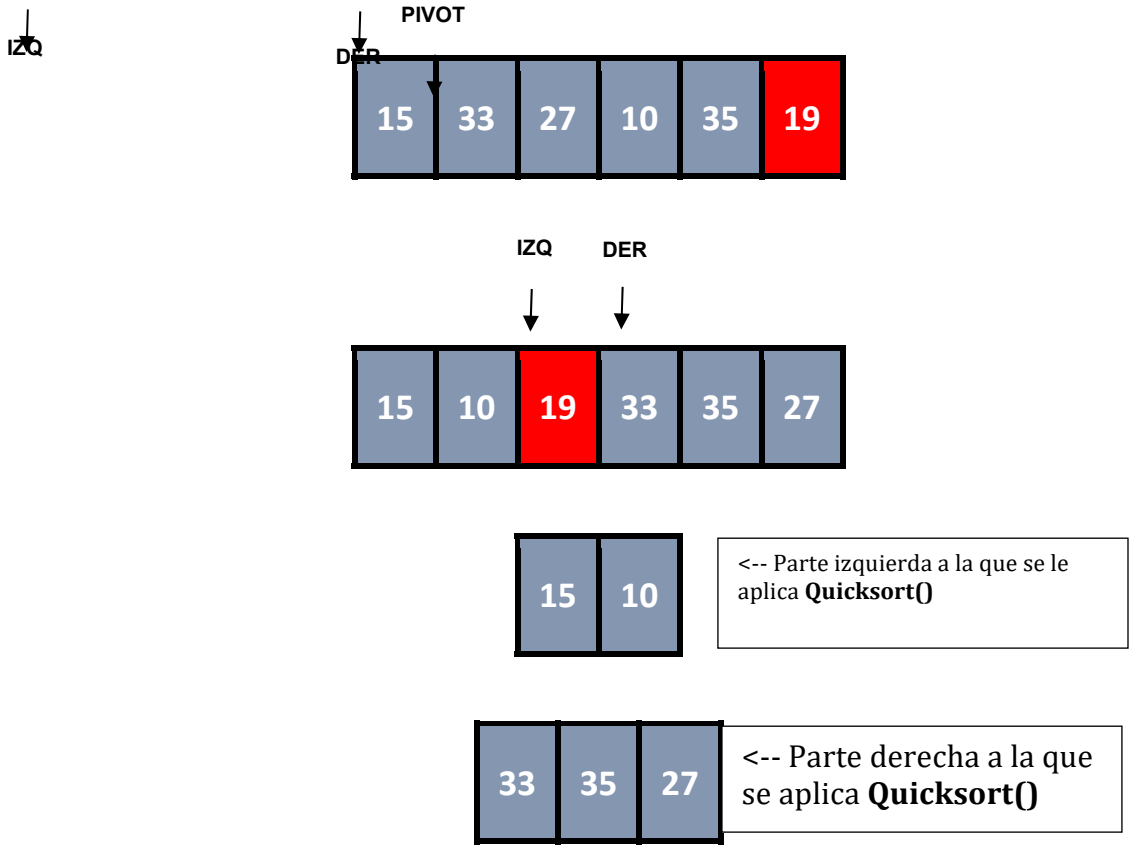


<-- Parte izquierda a la que se le aplica **Quicksort()**



<-- Parte derecha vacía, ya que no hubo elementos mayores al pivote

A continuación se muestra una de las interacciones posteriores en donde el pivote se intercambia, ya que existe un elemento mayor que él, en la posición del índice izquierdo, cuando se encuentra con el índice derecho, paso 1 al 7 de **Quicksort().**



En la siguiente imagen vemos la implementación de este método de ordenamiento en Lenguaje C.

Figura 4.8. Declaración en Lenguaje C de las funciones para el ordenamiento con el método Quicksort()

```
//Función para intercambiar valores en el arreglo.
void cambia(int num1, int num2) {
    int temp;

    temp = intArreglo[num1];
    intArreglo[num1] = intArreglo[num2];
    intArreglo[num2] = temp;
}
```

```
//Función para dejar los valores menores al pivote de lado
//izquierdo y los mayores del lado derecho.
int partir(int izq, int der, int pivote) {
   int izqApuntador = izq -1;
   int derApuntador = der;
   int noSalir=1;

   while(noSalir) {
      while(intArreglo[++izqApuntador] < pivote) {
         //No hacer nada
      }

      while(derApuntador > 0 && intArreglo[--derApuntador] > pivote) {
         //No hacer nada
      }

      if(izqApuntador >= derApuntador) {
         noSalir=0;
      } else {
         //Se cambian los elementos
         cambia(izqApuntador,derApuntador);
      }
   }//while
   cambia(izqApuntador,der);
   return izqApuntador;
}

void quickSort(int izq, int der){
   int pivote, partirPunto;

   if((der-izq) <= 0) {
      return;
   } else {
      pivote = intArreglo[der];
      partirPunto = partir(izq, der, pivote);
      //quicksort al lado izquierdo
      quickSort(izq,partirPunto-1);
      //quicksort al lado derecho
      quickSort(partirPunto + 1, der);
   }
}
```

Fuente: autoría propia, (2018)

Por último, para identificar los casos en los que utilizaremos éste algoritmo, es necesario considerar que éste no es recomendado cuando se van a ordenar una gran cantidad de elementos, pues en el peor de los casos su complejidad es O (n log2 n), una lista en completo desorden, de mayor a menor.

4.6. Criterios de selección de métodos de ordenamiento

Uno de los criterios para seleccionar el método que vamos a utilizar es el volumen de datos por procesar; es decir, que para mayor volumen hay que implementar métodos de ordenamientos rápidos y recursivos que consuman poca memoria.

Otro de los criterios para seleccionar el método que vamos a utilizar, es considerar si se requiere el ordenamiento interno o externo. Cuando hablamos de ordenamiento interno, se ejecuta completamente en memoria principal. Todos los objetos que se ordenan caben en la memoria principal de la computadora. En cambio, en el ordenamiento externo no cabe toda la información en memoria principal y es necesario ocupar memoria secundaria. El ordenamiento ocurre transfiriendo bloques de información a la memoria principal en donde se ordena el bloque y se regresa ya ordenado, a la memoria secundaria.

RESUMEN

En esta unidad partimos del método de ordenamiento *bubble sort*, ya que es un algoritmo de ordenamiento de datos simple y popular. Se utiliza frecuentemente como un ejercicio de programación, porque es relativamente fácil de entender y comprendimos cómo implementar el método de clasificación *shaker* en aplicaciones de estructuras de datos. La idea básica de este algoritmo consiste en mezclar las dos formas en que se puede realizar el método *bubble sort*. El algoritmo de la variable que almacena el extremo izquierdo del arreglo es mayor que el contenido de la variable que almacena el extremo derecho.

El análisis del método de la clasificación *shaker*, y, en general, el de los métodos mejorados y logarítmicos, son muy complejos. Respecto al método de ordenación por inserción directa, es el que generalmente utilizan los jugadores de cartas cuando ordenan éstas, de ahí que también se conozca con el nombre de método de la baraja, así como el método de selección directa, que se implementa en aplicaciones de estructuras de datos; de igual manera, el método *Shell sort* implica un algoritmo que realiza múltiples pases a través de la lista, y en cada pasada ordena un número igual de ítems.

El método *bubble Sort*, inserción y selección, tiene una complejidad normal de entrada n que puede resolverse en $n2$ pasos. Mientras que el método *Quicksort* tiene una complejidad mayor que incluye una búsqueda binaria, logarítmica y recursiva, por tanto, es el algoritmo de ordenamiento más rápido, cuya complejidad O(n log2 n), lo hace posiblemente el algoritmo más rápido.

BIBLIOGRAFÍA

SUGERIDA

Autor	Capítulo	Páginas
Martin, David (2007)	*Sorting Algorithm Animations*	http://www.sortingalgorithms.com/

Kruse, Robert L; Tondo, Clovis; Leung, Bruce (1997). *Data Structures & Program Design in C*. (2nd ed.). USA: Prentice Hall.

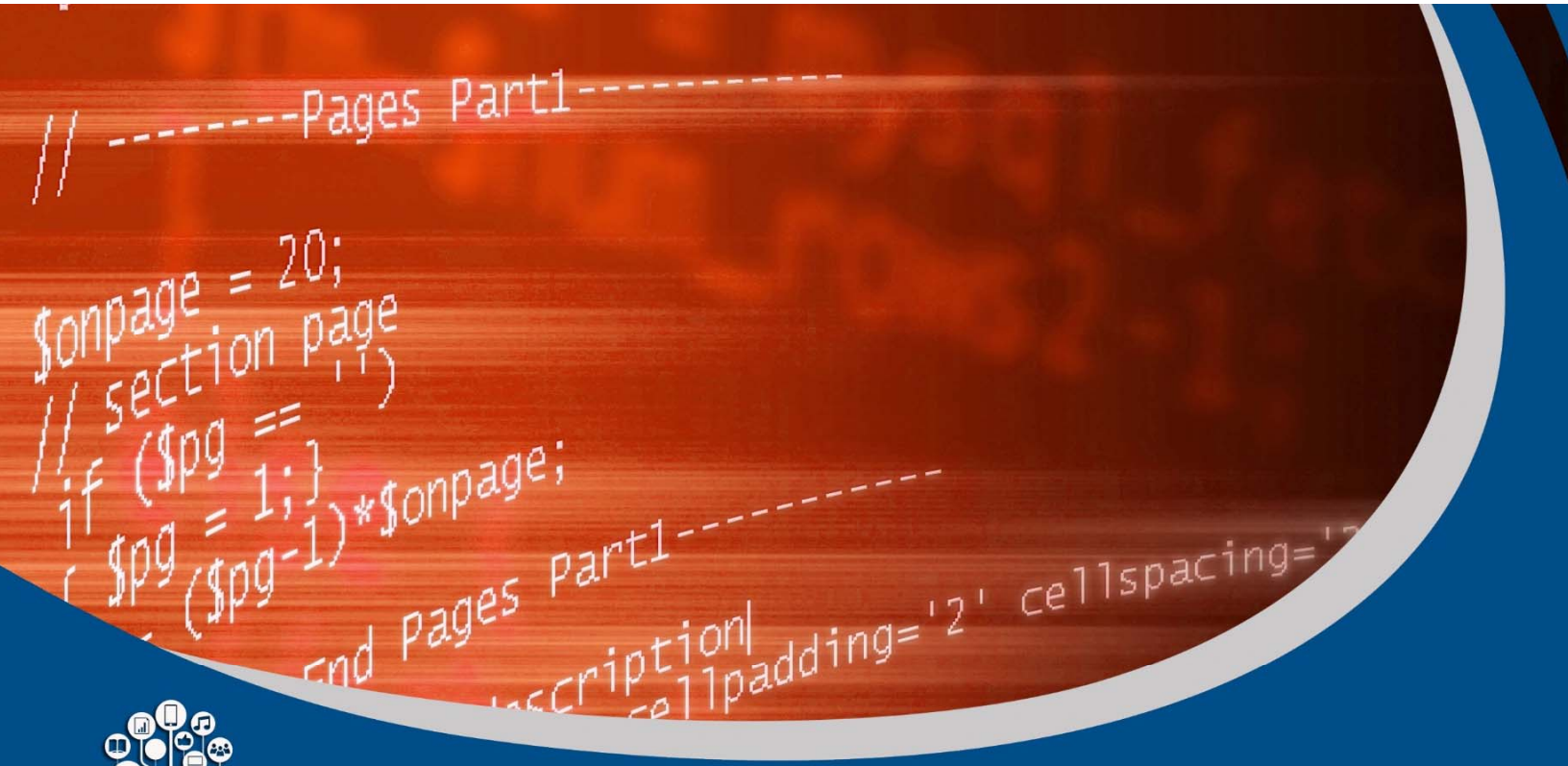
Joyanes Aguilar, L; Zahonero Martínez, I. (1998). *Estructuras de datos, algoritmos, abstracción y objetos (ejemplos en Pascal)*. México: McGraw-Hill.

Martínez, R; Quiroga, E. (2002). *Estructura de datos, referencia práctica con orientación a objetos*. México: Thomson Learning.

Cairó B, Osvaldo; Guardati, S. (2002). *Estructuras de datos* (2ª ed.). México: McGraw- Hill.

Unidad 5

Métodos de búsqueda



UNAM FCA
CED:GEC
Centro de Educación a Distancia y Gestión del Conocimiento

Plan 2012 2016 actualizado

OBJETIVO PARTICULAR

El alumno identificará y aplicará los métodos de búsqueda y podrá seleccionar el más adecuado para un conjunto de datos determinado.

TEMARIO DETALLADO
(12 horas)

5. Métodos de búsqueda

- 5.1. Búsqueda secuencial
- 5.2. Búsqueda binaria
- 5.3. Búsqueda mediante transformación de llaves (*hashing*)
  - 5.3.1. Funciones *hash*
  - 5.3.2. Resolución de colisiones
- 5.4. Árboles binarios de búsqueda

INTRODUCCIÓN

Una de las actividades más importantes de la informática es la búsqueda de los datos. El ejemplo más concreto lo vivimos a diario cuando tenemos que hacer consultas en cualquiera de los buscadores de internet.

La búsqueda de un elemento dentro de un arreglo es una de las operaciones más importantes en el procesamiento de la información pues permite la recuperación de datos previamente almacenados. El tipo de búsqueda se puede clasificar como interna o externa, esto de acuerdo al lugar en el que esté almacenada la información (en memoria o en dispositivos externos) y precisamente para establecer el método de búsqueda, se utilizan algoritmos, los cuales tienen dos finalidades:

1. Determinar si el elemento buscado se encuentra en el conjunto en el que se busca.
2. De ser así, hallar la posición en la que dicho dato se encuentra.

Específicamente en la búsqueda interna, tenemos como principales algoritmos la búsqueda secuencial, la binaria y la búsqueda utilizando tablas de *hash*, las cuales revisaremos a detalle a lo largo de la unidad.

5.1. Búsqueda secuencial

Imagina que tienes un amigo, que se adelantó a comprar los boletos del cine, él está formado en la fila de la taquilla; y tú empiezas a buscarlo, desde la primera persona que está formada en la fila; así tú observarás persona por persona, en orden, hasta encontrar a tu amigo, en caso de que no lo encuentres, habrás observado a todas las personas de la fila, hasta llegar a la última persona, sin haberlo encontrado. El método de búsqueda secuencial o lineal, consiste en recorrer y examinar cada uno de los elementos del arreglo hasta encontrar él o los elementos buscados, o hasta que se han mostrado todos los elementos del arreglo; así como lo vimos en el ejemplo anterior. Observa la siguiente imagen, en ella se muestra la implementación del algoritmo:

Figura 5.1

Declaración de la búsqueda secuencial de un elemento en Lenguaje C:

```
int busquedaLineal(int data) {

   int comparaciones = 0;
   int indice = -1;
   int i,encontrado=0;

   //Mientras no se llegue al final del arreglo y
   //no se encuentre el valor, seguir buscando
   for(i = 0;(i<MAX && encontrado==0);i++) {

   //Contar el número de comparaciones que se hace, antes de
   //encontrar el valor
   comparaciones++;

      //Si el dato buscado, coincide con el del Arreglo
      if(data == Arreglo[i]) {
         indice = i;
         encontrado=1;
      }
   }

   printf("Número de comparaciones: %d", comparaciones);
   return indice;
}
```

Fuente: autoría propia, (2017)

En el caso del programa de la imagen anterior, suponemos que el arreglo está ordenado, por eso al encontrar el valor terminamos la búsqueda; también se asume que en el arreglo no hay repeticiones; dichas características no son obligatorias para este algoritmo.

5.2. Búsqueda binaria

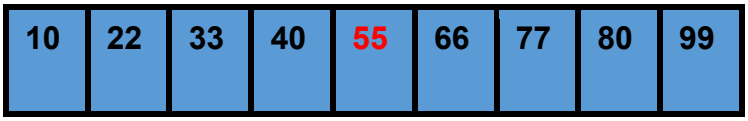
Imaginemos que buscamos a un joven de 20 años en una fila de personas, la cual está ordenada por edad, a saber, de la persona más joven a la más longeva, es importante mencionar que a todos les preguntaremos su edad. Para encontrar al joven de 20 años, dividimos la fila en dos partes, partiendo de la persona que está en medio; si resulta que ese primer sujeto tiene la edad que buscamos, ahí termina la búsqueda. Por otro lado, si no cuenta con 20 años, por ejemplo, que tuviera 40 años, comenzaríamos a indagar en la mitad de lado izquierdo, es decir, en el apartado en el que se encuentran las personas menores a 20 años, dividimos dicho apartado nuevamente a la mitad, indagamos la edad de los integrantes y así sucesivamente hasta localizar a la persona con la edad de nuestro interés: 20 años. ¿Complicado? En realidad, no lo es tanto y menos en los lenguajes de programación, más adelante revisaremos la manera de realizar la búsqueda anteriormente ejemplificada, conocida como búsqueda binaria.

La búsqueda binaria divide un arreglo ordenado en dos subarreglos más pequeños por su elemento medio y lo compara con el dato que se está buscando. Si coinciden, la búsqueda termina, pero, si el elemento es menor, entonces se busca en el primer subarreglo, por el contrario, si es mayor se buscará en el segundo subarreglo, como vimos en el ejemplo anterior.

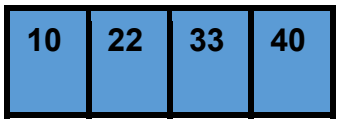
En el método de búsqueda binaria, es obligatorio que el arreglo de datos se encuentre **ordenado**, incluso es más eficiente que el de búsqueda secuencial ya que éste método se basa en ir comparando el elemento central del arreglo con el valor buscado. Si ambos coinciden, finaliza la búsqueda; si no ocurre así, el elemento buscado será mayor o menor en sentido estricto que el central del arreglo. Si el elemento buscado es mayor se procede a hacer la búsqueda binaria en la parte del subarreglo superior, si el elemento buscado es menor que el contenido de la casilla central, se debe cambiar el segmento a considerar al segmento que está a la izquierda de tal sitio central.

Observa el siguiente ejemplo, en él se detalla el proceso para realizar la búsqueda:

1. Se requiere buscar la clave **33** de un empleado por el método de búsqueda binaria en un arreglo de 9 elementos[18], los cuales son los que se encuentran a continuación.



2. El primer paso es tomar el elemento central que es **55** y dividimos en dos el arreglo. El elemento buscado **33** es menor que el central **55**, debe estar en el primer subarreglo:



3. El subarreglo anterior se divide nuevamente, quedando el **22** como elemento central, y como el que buscamos es mayor al intermedio, buscamos en el segundo subarreglo:

[18] Si el número de elementos es par, no tendrías un elemento exactamente a la mitad; es este caso el arreglo tendrá que dividirse en dos partes iguales y elegir cualquiera de los elementos que estén justo en medio, ya sea el de la derecha o el de la izquierda.

33	40

4. Se vuelve a dividir este subrango y como el elemento buscado coincide con el central **33**, podemos decir que hemos encontrado el elemento que buscábamos.

Si al final de la búsqueda todavía no hemos encontrado el elemento y el subarreglo por dividir está vacío {}, significa que el elemento no se encuentra en el arreglo.

En la siguiente imagen, vemos la implementación del algoritmo de búsqueda binaria en lenguaje C.

Figura 5.2. Declaración de la búsqueda binaria en Lenuaje C

```
int busquedaBinaria(int data) {
   int inferior = 0;
   int superior = MAX -1;
   int mitad = -1;
   int comparaciones = 0;
   int indice = -1;

   while(inferior <= superior) {

      comparaciones++;

      // Se calcula el punto medio
      mitad = inferior + (superior - inferior) / 2;

      // Si se encuentra el valor
      if(intArray[mitad] == data) {
         indice = mitad;
         break;
      } else {
         // Si el dato es mayor
         if(intArray[mitad] < data) {
            // El dato está con los números superiores
            inferior = mitad + 1;
         }
         // Si el dato es menor
         else {
            // El dato está con los números inferiores
            superior = mitad -1;
         }
      }
   }
   printf("Número de comparaciones: %d\n" , comparaciones);
   return indice;
}
```

Fuente: Cervantes, G. (2017)

Este método es más eficiente que el de búsqueda secuencial o lineal para casos de grandes volúmenes de datos, precisamente porque el dividir en dos el arreglo, nos permite hacer menos comparaciones entre elementos, ya que sólo comparas con los elementos menores o mayores al elemento que estás buscando.

Para mejorar la velocidad de ejecución de este algoritmo en un programa, se puede implementar en forma recursiva, utilizando en el programa una función que se llame a sí misma y repita el procedimiento de dividir el arreglo en dos arreglos más pequeños, uno izquierdo y otro derecho; del lado izquierdo quedarán los elementos

menores al elemento de en medio y del lado derecho los elementos mayores al elementos de en medio; ésta tarea se repetirá recursivamente para el subarreglo sobre el cuál se continúe la búsqueda del elemento, ya sea el izquierdo o el derecho.

La complejidad de éste algoritmo es de *log (2, N+1)*, que son el número de comparaciones que se realizarán, antes de encontrar el elemento que se está buscando o antes de descubrir que no está. El resultado de la operación *log (2,N+1)*, es muy inferior al número de comparaciones que se realizan en la búsqueda secuencial o lineal para casos de grandes volúmenes de datos.

5.3. Búsqueda por transformación

de llaves (hashing)

Imagina que tenemos un archivero, dentro del mismo hay carpetas con expedientes de empleados, para saber de qué empleado son los datos de una carpeta, en la pestaña del ésta ponemos el RFC de la persona a la que pertenecen los datos, el RFC se obtiene a partir de un algoritmo ya definido[19]; dicho RFC es equivalente a una llave que nos indica de quiénes serán los datos que encontraremos.

En informática, el término *hash* trata de la implementación de una función o método que permite generar claves o llaves que representen sin errores a un documento, registro, archivo, base de datos, etc. Esta técnica es útil para resumir o identificar un dato a través de la probabilidad, utilizando un método, algoritmo o función *hash*.

El método por transformación de claves o llaves (*hash*) consiste en asignar un índice (cómo el RFC del ejemplo) a cada elemento mediante una transformación de los elementos almacenados, esto se hace mediante una función de conversión conocida

[19] Si deseas revisar a detalle dicho algoritmo, puedes revisar el siguiente link http://bit.ly/2GlKwkH

como función *hash*. Hay diferentes funciones hash para transformar los elementos en llaves y el número obtenido será el índice que nos indique los datos que encontraremos. Lo ideal es que una función hash sea biyectiva; es decir, que a cada elemento o conjunto de elementos almacenados le corresponda un único índice y que a cada índice le corresponda el mismo elemento o conjunto de elementos, pero no siempre es fácil encontrar esa función.

Tabla 5.1. Ejemplo de conversión de datos en llaves

Nombres	Teléfonos
José Luis	5589343464
María	5577003423
Pedro	5598334388
Raúl	5589233390

Índice generado a partir de la llave	Teléfonos
78	5589343464
55	5577003423
12	5598334388
42	5589233390

En el ejemplo anterior la función hash es una caja negra, posteriormente veremos el ejemplo de el algoritmo de una función hash.

A una estructura de datos en donde asignamos un índice que hace referencia a los datos de un empleado, como en los ejemplos que hemos visto, se le llama también

arreglo asociativo, ya que se asocia el índice a los datos de una persona, lugar, cosa, etc. como en el ejemplo el RFC está asociado a los datos de un empleado.

5.3.1. Función hash

La función *hash*, tiene como entrada un conjunto de cadenas o números que se transforman en una cadena de longitud fija o en un número. La idea básica de la salida que genera una función hash, es que sirva como una representación compacta de la cadena de entrada. Por esta razón decimos que estas funciones resumen datos del conjunto **dominio**.
De las funciones *hash* más utilizadas podemos mencionar las siguientes:

- **Restas sucesivas**: En esta función se emplean claves numéricas entre las que existen huecos de tamaño conocido, obteniéndose direcciones consecutivas. Observa el siguiente ejemplo:

  Tenemos una clave que se le asigna a cada uno de los alumnos de una escuela, en donde la base es la clave de la carrera (0004) y la sede (0001), después a esa base se le asigna un número consecutivo que identificará a cada alumno. La diferencia entre la clave del alumno y la base de la clave nos da un valor que no se repite, por lo que se puede usar como índice para los datos de los alumnos.

000400010000 - 000400010000 = 0
000400010001 - 000400010000 = 1
000400010002 - 000400010000 = 2
….
000400010929 - 000400010000 = 929

- **Aritmética modular**: Esta función se basa en aritmética modular para generar un índice que es el residuo de la división del número a almacenar entre un número N prefijado. En éste caso el residuo si se podría repetir, ya que dos números pudieran tener el mismo residuo; es decir, dos elementos tendrían el mismo índice; a éste fenómeno se le llama colisión. Observa el siguiente ejemplo, en el que el número N prefijado es **99**

876 mod 99 = 84
1087 mod 99 = 97
765 mod 99 = 72
14567 mod 99 = 14

- **Mitad del cuadrado**: Con este método, se eleva al cuadrado el número al almacenar y se toman las cifras centrales. También puede haber colisiones. Ejemplo: elevamos al cuadrado varios números y tomamos los que se indican a continuación.

$56^2 = 3\mathbf{13}6 = 13$
$23^2 = 5\mathbf{2}9 = 2$
$12^2 = 1\mathbf{4}4 = 4$

- **Truncamiento:** Consiste en ignorar parte del número y utilizar los elementos restantes como índice. También hay colisiones. Ejemplo: sé que almacenaré números de 9 cifras, que al inicio son casi todas iguales, pero que varían en los tres últimos dígitos, entonces tomo las 3 últimas cifras como él índice.

415678**765** = 765
415675**431** = 431
415954**203** = 203

- **Plegamiento:** Se divide el número en diferentes partes y se realizan operaciones con ellos (se suman o se multiplican normalmente). También hay colisión. Ejemplo: partiendo del ejemplo anterior, si dividimos el número de 9 cifras en 3 tercios, podemos sumar los dos últimos tercios y del resultado tomar las últimas 3 cifras.

415678765 = 678 + 765 = 1443 = 443
415675431 = 675 + 431 = 1106 = 106
415954203 = 954 + 203 = 1157 = 157

5.3.2. Resolución de colisiones

Cuando hay un conflicto entre direcciones, por ejemplo, en las direcciones IPs de una red, se presenta un estado de colisión. Una colisión es la coincidencia de una misma dirección a dos o más índices diferentes. Cuando sucede este conflicto entre índices, entonces se busca el mejor método que ayude a resolver el problema.

Existen diferentes maneras de resolución de colisiones, una de ellas es crear un arreglo de apuntadores, donde cada apuntador señale el principio de una lista enlazada. De esta manera, cada elemento que llega a incorporarse, se pone en el último lugar de la lista de ese índice con una clave nueva. El tiempo de búsqueda se reduce considerablemente, y no hace falta poner restricciones al tamaño del arreglo, ya que se pueden añadir nodos dinámicamente a la lista.

Otro método de resolución de colisiones es el método de *prueba lineal,* que consiste en que una vez que se detecta la colisión se debe recorrer el arreglo secuencialmente a partir del punto de colisión, buscando otro índice vacío donde se pueda almacenar el elemento. El proceso de búsqueda concluye cuando el índice con la posición vacía es hallado. Por ejemplo, si la posición con el índice 97 ya estaba ocupada, el registro (415954203) el cual también le tocó el índice 97, es colocado en la posición 98, la cual se encuentra disponible. Una vez que el registro ha sido insertado en esta posición, otro registro que genere la posición 97 o la 98 es insertado en la posición disponible siguiente.

A continuación, resolveremos una colisión con aritmética modular que es uno de los métodos de *hash* que vimos anteriormente. Como se ha revisado hasta el momento, la operación módulo (signo % o mod) obtiene el residuo de la división de un número entre otro.

Por ejemplo: la operación, 5 módulo 2 nos da un resultado de 1, porque 5 dividido entre 2, nos da un cociente de 2 y un residuo de 1. Esto se puede escribir también como:

5 mod 2 = 1

5 % 2 = 1

El siguiente es un ejemplo de la misma operación, pero utilizándola para generar un rango de llaves. Partamos del hecho de contar con una tabla hash de 20 elementos, los cuales serán almacenados en la forma (índice, valor):

- **(1,20)**
- **(2,70)**
- **(42,80)**
- **(4,25)**
- **(12,44)**
- **(14,32)**
- **(17,11)**

El almacenamiento de los datos anteriores quedaría de la siguiente manera:

Tabla 5.2. Almacenamiento del rango de llaves

Llave	Hash	Índice del Arreglo
1	1 % 20 = 1	1
2	2 % 20 = 2	2
42	42 % 20 = 2	2
4	4 % 20 = 4	4
12	12 % 20 = 12	12
14	14 % 20 = 14	14
17	17 % 20 = 17	17
13	13 % 20 = 13	13
37	37 % 20 = 17	17

Como puedes apreciar en la tabla anterior, por medio del operador módulo se está garantizando que el valor del índice no exceda de 20, que es el tamaño de nuestro arreglo.

También podemos apreciar lo que puede suceder con nuestra técnica de *hashing*, es decir, generar un índice del arreglo que ya está siendo utilizado; en ese caso, se busca celda por celda la siguiente localidad vacía en el arreglo; es decir, con el método de prueba lineal que ya estudiamos. La tabla anterior quedaría de la siguiente manera.

Tabla 5.3.

Llave	Hash	Índice del Arreglo
1	1 % 20 = 1	1
2	2 % 20 = 2	2
42	42 % 20 = 2	**3**
4	4 % 20 = 4	4
12	12 % 20 = 12	12
14	14 % 20 = 14	14
17	17 % 20 = 17	17
13	13 % 20 = 13	13
37	37 % 20 = 17	**18**

En la siguiente imagen vemos la implementación en Lenguaje C de las funciones de inserta() y buscar() elementos con el método de aritmética modular. Ingresa al siguiente link para descargar la imagen que muestra la declaración completa de las funciones inserta() y busca() con el método de aritmética modular en Lenguaje C http://bit.ly/2tIe7Cp

5.4. Árboles binarios de búsqueda

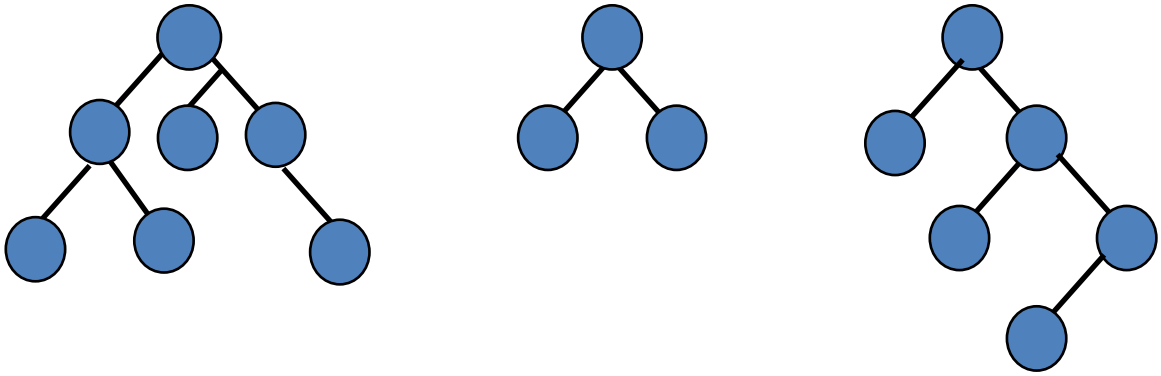
La búsqueda en árboles binarios es un método de búsqueda simple, dinámico y eficiente considerado como uno de los fundamentales en informática.

Recordemos que un árbol binario tiene las siguientes características:

- Es el que en cada nodo tiene a lo más un hijo a la izquierda y uno a la derecha.
- Es una estructura de datos jerárquica.
- La relación entre los elementos es de uno a muchos.

Observa las siguientes imágenes ejemplo:

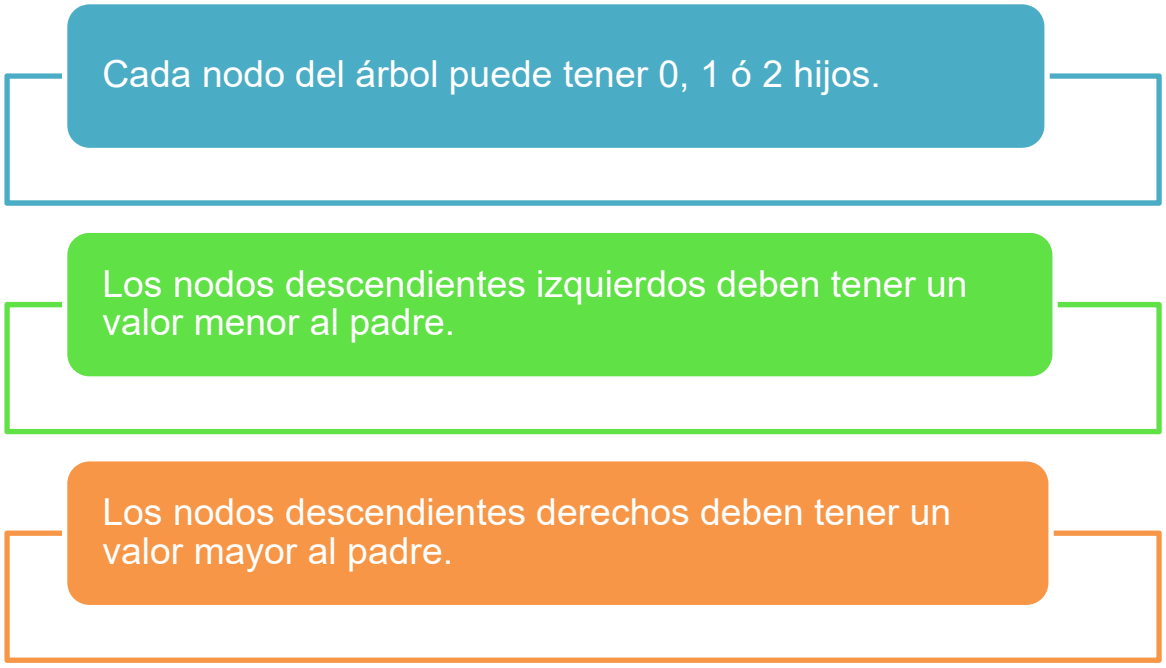
:



Elaborado por: autoría propia, (2018)

Para el caso de este tema, el **Á**rbol **B**inario de **B**úsqueda (ABB) es aquel que permite almacenar información ordenada, siempre y cuando se cumplan las siguientes reglas:

Figura 5.3. Reglas del ABB

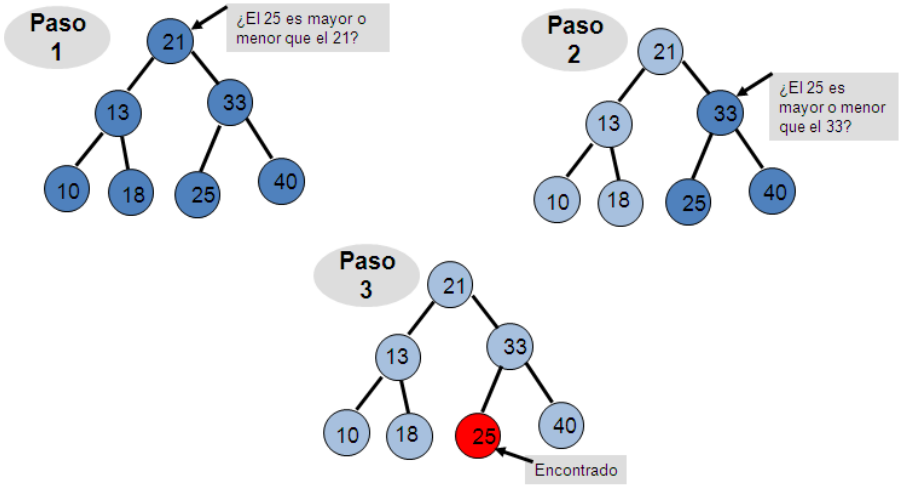


Fuente: Cervantes, G. (2018)

Elaborado por: Michel, M. (2018)

Pon atención a los siguientes ejemplos, en ellos se muestra un recorrido para buscar el nodo de número 25, observa los pasos a seguir:

Figura 5.4. Ejemplos de recorrido



Elaborado por: autoría propia, (2018)

En el ejemplo anterior, se requieren tres pasos para encontrar el nodo deseado. En caso de encontrar el elemento, se regresará el valor buscado (25 en el ejemplo) dependiendo de si se encontró en el nodo descendiente izquierdo o derecho. En caso contrario, se regresará *FALSE* (falso) para indicar que no se encontró el nodo, como se muestra en la función de C que se presenta a continuación.

Figura 5.5. Declaración de la función de búsqueda para un árbol binario en Lenguaje C

```
int Buscar(Arbol a, int dat) {
  pNodo actual = a;

  while(!Vacio(actual)) {
    if(dat == actual->dato) return TRUE;
    else if(dat < actual->dato) actual = actual->izquierdo;
    else if(dat > actual->dato) actual = actual->derecho;
  }
  return FALSE; /* No está en árbol */
}
```

RESUMEN

En esta unidad identificaste los diferentes métodos de búsqueda aplicados a las estructuras de datos para el desarrollo de sus aplicaciones, entendimos que todos los algoritmos de búsqueda tienen dos finalidades:

- Determinar si el elemento buscado se encuentra en el conjunto en el que se busca.
- Si el elemento está en el conjunto, hallar la posición en la que se encuentra.

En la búsqueda secuencial hay que recorrer y examinar cada uno de los elementos del arreglo hasta encontrar el o los elementos buscados, o hasta que se han mostrado todos los elementos del arreglo.

Para la búsqueda binaria, el arreglo debe estar ordenado. Este tipo de búsqueda consiste en dividir el arreglo por su elemento medio en dos *subarrays* más pequeños, y comparar el elemento con el del centro. Este método también se puede implementar de forma recursiva, siendo la función recursiva la que divide al arreglo en dos más pequeños.

Otro método de búsqueda es el de transformación de llaves *hash* que consiste en asignar un índice a cada elemento mediante una transformación del elemento, esto se hace mediante una función de conversión llamada función *hash*.

BIBLIOGRAFÍA

SUGERIDA

Autor	Capítulo	Páginas
Luis Joyanes Aguilar	Fundamentos de programación: algoritmos y estructura de datos	355p

Cairó, Osvaldo; Guardati, Silvia (2006). *Estructura de datos* (3ª ed.). México: McGraw-Hill.

Joyanes Aguilar, Luis (1990). *Problemas de metodología de la programación*. Madrid: McGraw-Hill.

---------- (2008). *Fundamentos de la programación*. Madrid: McGraw-Hill.

Rodríguez Baena, Luis; Fernández Azuela, Matilde y Joyanes Aguilar, Luis. (2003). *Fundamentos de programación: algoritmos, estructuras de datos y objetos* (2ª ed.). Madrid: McGraw-Hill.

Tenenbaum, Aarón (1993). *Estructuras de datos en C*. México: Prentice Hall.

Plan 2012

2016

actualizado