Universidad Nacional Autónoma de México
Facultad de Contaduría y Administración
Sistema Universidad Abierta y Educación a Distancia

Licenciatura en Informática

Programación (Estructura de Datos)

SUAyED

COLABORADORES

DIRECTOR DE LA FCA
Dr. Juan Alberto Adam Siade

SECRETARIO GENERAL
L.C. y E.F. Leonel Sebastián Chavarría

– – – –

COORDINACIÓN GENERAL
Mtra. Gabriela Montero Montiel
Jefe de la División SUAyED-FCA-UNAM

COORDINACIÓN ACADÉMICA
Mtro. Francisco Hernández Mendoza
FCA-UNAM

– – – –

AUTORES
Mtro. Juan Manuel Martínez Fernández
Lic. Ramón Castro Liceaga

DISEÑO INSTRUCCIONAL
L.P. Joel Guzmán Mosqueda

CORRECCIÓN DE ESTILO
Mtro. Francisco Vladimir Aceves Gaytan

DISEÑO DE PORTADAS
L.CG. Ricardo Alberto Báez Caballero
Mtra. Marlene Olga Ramírez Chavero
L.DP. Ethel Alejandra Butrón Gutiérrez

DISEÑO EDITORIAL
Mtra. Marlene Olga Ramírez Chavero

OBJETIVO GENERAL

Al finalizar el curso el alumno será capaz de entender la abstracción; implantar, en un lenguaje de programación, las estructuras de datos fundamentales y avanzadas y realizar ordenamientos y búsquedas.

TEMARIO OFICIAL
(64 horas)

	Horas
1. Fundamentos de las estructuras de datos	8
2. Estructuras de datos fundamentales	16
3. Estructuras de datos avanzadas	16
4. Métodos de ordenamiento	12
5. Métodos de búsqueda	12

INTRODUCCIÓN

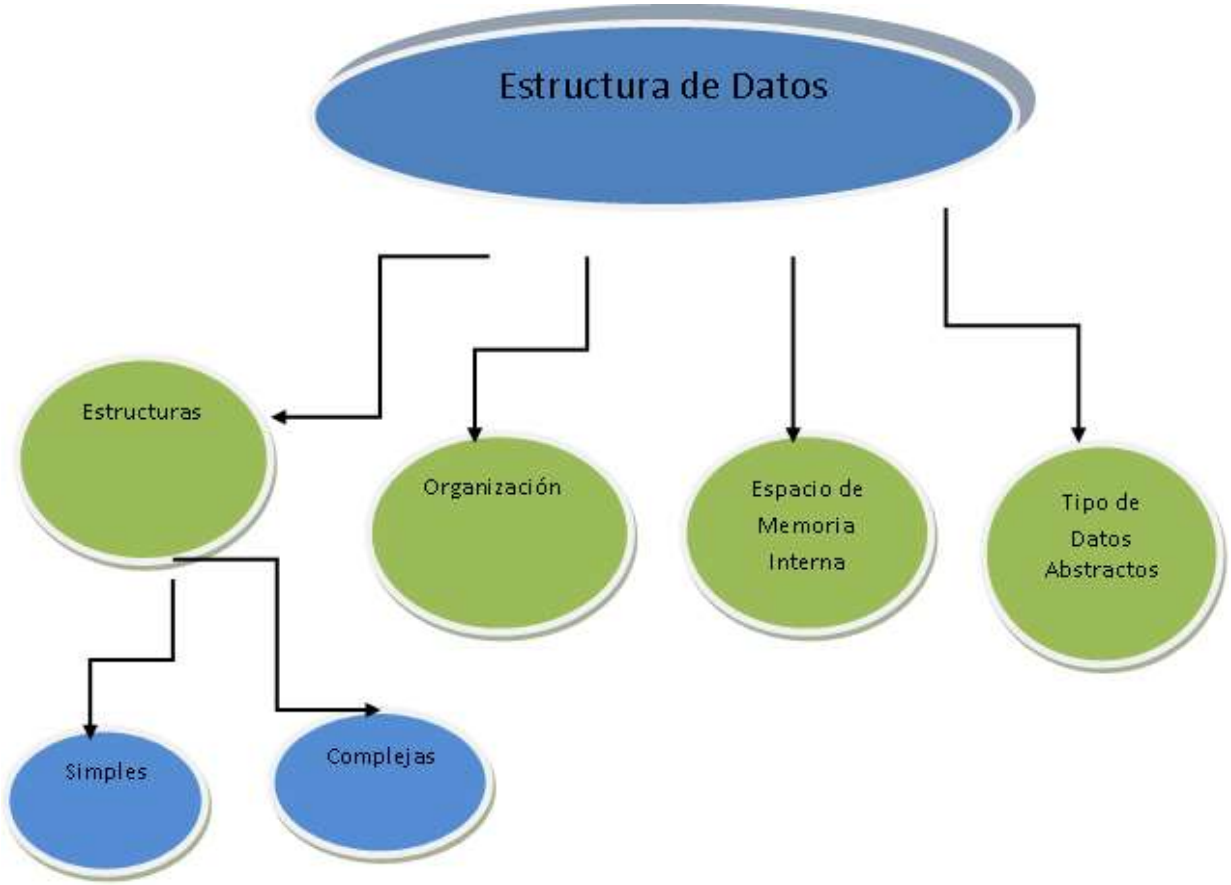
Las computadoras fueron diseñadas como una herramienta mediante la cual se puede realizar operaciones de cálculo complicadas en un tiempo mínimo. La mayoría de sus aplicaciones son las de almacenamiento, clasificación y acceso de grandes cantidades de información, como por ejemplo el caso de un buscador en Internet.

La información que se procesa en la computadora es un conjunto de datos que pueden ser simples o estructurados. Los datos simples son aquellos que ocupan sólo una localidad de memoria, mientras que los estructurados son un conjunto de casillas de memoria a las cuales hacemos referencia mediante un identificador único.

Debido a que por lo general tenemos que tratar con grandes volúmenes de datos, y no con datos simples (enteros, reales, booleanos, etc.) que por sí solos no nos dicen nada, ni nos sirven de mucho, es necesario tratar con estructuras de datos adecuadas a cada necesidad.

Las estructuras de datos son objetos con los cuales se representa el manejo y ubicación de la información en la memoria interna de la computadora. Se explican a través de modelos con los cuales se interpreta el manejo de la información en la memoria interna de la computadora, por lo cual se administra el espacio en dicha memoria.

ESTRUCTURA CONCEPTUAL



Unidad 1.

Fundamentos de las estructuras de datos

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

Los tipos de datos complejos se componen de varios tipos de datos simples ya sean del mismo tipo o de distintos, según las necesidades, pero aquellos se dividen en estáticos y dinámicos. En esta unidad estudiaremos los diferentes tipos de datos, especialmente los abstractos y las estructuras más simples alojadas en la memoria principal que se estudian por dos razones fundamentales: la primera porque de ellas se forman otras estructuras más complejas y, la segunda, porque varios compiladores actualmente tienen incluidos los tipos de datos estándar.

1.1. Definición de estructuras de datos

La organización de la información en la memoria interna de la computadora (*Random Access Memory*, RAM) se representa con estructuras de datos y el contenido de las páginas se representa con tipos de datos simples con los cuales se puede acceder, manejar y almacenar la información.

Definición de estructuras de datos

> Es la representación conceptual de la organización interna de los datos en la memoria interna de la computadora con el fin de optimizar el empleo del espacio de dicha memoria. (Cf. Joyanes, 1996, p. 465)

Los Tipos de Datos son objetos que representan tipos de información determinada almacenada y manejada en las páginas de memoria interna de la computadora y 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, que se comparte con otros procesos en fracciones de segundo; si no se sabe administrar dicho espacio, tenemos un desperdicio para otra aplicación, es decir, se desaprovechará la capacidad y potencial de la computadora. Baste un ejemplo, si empleamos en una localidad de memoria para almacenar el número uno 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.

Si el programador o el usuario no sabe desde el inicio cuál tipo de dato se debe emplear para una variable en una aplicación, entonces puede aplicar otros tipos de datos como el *variant* que ofrece el compilador de Visual Basic, aunque no es recomendable, ya que emplea demasiada memoria interna al manejar el dato como de un formato único pero de gran tamaño, y el programa empleará más recursos de la computadora y se podrá garantizar su desempeño para grandes programas con gran número de líneas de código. La administración del espacio en la memoria interna de la computadora también es aplicable a la memoria externa así como a los demás tipos de memoria empleados en la actualidad, en virtud de emplear los mismos mecanismos para leer y manipular la información de los periféricos de la computadora hacia la memoria RAM.

Los tipos de datos ofrecidos por defecto en los compiladores son llamados *estándar* o *primitivos*. Estos los abordaremos más adelante.

Ahora estudiaremos los datos simples en su forma conceptual y su forma de representación en la memoria interna de la computadora. La manera de concatenar los ítems (elementos) para integrar estructuras más complejas con las cuales realizar alguna aplicación en particular. Las estructuras simples no pueden contener a otras estructuras de información.

Los tipos de datos manejados por la computadora básicamente son dos, los *numéricos* y los *alfanuméricos*. Los primeros están representados por los números naturales y reales (enteros y fraccionarios) los cuales tienen ciertos rangos, en los cuales su capacidad de alojar información está representada en primer instancia con un entero (mantisa) y un exponente (fracción) que en notación científica sería $1000 = 10 * 10^2$ o 10 E 3. En el ámbito de la computación; además de considerar la mantisa y la fracción, hay que representar el signo, que para el positivo se usa el cero (0) y los negativos se representan con el número uno (1).

Las estructuras de datos son 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 las páginas de memoria interna (memoria RAM) de la computadora para después ir construyendo estructuras más amplias.

Los tipos de datos están íntimamente relacionados con las estructuras de datos, ya que los tipos los ofrecen los compiladores actuales, posteriormente se les asignarán nombres a variables, y las estructuras son organizaciones de datos que conforman estructuras con propiedades y formatos englobados bajo identificadores asignados por el usuario para emplearse en estructuras de programación y así ofrecer una mayor versatilidad que imponen las aplicaciones actuales del mundo real.

1.2. Tipos de datos

Al hablar de **tipos de datos**, nos referimos a los atributos o características de los datos, que le indican a la computadora o al programador la clase de datos que se van a procesar. Esto implica que los datos poseen ciertas restricciones, por ejemplo qué valores pueden tomar y qué operaciones se pueden realizar para su procesamiento; es decir, que existe una estrecha relación entre los lenguajes de programación y los tipos de datos, ya que una forma de instruir y proporcionar información a una computadora es precisamente a través de un 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 veremos cómo se relacionan diferentes y actuales lenguajes de programación, con el manejo de las estructuras de datos.

Los datos de tipo *carácter* se pueden unir y formar una cadena y después otro tipo de estructuras llamadas arreglos. Los de tipo entero pueden ser convertidos en su punto decimal que estará en posiciones flotantes y no de manera fija como en el entero. En el caso de Visual Basic hay otro tipo de dato particular como el *Fix* cuyo punto flotante está fijo y el número de decimales se determinará como constante.

Los tipos de datos los podemos abordar como: *tipos de datos simples* ó *estándar,* que son un conjunto de datos más amplio que los tipos de datos *primitivos,* que son datos necesarios para formar estructuras de datos; además de los datos *definidos por el programador*, también conocidos como datos incorporados . Veamos de qué se trata cada uno.

Tipos de datos estándar Los tipos de datos estándar son aquellos que 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 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. Esta discusión la tocaremos más adelante.

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).

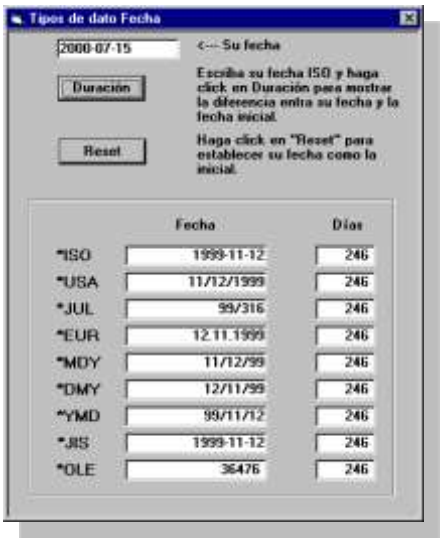
Definidos por el programador Este tipo de datos nos ayuda a delimitar los datos que podemos manejar dentro de un programa y nos sirve como una manera segura de validar la entrada/salida de éste.	
Subrango	En este tipo de datos delimitamos el rango, menor y mayor, de los posibles valores que puede tomar una variable, de esta manera podemos asegurar que la entrada o salida de nuestro programa esté controlada. 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.

Como ejemplo de esta jerarquía de datos, podemos mencionar los siguientes tipos de datos en el lenguaje Visual Basic.

String	Datos que pueden tener texto o cualquier carácter.
Integer	Datos que pueden tener cualquier número entero, o sea, no tiene punto decimal. Puede tener valores desde –32,768 hasta 32,767.
Long integer	Puede tener cualquier número entero, desde –2,147,483,648 hasta 2,147,483,647.
Single-precision (floating point)	Número con un máximo de seis (6) lugares decimales.

Double-precision (floating point)	Número con un máximo de catorce (14) lugares decimales.
Variant	Este tipo de datos es específico de Visual Basic y puede tener cualquier tipo de datos, pues deja que el lenguaje encuentre la mejor forma de guardar datos. Por esa razón, toma más memoria y hace los programas más lentos que si se usan los otros tipos de datos.
Currency	Otro tipo de "*floating point*". Puede tener valores desde –922 trillones hasta 922 trillones.
Bolean	Tiene sólo los valores True (cierto) o False (falso).
Byte	Tiene números enteros desde 0 a 255.

Otro tipo de datos de este lenguaje, son los de tipo fecha *(Date),* que pueden tener varios formatos, con o sin separador, como puede ser DDMMYY (día, mes y año) con sus diferentes combinaciones como AAAAMMDD (año, mes y día).



La definición del tipo booleano o lógico no es soportada o conocida en todos los lenguajes de programación. El tipo de dato lógico o booleano representa valores de lógica binaria; es decir, dos valores que se expresa en falso o verdadero. Una

constante booleana o lógica, acepta el valor falso o verdadero (*false* o *true*), que se representan con 0 y 1 respectivamente, en muchos lenguajes de programación. El valor de una constante booleana no cambia durante la ejecución del programa.

Una variable booleana o lógica también acepta solamente uno de dos valores: verdadero (*true*) o falso (*false*), pero el valor en cuestión puede cambiar durante la ejecución del programa.

En el siguiente ejemplo se muestra la implementación de una variable de tipo *Boolean* en lenguaje Visual Basic que almacena un único parámetro de tipo *sí / no*.

```
Dim runningVB As Boolean
' Checa si un programa es ejecutado en Visual Basic.
If scriptEngine = "VB" Then
   runningVB = True
End If
```

En otros casos, 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:

(3>2)= 1 = verdadero

(7>9)= 0 = falso

En el Lenguaje C, como no posee ninguna implementación primitiva del tipo booleano, para declarar variables de este estilo, hace falta definirlo previamente. La manera de hacerlo es añadir entre las declaraciones de tipos la siguiente instrucción:

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

Ésta define el tipo booleano y asigna a sus elementos *FALSE* y *TRUE*, los valores 0 y 1 respectivamente.

En Visual Basic, los booleanos o de tipo lógico son de tamaño de un *byte* y sólo podrán contener un valor, ya sea un cero o un número uno; es decir falso o verdadero, y son muy útiles en programación para asignar un valor inicial a una “bandera” en un programa de computadora, lo que podrá cambiar dicho valor y entonces el programa realizará otras acciones.

Operandos	Operador	Operación	Resultado
35,9 (enteros)	>	35 > 9	verdadero
35,9 (enteros)	<	35 < 9	falso
35,9 (enteros)	= =	35 = = 9	falso
35,9 (enteros)	! =	35 ! = 9	verdadero
5,5 (enteros)	<	5 < 5	falso
5,5 (enteros)	< =	5 < = 5	verdadero
5,5 (enteros)	! =	5 ! = 5	falso
“a”, “c” (caracteres)	= =	“a” = = “c”	falso
“a”, “c” (caracteres)	> =	“a” >= “c”	falso
“a”, “c” (caracteres)	< =	“a” < = “c”	verdadero

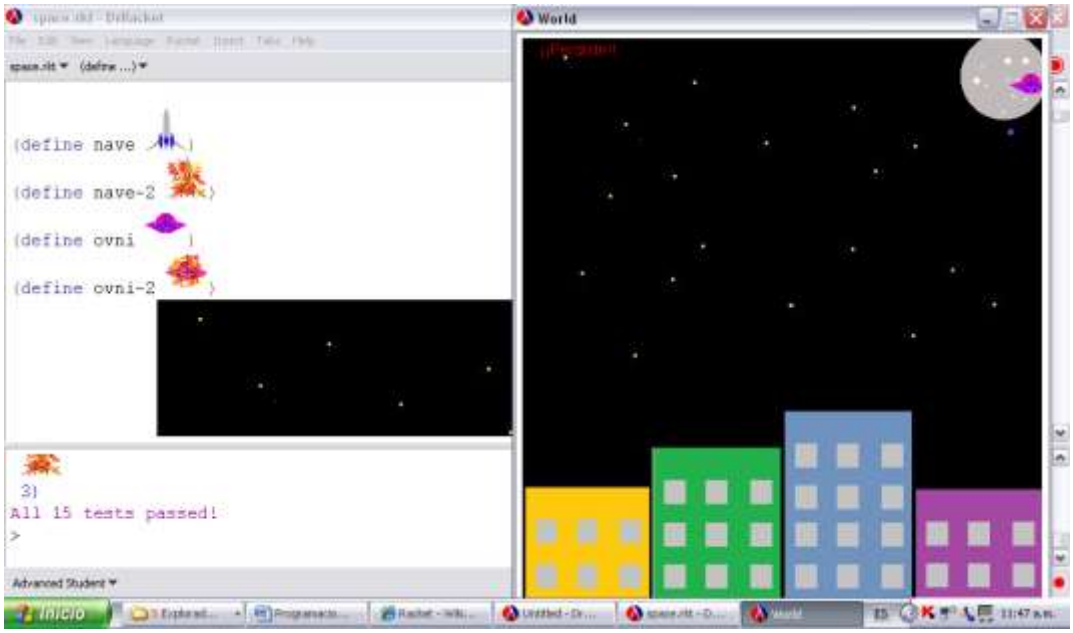
El tipo de dato *variant,* es exclusivo de Visual Basic; consiste en un tipo de dato especial que contiene datos numéricos, de cadena o de fecha, así como tipos definidos por el usuario, y los valores especiales *Empty y Null*. El tipo de datos *variant* tiene un tamaño de almacenamiento numérico de 16 bytes, puede contener datos hasta el intervalo de un tipo decimal o un tamaño de almacenamiento de caracteres de 22 bytes (más la longitud de cadena), y almacenar cualquier texto. La función *VarType* define el tratamiento que reciben los datos de un *variant*. Todas las variables son del tipo *variant,* a menos que se declaren explícitamente como de cualquier otro tipo. Los de tipo *variant* se encuentran en el Compilador de Visual Basic y ocupan una gran

cantidad de memoria interna, por eso se deben ocupar lo menos posible en un programa, con la salvedad de que no se sepa *a priori* el formato idóneo de un campo en una aplicación en particular.

En aplicaciones específicas, como Access, 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.

Actualmente las computadoras procesan, en mayor grado, información simbólica como nombres, palabras, direcciones o imágenes. Los lenguajes de programación modernos cuentan cuando menos con una forma para representar la información simbólica, como es el caso del lenguaje de programación Racket (formalmente llamado PLT Scheme o Dr. Racket). Este lenguaje funcional soporta diversas formas de expresión simbólica: símbolos, cadenas, caracteres e imágenes en movimiento.

El siguiente es un ejemplo de programación en Racket del juego de la guerra de los mundos.



1.3. Tipos de datos abstractos

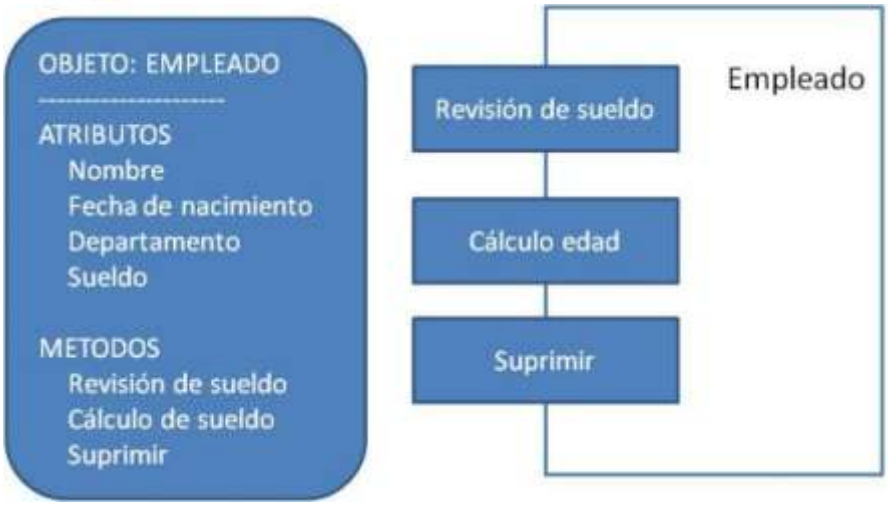
El concepto de Tipo de Dato Abstracto (TDA, Abstract Data Type), surge en 1974 por John Guttag, pero fue hasta 1975 que por primera vez, Bárbara Liskov, lo propuso para el lenguaje de programación llamado CLU[1]. Posteriormente fue implementado por lenguajes modulares como Turbo Pascal y Ada, y en la 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 que involucra objetos, los cuales tienen atributos, y todos ellos servirán para desarrollar una aplicación en particular.

Los TDA no hacen referencia a un tipo de dato específico, sino que su finalidad es la generalización de la definición del objeto dentro del TDA con sus propiedades establecidas. Los datos abstractos son el resultado de empacar un tipo de datos junto con sus operaciones, de modo que pueda considerarse en términos de sus generalidades, sin que el programador tenga que preocuparse por una representación en memoria o la instrucción de sus operaciones.

Por ejemplo:

[1] Para mayor información, ver: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-561.pdf (consultado el 28 de enero de 2013)



Al comenzar el diseño de un TDA (tipo abstracto), es necesario tener una representación genérica del objeto sobre el cual se quiere trabajar, sin establecer un compromiso con ninguna estructura de datos concreta y el tipo de dato del lenguaje de programación seleccionado. 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 abstracto, utilizando algún formalismo matemático o gráfico.

Siempre hay que tomar en cuenta que un TDA es un modelo abstracto para resolver un problema en particular y generar el programa correspondiente. La finalidad es aplicarse de forma general a los problemas semejantes, al originalmente planteado, por lo que resulta independiente del compilador al cual se le encargará de interpretar el programa fuente que creó el programador.

Un TDA se define con un nombre, un formalismo para expresar un objeto abstracto, un invariante[2] o un conjunto de operaciones sobre este objeto.

[2] Parte de un sistema que no admite las variaciones que afectan a otras partes del mismo.(http://www.wordreference.com/definicion/invariante, ref. 28/01/2013)

Veamos el siguiente esquema:

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

Object1		Precondición	B	……	J
Funcionalidad	A				
	Postcondición				
	I				X(I J)
Object2					
Propiedad	Dominio				X(A,J)

Estructura de un TDA. Elaboración propia.

En cuanto a la estructura de un TDA, podemos decir que la especificación de las operaciones consta de dos partes: primero se coloca la funcionalidad de cada una de ellas (dominio y codominio de la operación) y, luego su comportamiento, mediante dos aserciones (precondición y postcondición) que indican la manera como se va afectando el estado del objeto una vez ejecutada la operación.

A continuación se muestra un ejemplo de una estructura de un TDA y aplicación de la misma.

Ahora veamos de forma muy general un TDA para la Aplicación Auto.

Tamaño		Altura	Anchura	Largo	
Centímetros		A	B	C	
		:	:	:	
		I	I	I	
Motor					
Centímetros Cúbicos		CAPACIDAD			

TDA de Auto

El TDA Auto de la tabla anterior, se escribe en pseudocódigo de la siguiente forma:

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

Si se definen más objetos para este 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 de datos que se procesan.

Hemos visto que las Estructuras de Datos son modelos teóricos que muestran la forma en que la computadora maneja la información en la memoria interna. Son organizaciones de datos que conforman estructuras con propiedades y formatos englobados bajo identificadores asignados por el usuario para emplearse en estructuras de programación, simples o complejas 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 aún 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 matemático compuesto por una colección de operaciones definidas sobre un conjunto de datos para su aplicación en un modelo.

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*. (2ª ed.) México: McGraw-Hill [Vista previa de la 3ª ed.]

Unidad 2.

Estructuras de datos fundamentales

OBJETIVO PARTICULAR

Al terminar la unidad, 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. Arreglos
  - 2.1.1. Unidimensionales
  - 2.1.2. Multidimensionales
  - 2.1.3. Operaciones con arreglos
- 2.2. Pilas
  - 2.2.1. Operaciones con pilas
- 2.3. Colas
  - 2.3.1. Operaciones con colas
  - 2.3.2. Bicolas
- 2.4. Listas
  - 2.4.1. Listas simplemente enlazadas
  - 2.4.2. Listas doblemente enlazadas
  - 2.4.3. Listas circulares
  - 2.4.4. Operaciones con listas

INTRODUCCIÓN

El manejo de los datos complejos se integran de varios tipos de datos simples, puede ser procesando datos simples ya sea del mismo tipo o de varios tipos, según sus necesidades, para lo cual se dividen en Estáticos y Dinámicos. 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. Arreglos

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 (Cf. Joyanes, 1996).

El arreglo es una estructura de datos que hace referencia a un grupo de casillas de memoria que se puede ver como una colección finita[3], homogénea[4] y ordenada[5] de elementos.

Un arreglo tiene dos tipos de datos asociados, los numéricos y los caracteres. Las dos operaciones básicas a realizar en un arreglo son la alimentación y la extracción. La primera operación, acepta un acceso a una posición del arreglo con la ayuda de un dato de tipo índice, ya sea ordinario (entero), inicializándolo desde el 0. La segunda operación hace uso del índice para llegar al elemento deseado para después eliminarlo.

El elemento más pequeño de un arreglo del tipo índice es su límite inferior, y el más alto, su límite superior.

[3] Finita: quiere decir que indica el número máximo de elementos.
[4] Homogénea: Quiere decir que son 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.
[5] Ordenada: quiere decir que llevan un orden consecutivo a través de un índice.

Ejemplo.

A=	34	45	12	05	93	Datos
(0)	(1)	(2)	(3)	(4)	índices	

2.1.1. Arreglos Unidimensionales

Es una estructura que utiliza el mismo tipo de dato en forma secuencial comúnmente denominada vector, por estar definida por una sola dimensión y sus nodos leídos en una sola dirección. Una matriz de una dimensión se le llama Vector y está definida por la notación V = [0, 1, 2, 3, 4, 5].

Un vector es un arreglo unidimensional que sólo utiliza un índice para referenciar a cada uno de los elementos. Su declaración es la siguiente:

```
tipo nombre [tamaño];
```

Veamos un ejemplo de programación de un arreglo, usando lenguaje C:

```
#include <stdio.h>
main() /* Rellenamos el arreglo del 0 al 9 */
{
int vector[10],i;
        for (i=0;i<10;i++) vector[i]=i;
        for (i=0;i<10;i++) printf(" %d",vector[i]);
}
```

En este ejemplo definimos el arreglo, llamado vector, de tamaño 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 otro para mostrar los valores utilizando la instrucción *printf* del lenguaje C.

2.1.2. Arreglos Multidimensionales

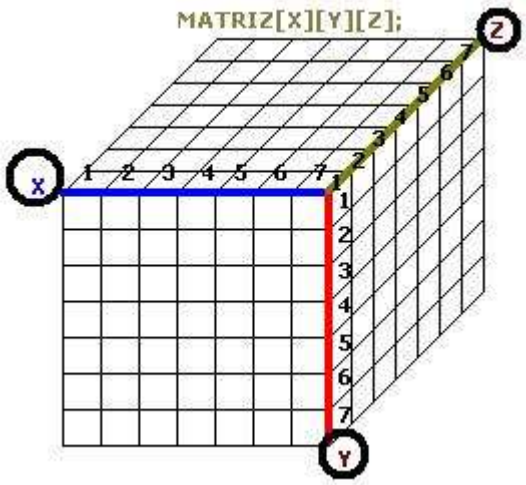
Un arreglo también puede ser de más de dos dimensiones: una matriz es un arreglo multidimensional.

Por ejemplo, en lenguaje C se definen igual que los vectores, excepto que se requiere un índice por cada dimensión. Su sintaxis es la siguiente:

```
tipo nombre [tamaño 1][tamaño 2]...;
```

La representación de arreglos en la forma de fila mayor puede extenderse a arreglos de más de dos dimensiones.

Para el caso de un arreglo con *estructura tridimensional*, estará especificado por medio de tres subíndices: El primer índice precisa el número del **plano**; el segundo el de la **fila**; y el tercero, el de la **columna**. Este tipo de arreglo es útil cuando se determina un valor mediante tres entradas.



Descripción Gráfica de un Arreglo de tres Dimensiones

Véase en: http://www.wikilearning.com/curso_gratis/aprende_c_con_paranoix-matriz_tridimensional/22916-17

El último subíndice varía rápidamente y no aumenta, sino hasta que todas las combinaciones posibles de los subíndices a su derecha hayan sido completadas.

2.1.3. 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** - **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[6].
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[7].

[6] En la unidad 4 abordaremos a detalle estos métodos.
[7] Posteriormente, en la unidad 5, abordaremos estos métodos a detalle.

Arreglos bidimensionales

En el caso de la matriz bidimensional, 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++:

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

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
```

Otra manera de llenar el arreglo, es mediante una instrucción *FOR* anidada, como se muestra en el siguiente código:

```
/* 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]);
}
```

En este ejemplo 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.

Arreglo de dos Dimensiones

Un arreglo de dos dimensiones ilustra claramente las diferencias lógica y física de un dato, 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 y dicha memoria es usualmente lineal; es decir, las computadoras tienen una arquitectura cuyo procesador ingresa la información secuencialmente hasta que una página de memoria se completa para continuar con la siguiente.

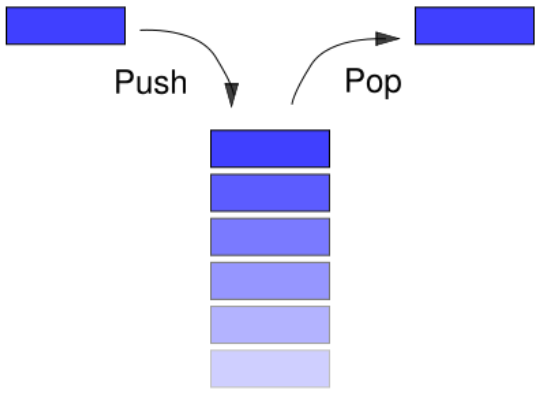
Un método para mostrar un arreglo de dos dimensiones en memoria es la representación fila mayor. Bajo esta representación, la primera fila del arreglo ocupa el primer conjunto de posiciones en memoria reservada para el arreglo; la segunda el segundo y así sucesivamente.

2.2 Pilas

Una pila (*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 (SUA Informática II, 1998, pp. 63 y ss.).

Definición del tipo de dato abstracto pila

Una pila es una colección ordenada de elementos en la cual, en un extremo, pueden insertarse o retirarse otros elementos, ubicados 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.



Véase en: http://commons.wikimedia.org/wiki/File:Pila.svg

Un ejemplo de pila o *stack* se puede observar en el mismo procesador; 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.

2.2.1. Operaciones con pilas.

Los dos cambios que pueden hacerse en una pila, tienen nombres especiales; cuando se agrega un elemento a la pila, éste es “empujado” (*pushed)* dentro de la pila; la operación *pop,* retira el elemento superior y lo regresa como el valor de una función; la *empty* determina si la pila está o no vacía; y la *stacktop* determina el elemento superior de la pila sin retirarlo (debe retomarse el valor del elemento de la parte superior de la pila).

Asimismo, una pila cuenta con 2 operaciones imprescindibles: apilar y desapilar. Sin embargo las implementaciones modernas consideran otras operaciones adicionales como son creación de la pila, obtener el número de elementos de la pila, verificar el elemento que está en el tope de la pila y saber si la pila se encuentra vacía.

Implantación de una pila basada en un arreglo estático.

La implantación de una pila, al igual que otras estructuras de datos, puede estar basada en estructuras estáticas o dinámicas y ser desarrollada en cualquier lenguaje de programación que soporte dichas estructuras.

Utilizando la estructura estática de un arreglo podemos implementar una pila, por ejemplo en el lenguaje C++, considerando las siguientes operaciones y definición del arreglo:

put(), poner un elemento en la pila
get(), retirar un elemento de la pila
empty(), regresa 1 (TRUE) si la pila está vacía
size(), número de elementos en la pila

El atributo SP de la clase Stack es el apuntador de lectura/escritura; es decir, el SP indica la posición dentro de la pila en donde la función put() insertará el siguiente dato, y la posición dentro de la pila de donde la función get() leerá el siguiente dato. Cada vez que put() inserta un elemento, el SP se decrementa. Cada vez que get() retira un elemento, el SP se incrementa.

La implementación del arreglo para el lenguaje C++ quedaría de la siguiente manera:

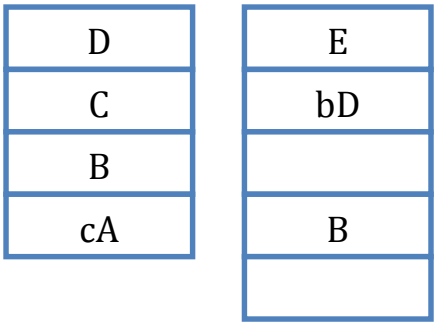
```
#define STACK_SIZE 256  /* capacidad máxima de la Pila*/
typedef char almacen[STACK_SIZE];
```

Operación para insertar un elemento a una pila

Los nuevos elementos de la pila deben colocarse en su parte superior que se mueve hacia arriba para dar lugar a un nuevo elemento más alto; además, los que están en este lugar pueden ser removidos (en este caso, la parte superior se desliza hacia abajo para corresponder al nuevo elemento más alto).

Ejemplo:

En una película en movimiento de una pila se agrega el elemento G a la pila. A medida que nuestra película avanza, puede verse que los elementos F, G y H han sido agregados sucesivamente a la pila. A esta operación se le llama lista empujada hacia abajo.



Movimiento en la Pila

Operación para revisar si una pila es vacía o no

Una pila puede utilizarse para registrar los diferentes tipos de signos de agrupación. En cualquier momento que se encuentre un signo de estos abriendo la expresión, es empujado hacia la pila, y cada vez que se detecte el correspondiente signo terminal, ésta se examina. Si la pila está vacía, quiere decir que el signo de la agrupación terminal no tiene su correspondiente apertura, por lo tanto, la hilera es inválida. En C++ podemos definir una función vacía (empty) para que regrese 1 si no hay elementos en la lista; es decir, si la lista está vacía.

```
int empty() { return ITEMS == 0; }
```

Operación para obtener el último elemento insertado en la pila

La operación *pop* retira el último elemento superior y lo regresa como un valor de la función (en cada punto se aleja el elemento superior, porque la operación sólo puede hacerse desde este lugar). El atributo más importante consiste en que el último elemento insertado en una pila es el primero en ser retirado. En la siguiente función de C++, cada vez que put() inserta un elemento el SP (indicador de posición) se decrementa.

```
/* insertar elemento a la pila */
int put(char d)
{
   if ( SP >= 0) {
      PILA[SP] = d;
      SP --;
      ITEMS ++;
   }
   return d;
}
```

Operación para remover el último elemento insertado en la pila

La operación *stackpop* (avance de elementos) determina el elemento superior de la pila, basta con retirarlo y reasignar el valor del elemento de la parte superior de la pila.

En la siguiente función, usando C++, cada vez que get() retira un elemento, el SP se incrementa.

```
/* retirar elemento de la Pila */
int get()
{
   if ( ! empty() ) {
      SP ++;
      ITEMS --;
   }
   return PILA[SP];
}
```

Implantación de una Pila Dinámica

Esta implantación es igual a la anteriormente mencionada, con la diferencia de que una pila enlazada dinámicamente no tiene, de forma natural, el mecanismo de acceso por índices, por lo tanto el programador puede crear los algoritmos necesarios para permitir tal comportamiento utilizando apuntadores o ligas a los nodos. Para tal efecto, se crean estructuras conocidas como nodos o registros. Un registro en Lenguaje C o C++, se define de la siguiente forma.

```
/* tipo de dato que contendrá la Pila */
typedef char DATA_TYPE;

// declaración de estructura nodo o registro
struct nodo {
        DATA_TYPE data;
        nodo *next;
};
```

En esta estructura nodo, de tipo carácter, se define un apuntador que liga a *next* (siguiente).

La pila incorpora la inserción y supresión de elementos, por lo que ésta es un objeto dinámico constantemente variable. La definición especifica que un extremo de la pila se designa como tope de la misma. Pueden agregarse nuevos elementos en el tope de la pila, o quitarse.

PILAS DINÁMICAS

1) Al principio la lista está vacía, en ese caso el SP es igual a NULL y, en consecuencia, el puntero next también es NULL.

```
SP = NULL
+------+------+
| ???? | next |--> NULL
+------+------+
```

2) Después de agregar el primer elemento la situación se vería así:

```
SP = asignado
       1
+------+------+
| data | next |--> NULL
+------+------+
```

3) Después de agregar otro elemento la situación se vería así:

```
SP = asignado
      2               1
+------+------+    +------+------+
| data | next |--> | data | next |--> NULL
+------+------+    +------+------+
```

Véase en http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C%2B%2B/Estructuras_II

La implantación de la clase Pila Dinámica (*StackDim*) quedará de la siguiente forma:

```
class StackDin {

  // atributos
  int ITEMS;    /* número de elementos en la lista */
  int ITEMSIZE; /* tamaño de cada elemento */
```

```
    nodo *SP;     /* puntero de lectura/escritura */

public:
    // constructor de la pila dinámica
    StackDin() : SP(NULL), ITEMS(0), ITEMSIZE(sizeof(DATA_TYPE)) {}

    // destructor
    ~StackDin() {}

    /* agregar componente a la lista */
    DATA_TYPE put(DATA_TYPE valor)
    {
            nodo *temp;

            temp = new nodo;
            if (temp == NULL) return -1;

            temp->data = valor;
            temp->next = SP;
            SP = temp;
            ITEMS ++;
            return valor;
    }

    int empty() { return ITEMS == 0; }

     /* retirar elemento de la lista */
    DATA_TYPE get()
    {
            nodo *temp;
            DATA_TYPE d;

            if ( empty() ) return -1;

            d = SP->data;
            temp = SP->next;
            if (SP) delete SP;
            SP = temp;
            ITEMS --;
            return d;
    }

}; // fin de la clase StackDin
```

La implementación de clases también se conoce como Programación Orientada a Objetos o POO. Es un paradigma de programación que usa los objetos en sus interacciones para diseñar aplicaciones y programas informáticos. Las clases son representaciones abstractas de los objetos que nos permiten definir las propiedades y comportamiento de los mismos, como en este caso la representación abstracta de una Pila Dinámica.

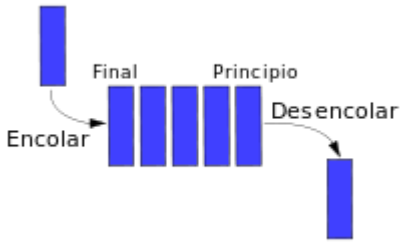
Con esta forma de implantación de Pilas concluimos este tema, para pasar al siguiente, de igual importancia en el ámbito de estructuras de datos, como son las Colas.

2.3. Colas

En la vida cotidiana es muy común ver las colas como formas organizativas para agilizar los servicios de una empresa u organización escolar. 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*), 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. Este tipo de estructura de datos abstracta también *se implementa en lenguajes orientados a objetos mediante clases, en forma de listas enlazadas.*

En este ejemplo vemos una representación gráfica de una cola.



Véase en: http://commons.wikimedia.org/wiki/File:Cola.svg

2.3.1 Operaciones con colas

Las operaciones que se pueden realizar con una cola son:

- **Crear**: se crea la cola vacía (constructor).
- **Encolar (añadir, entrar, insertar):** se añade un elemento a la cola. Se añade al final de ésta.
- **Desencolar (sacar, salir, eliminar):** se elimina el elemento frontal de la cola; es decir, el primer elemento que entró.
- **Frente** (consultar, *front*): se devuelve el elemento frontal de la cola; es decir, el primer elemento que entró.

Implantación de cola dinámica

Siguiendo con el lenguaje C++, en esta estructura nodo de tipo carácter se define un apuntador que liga a *next* (siguiente).

```
typedef char DATA_TYPE;
 struct nodo {
      DATA_TYPE data;
      nodo *next;
};
```

Se crea la cola vacía y se inicializan los atributos.

```
   // definición de atributos
   int ITEMS, ITEMSIZE;
   nodo *cola, *cabeza;



public:
   // constructor
   QueueDin() : cola(NULL), cabeza(NULL), ITEMS(0),
ITEMSIZE(sizeof(DATA_TYPE)) {}
```

En la siguiente función de C++, cada vez que put() inserta un elemento a la cola queda como cabeza y se incrementan los elementos (ITEMS) .

```
/* encolar un componente */
   DATA_TYPE put(DATA_TYPE valor)
   {
      nodo *temp;

      temp = new nodo;
      if (temp == NULL) return -1;

      ITEMS ++;
      temp->data = valor;
      temp->next = NULL;

      if (cabeza == NULL)
      {
         cabeza = temp;
         cola   = temp;
      }  else
      {
         cola->next = temp;
         cola = temp;
      }
      return valor;
   }
```

En la siguiente función de C++, cada vez que get() saca un elemento de la cola la cabeza se van recorriendo o disminuyendo los elementos (ITEMS)

```
/* desencolar un  elemento */
DATA_TYPE get()
{
   nodo *temp;
   DATA_TYPE d;

   if ( empty() ) return -1;

   d = cabeza->data;
   temp = cabeza->next;
   if (cabeza) delete cabeza;
   cabeza = temp;
   ITEMS --;
   return d;
}
```

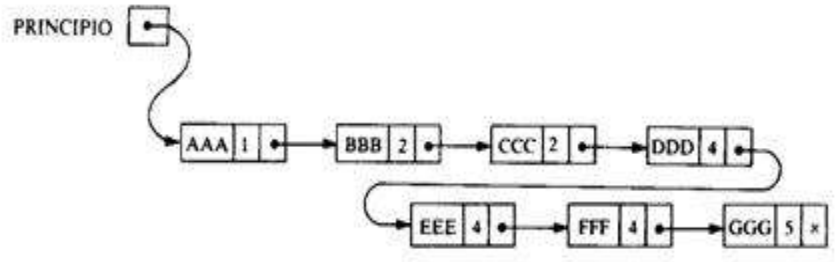
Como ejercicio personal, puedes integrar los códigos anteriores y hacer un programa en C++ llamado *ColaDinámica*, y probarlo en un compilador para que tengas mayor claridad de su funcionamiento.

Colas de prioridades

En ocasiones necesitaremos un algoritmo que nos ayude a seleccionar un elemento de un grupo. A uno de los valores de la información agrupada en la estructura se le llama prioridad, este es un valor entero, y el menor valor entero está asociado a la estructura que tiene mayor prioridad. Prioridad se entiende como sinónimo de lo más importante. Puede haber varias estructuras con igual prioridad y en este sentido no serán conjuntos.

Para efectos de estructuras de datos, una cola de prioridades, es una estructura en la que los elementos se procesan en el orden indicado por una prioridad asociada a cada uno de estos. En el caso de que varios elementos tengan la misma prioridad, el

procesamiento se atenderá de modo convencional según la posición en que tengan en la estructura.



Véase en: http://www.udg.co.cu/cmap/estrdatos/colas/ColasPrioRepListasUnicas.htm

Como un ejemplo de este tipo de estructura, es la que aplican los bancos al atender a clientes especiales. Otro ejemplo típico es la programación, formando colas de prioridades en el sistema de tiempo compartido necesario para mantener un conjunto de procesos que esperan servicio para trabajar. Los diseñadores y programadores de esta clase de sistemas asignan cierta prioridad a cada proceso.

La prioridad se define como un valor numérico asignando a altas prioridades valores pequeños, las colas de prioridad nos permiten añadir elementos en cualquier orden y recuperarlos de menor a mayor.

Las operaciones que se pueden realizar con colas de prioridades.

Las operaciones que se pueden realizar con una cola son:

Crear	Se crea la cola vacía.
Añadir	Se añade un elemento a la cola, con su correspondiente prioridad.
Eliminar	Se elimina el elemento frontal de la cola.
Frente	Se devuelve el elemento frontal de la cola.
Destruye	Elimina la cola de memoria.

Implantación dinámica de una cola de prioridades

En esta implementación, se trata de crear tantas colas como prioridades haya, y almacenar cada elemento en su cola utilizando clases en lenguaje de programación Java. Se crea clase principal **ColaPrioridad** que implementa *colaPrioridadInterface*. Posteriormente, el constructor crea el objeto cola de la clase Celda y cola.sig inicializada en *null*. Se define el método público booleano **vacía** (*empty*) que regrese 1 si no hay elementos en la cola; es decir, si la cola está vacía. También se define el método **primero** y **primero_prioridad** que devuelven el elemento frontal de la cola y su prioridad respectiva. Con el método *public* **inserta** se añade un elemento a la cola, con su correspondiente prioridad. El método **suprime** eliminará el elemento de la cola en la memoria.

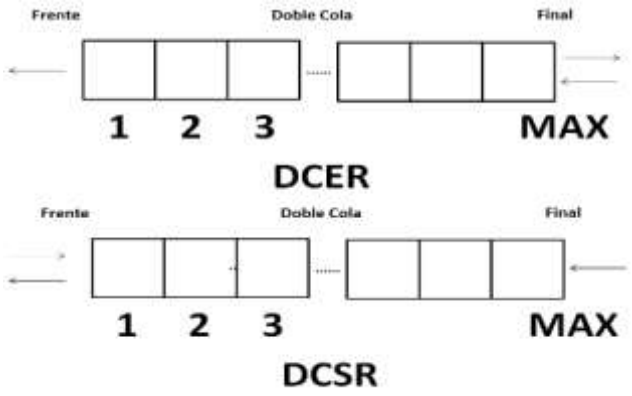
Como ejercicio personal, implementa éste programa en lenguaje de programación C++ que se encuentra en la siguiente ruta:
http://casicodigo.blogspot.mx/2012/11/cola-con-prrioridad-en-c.html

2.3.2. Bicolas (SUA Informática II, 1998, pp.59 y ss.)

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. 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 dos métodos para leer (uno para leer por el frente y uno para leer por atrás) y dos métodos para escribir (uno para escribir por el frente y uno para escribir por atrás). Se conocen dos variantes de las dobles colas:

La doble cola con entrada restringida (DCER) donde 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) donde las inserciones se realizan por cualquiera de los dos extremos, mientras que las eliminaciones solo por el frente de la cola. Si bien estas estructuras están ligadas a la computación, impresión y los sistemas de tiempo compartido, también las podemos observar en las vías de los trenes.



Las operaciones básicas que definen una bicola son:

Crear	Inicializa una bicola sin elementos.
Esvacia	Devuelve verdadero si la bicola no tiene elementos
InsertIzq	Añade un elemento por el extremo izquierdo (EncolarIzquierda).
InserDer	Añade un elemento por el extremo derecho (EncolarDerecha)
ElimnIzq	Devuelve el elemento izquierdo y lo retira de la bicola (DesencolarIzquierda)
EliminDer	Devuelve el elemento derecho y lo retira de la bicola (DesencolarDerecha)

Véase en : http://www.geocities.ws/profeprog/P2TP05.PDF

Implantación de una Cola Dinámica doblemente enlazada

En el ámbito de la Informática, el término *Implantar* se le da la connotación de adecuar las características de un compilador, el cual no tiene definida o precargada una estructura determinada para que, por medio de algoritmos y estructuras ya definidas por el compilador, se pueda realizar alguna aplicación requerida. En este caso, tomando como base el Lenguaje C++, se implantará una cola doblemente encadenada como una estructura en donde cada elemento puede ser insertado y recuperado por la parte del frente (cabeza) o por la parte de atrás (cola) de la lista. A diferencia de una cola simple, en donde sólo se necesita un apuntador a un siguiente elemento, la estructura del nodo para una doble cola debe poseer un apuntador a un posible siguiente elemento y un apuntador a otro posible anterior elemento como se muestra en la siguiente estructura.

```
typedef char DATA_TYPE;
 struct nodo {
   DATA_TYPE data;
   nodo *next, *prev;
};
```

Para la clase DobleCola podemos definir los siguientes métodos:

put_front(),	poner un elemento en el frente de la cola
put_back(),	poner un elemento en la parte trasera de la cola
get_front(),	retirar un elemento de la parte frontal de la cola
get_back(),	retirar un elemento de la parte trasera de la cola
empty(),	regresa 1 (TRUE) si la cola está vacía
size(),	número de elementos en la cola

Véase en. http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C%2B%2B/Estructuras_II

El método *put_back()*, pone un elemento en la parte trasera de la cola:

```
DATA_TYPE put_back(DATA_TYPE valor)
{
   nodo *temp;

   temp = new nodo;
   if (temp == NULL) return -1;

   temp->data = valor;

   items ++;
   if (cabeza == NULL )
   {
      temp->next = NULL;
      temp->prev = NULL;
      cabeza = temp;
      cola = temp;
   } else
      {
       cola->next = temp;
       temp->prev = cola;
       cola = temp;
       cola->next = NULL;
      }
   return valor;
}
```

El método *put_front()*, coloca un elemento en la parte frontal de la cola:

```
DATA_TYPE put_front(DATA_TYPE valor)
{
   nodo *temp;

   temp = new nodo;
   if (temp == NULL) return -1;

   temp->data = valor;
```

```
   items ++;
   if (cabeza == NULL )
   {
      temp->next = NULL;
      temp->prev = NULL;
      cabeza = temp;
      cola = temp;
   } else
      {
       cabeza->prev = temp;
       temp->next = cabeza;
       cabeza = temp;
       cabeza->prev = NULL;
      }
   return valor;
}
```

El método *empty(),* regresa *true* si la cola está vacía:

```
int empty() { return items == 0; }
```

El método *get_front*(), retira el elemento en la parte frontal de la cola:

```
DATA_TYPE get_front()
{
   nodo *temp;
   DATA_TYPE d;

   if ( empty() ) return -1;

   items --;
   d = cabeza->data;
   temp = cabeza->next;
   if (cabeza) delete cabeza;
   cabeza = temp;
   return d;
```

El método *get_back(),* retira un elemento de la parte trasera de la cola:

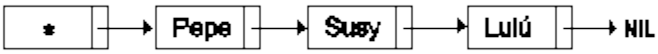
```
DATA_TYPE get_back()
{
   nodo *temp;
   DATA_TYPE d;

   if ( empty() ) return -1;

   items--;
   d = cola->data;
   temp = cola->prev;
   if (cola) delete cola;
   cola = temp;
   return d;
}
```

Como hemos señalado anteriormente, se consideran listas lineales a las listas enlazadas, pilas y colas, con todas sus variantes. En el siguiente tema detallaremos el concepto de **Listas.**

2.4. Listas

Una lista lineal es un conjunto de elementos de un tipo dado que se encuentren ordenados (pueden variar en número). Los elementos de una lista se almacenan normalmente de manera contigua (un elemento detrás de otro) en posiciones de la memoria (véase, SUA, 1998, pp. 37 y ss.).

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 y las colas simples y doble cola. Tiene 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.



2.4.1. Listas simplemente enlazadas

Una lista enlazada es un conjunto de elementos que contienen la posición -o dirección- del siguiente. Cada elemento de una lista enlazada debe tener al menos dos campos: uno con el valor del elemento y otro (link) que contiene la posición del siguiente elemento o encadenamiento.
La lista enlazada básica es la lista enlazada simple la cual tiene un enlace por nodo. Este enlace apunta al siguiente nodo en la lista, o al valor *NULL* o a la lista vacía, si es el último nodo.

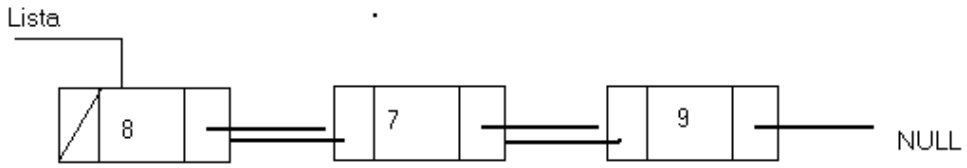
Se define una lista como una secuencia de cero o más elementos de un mismo tipo. El formalismo encogido para representar este tipo de objeto abstracto es:

<e1, e2 ....., en>

Cada ejemplo modela un elemento del agrupamiento. Así, *e1* es el primero de la lista; *en,* el último; y la lista formada por los elementos <e2, e3,.....,*en* corresponde al resto de la lista inicial.

2.4.2. Listas doblemente enlazadas

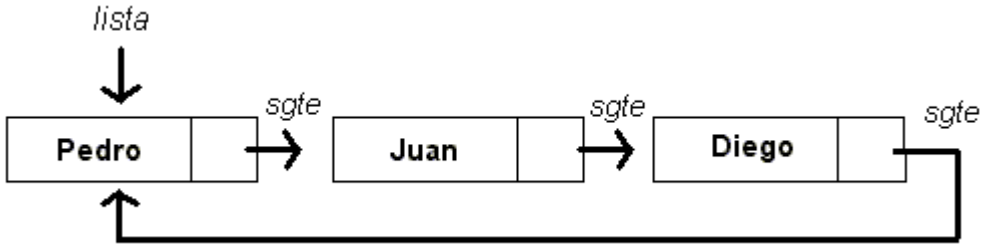
Un tipo de lista enlazada más sofisticado es la lista doblemente enlazada o lista enlazadas de dos vías. Cada nodo tiene dos enlaces: uno apunta al nodo anterior (o al valor NULL si es el primer nodo); y otro apunta al nodo siguiente (o al valor NULL si es el último nodo).



En algún lenguaje de muy bajo nivel, como XOR-Linking, ofrece una vía para implementar listas doblemente enlazadas, usando una sola palabra para ambos enlaces, aunque el uso de esta técnica no suele utilizarse.

2.4.3. Listas circulares

Esta estructura se aplica tanto a listas simples como en listas doblemente enlazadas. La lista circular consiste en que la referencia siguiente del último nodo apunta al primero en vez de ser *null*, de ahí que sea circular.



Véase en
http://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/Estructuras/http://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/Estructuras/

En algunas listas circulares se añade un nodo especial de cabecera, de ese modo, se evita la única excepción posible, la de que la lista esté vacía.

2.4.4. Operaciones con listas

Las operaciones que pueden realizarse con listas lineales contiguas son:

1. Insertar, eliminar o localizar un elemento.
2. Determinar el tamaño de la lista.
3. Recorrer la lista para localizar un determinado elemento.
4. Clasificar los elementos de la lista en orden ascendente o descendente.
5. Unir dos o más listas en una sola.
6. Dividir una lista en varias sublistas.
7. Copiar la lista.
8. Borrar la lista.

Operaciones que normalmente se ejecutan con las listas enlazadas:

1. Recuperar información de un nodo especificado.
2. Encontrar un nodo nuevo que contenga información específica.
3. Insertar un nodo nuevo en un lugar específico de la lista.
4. Insertar un nuevo nodo en relación con una información particular.
5. Borrar un nodo existente que contenga información específica.

Implantación de listas enlazadas

Las listas enlazadas pueden ser implementadas en muchos lenguajes. Como mencionamos anteriormente, Lisp y Scheme (Dr. Racket) tiene estructuras de datos ya construidas, junto con operaciones para acceder a las listas enlazadas. Lenguajes imperativos u orientados a objetos tales como C o C++ y Java, respectivamente, disponen de referencias para crear listas enlazadas como se muestra en el siguiente ejemplo en C.

Implantación de una lista enlazada en C.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct ns {
      int data;
      struct ns *next;
} node;

node *list_add(node **p, int i) {
   node *n = (node *)malloc(sizeof(node));
   if (n == NULL)
      return NULL;
   n->next = *p;
   *p = n;
   n->data = i;
   return n;
}

void list_remove(node **p) { /* borrar cabeza*/
   if (*p != NULL) {
      node *n = *p;
      *p = (*p)->next;
      free(n);
   }
}

node **list_search(node **n, int i) {
   while (*n != NULL) {
      if ((*n)->data == i) {
         return n;
      }
      n = &(*n)->next;
   }
   return NULL;
}

void list_print(node *n) {
   if (n == NULL) {
      printf("lista esta vacía\n");
   }
   while (n != NULL) {
```

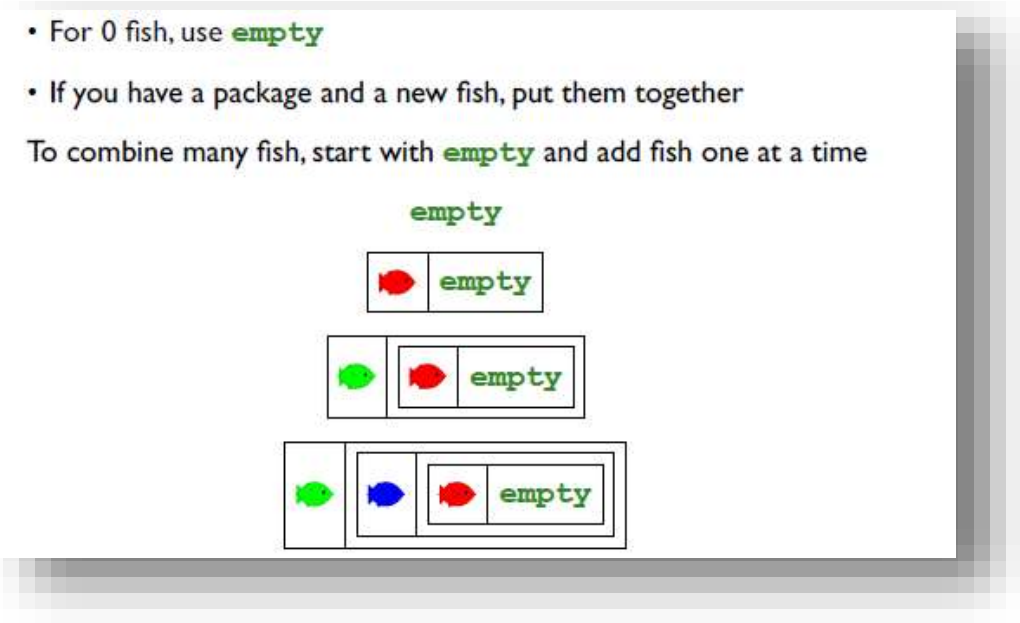
```
        printf("print %p %p %d\n", n, n->next, n->data);
        n = n->next;
    }
}

int main(void) {
    node *n = NULL;

    list_add(&n, 0); /* lista: 0 */
    list_add(&n, 1); /* lista: 1 0 */
    list_add(&n, 2); /* lista: 2 1 0 */
    list_add(&n, 3); /* lista: 3 2 1 0 */
    list_add(&n, 4); /* lista: 4 3 2 1 0 */
    list_print(n);
    list_remove(&n);          /* borrar primero(4) */
    list_remove(&n->next);    /* borrar nuevo segundo (2) */
    list_remove(list_search(&n, 1)); /* eliminar la celda que
contiene el 1 (primera) */
    list_remove(&n->next);          /* eliminar segundo nodo del
final(0)*/
    list_remove(&n);          /* eliminar ultimo nodo (3) */
    list_print(n);

    return 0;
}
```

En esta implantación de lista enlazada en C se definen las librerías *stdio.h* para utilizar instrucciones de salida como *printf* y *stdlib.h,* para facilitar el manejo dinámico de la memoria con la función *malloc*. Se define la función ***list_add*** para agregar nuevos elementos a la lista. En algunos compiladores no requieren un casting del valor del retorno para malloc. La función ***list_remove*** eliminará los nodos de la lista. Se define la función ***list_search*** para buscar nodos en la lista. Se define la función ***list_print*** para mostrar los nodos de la lista. Al final del programa tenemos la función ***main()*** o principal que ejecuta diferentes operaciones realizadas en esta lista enlazada.

Implantación de listas en Dr. Racket.

Los lenguajes de programación tales como Lisp y Scheme (Dr. Racket), tienen listas enlazadas simples ya construidas y la programación de estas estructuras es mucho más sencilla.

Este lenguaje nos especifica que formar listas es algo que todos hacemos, ya que es una estructura de longitud arbitraria con un número indeterminado de datos, por ejemplo: cuando planeamos una fiesta se hace una lista de invitados, en un acuario se hace una lista de las especies disponibles de peces, etc.



Véase en http://www.eng.utah.edu/~cs1410-20/f10/lecture7.pdf

Este ejemplo del acuario nos muestra las diferentes combinaciones de peces, empezando con la lista vacía ***empty*.** Esta es la base por la cual se puede construir una lista mayor, mediante la operación **cons**. Por ejemplo:

```
(cons 'piraña empty)


; Se construye una lista con el elemento piraña
; en este lenguaje los comentarios inician con punto y coma
```

La caja *cons* contiene dos campos: *first* y *rest* (el primer elemento y el resto de la lista). En el ejemplo de los peces, el campo *first* contiene 'piraña y el campo *rest* contiene *empty*.

Una vez que se cuenta con una lista que contiene un elemento, se pueden **cons**truir con dos o más elementos, empleando *cons* nuevamente.

```
(cons 'piraña (cons 'angel (cons 'globo empty)))
; Se construye una lista con tres elementos
```

Su campo *rest* contiene una caja que contiene una caja a su vez (pueden incluir estructuras más complejas).

Para analizar mejor la relación que tiene *first*, *rest* y *cons*, se pueden emplear ecuaciones similares a las ecuaciones que gobiernan la adición, resta, creación de estructuras y extracción de campos.

Ejemplos:

```
> (first (cons 'piraña empty))
'piraña
> (rest (cons 'globo empty))
empty
```

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. Éstos se dividen en estáticos y dinámicos, a fin de optimizar el uso de memoria a través del procesamiento de estructuras de datos adecuadas. 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: son estructuras de datos básicas porque se caracterizan en que todos sus elementos están en secuencia, en forma relacionada y lineal, uno luego del otro. Donde cada nodo puede estar formado por uno o varios elementos que pueden pertenecer a cualquier tipo de dato de los cuales normalmente son tipos básicos; las listas, que pueden ser listas simples enlazadas, doblemente enlazadas o enlazadas circulares.

En las 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 esta vacía, si es el último nodo). En la lista doblemente enlazada tenemos una estructura de datos en forma de lista enlazada de dos vías donde cada nodo tiene dos enlaces uno apunta al nodo anterior, o apunta al valor *NULL* si es el primer nodo; y otro que apunta al nodo siguiente, o apunta al valor *NULL* si es el último nodo. En una lista enlazada circular, el primer y el último nodo están enlazados en forma de anillo o círculo. Esto se puede hacer tanto para listas enlazadas simples como para las doblemente enlazadas.

Este conjunto 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 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
Olimpiadas de Informática	Estructura de datos	http://www.olimpiadadeinformatica.org.mx/archivos/apuntes/EstructurasDeDatos.htm

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

Toledo Salinas, Ana María. (2005). "¿Qué es pilas?", (22/03/05), disponible en línea.

Unidad 3.

Estructuras de datos avanzadas

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. Estructura de datos avanzadas

- 3.1. Árboles
  - 3.1.1. Árboles binarios de búsqueda
  - 3.1.2. Recorridos
  - 3.1.3. Operaciones con árboles binarios de búsqueda
- 3.2. Grafos
  - 3.2.1. Grafos dirigidos
  - 3.2.2. Grafos ponderados
  - 3.2.3. Operaciones con grafos

INTRODUCCIÓN

En esta unidad veremos dos de las estructuras más importantes por el número de aplicaciones que tienen en la vida real: los árboles y los grafos.

Los árboles y grafos son estructuras de datos que de alguna manera les llamamos complejos porque permiten organizar y mantener información flexible y dinámica en una computadora. Esta forma sale de la idea de organizar información con lápiz y papel usando nodos y flechas entre los nodos (a esas flechas también se les llama arcos, a los nodos también se les llama vértices). Los grafos y árboles en papel son muy utilizados en informática y teoría de redes, son muy apropiados para capturar sólo una parte de la información de objetos, situaciones y otros tipos de información relacional.

En la computadora, además de permitir organizar información, resultan ser estructuras útiles para resolver ciertos tipos de problemas de abstracción (por ejemplo, pueden emplearse árboles AVL para mantener información ordenada de forma eficiente).

3.1. Árboles

La organización de la información en la memoria interna de la computadora se representa con estructuras de datos y el contenido de las páginas se representa con tipos de datos simples con los cuales se pueden acceder, manejar y almacenar la información.

Definiciones

a) “Son un conjunto no vacío de vértices (nodos) y aristas (enlaces) que cumple una serie de requisitos” (Sedgewick, 2000, p. 40).

b) “Son Estructuras de Datos no lineales. Es una colección de nodos donde cada uno, además de almacenar información, guarda la dirección de sus sucesores” (Guardati, 2007, p. 313).

c) Los Árboles, a diferencia de las Listas, sirven para representar Estructuras Jerárquicas, hecho que no se puede realizar con las Listas Lineales y mucho menos con los Arreglos. Las Pilas y Colas, aunque representan cierta jerarquía, están limitadas a emplear una sola dimensión” (Drozdek, 2007, p. 214).

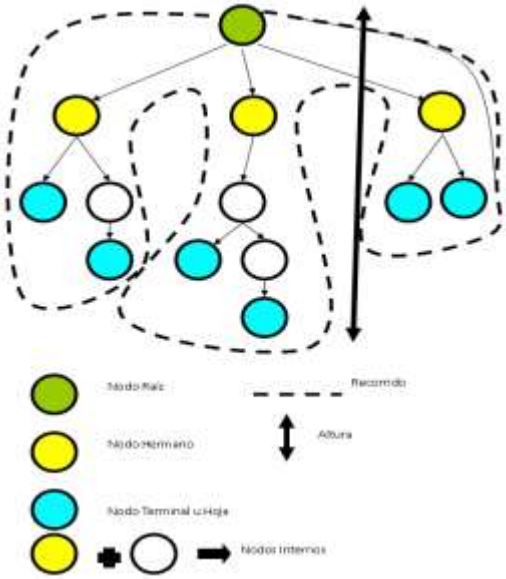
d) Un Árbol se puede representar como Conjuntos Venn, Anidación de Paréntesis, Grafos y como una Estructura Jerárquica.

Elementos del árbol

- Nodo Raíz, Nodos Hijos, Nodos Hermanos, Altura, Recorridos, Dirección.
- Todo Árbol tiene un solo Nodo Raíz.
- Los Árboles pueden tener o no Nodos Hijos. En caso de tenerlos, pueden existir Nodos Hermanos.

- Si únicamente tiene un Nodo Raíz, su Altura = 0 y su Nivel = 1.
- Los Recorridos pueden ser en preorden, postorden y en orden.
- Un Árbol puede recorrerse en dirección Top-Down de arriba abajo, de abajo arriba, (Down-Top). A partir de su rama Izquierda o a partir de su rama Derecha.

Para mayor claridad se representa la siguiente figura.



Elementos de un Árbol

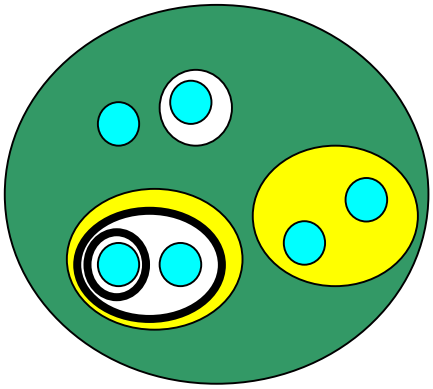
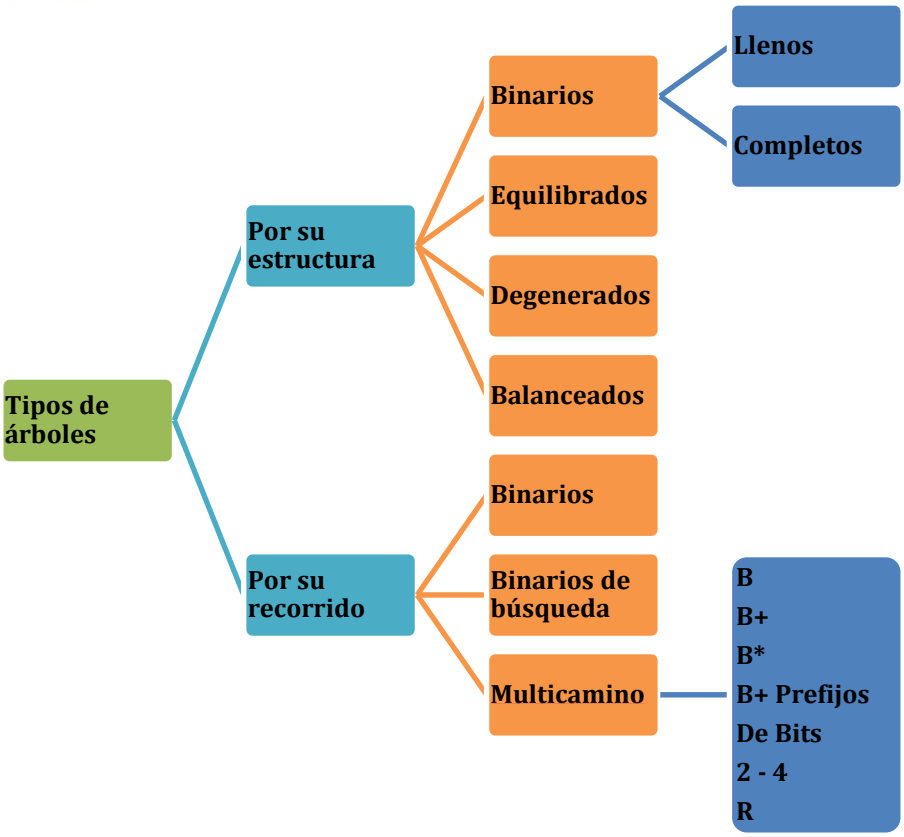


Diagrama de Venn del Árbol



Clasificación de los árboles. Elaboración propia.

3.1.1. Árboles binarios de búsqueda

Un árbol binario T se define como un conjunto finito de elementos, llamados nodos, de forma que:

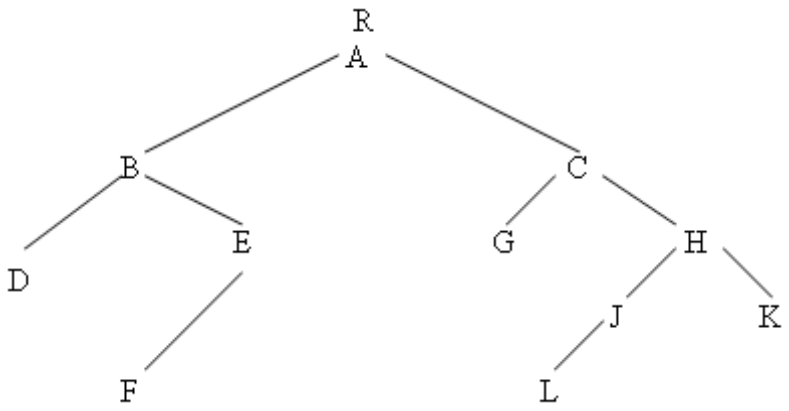
a) T es vacío (en cuyo caso se llama árbol nulo o árbol vacío) o
b) T contiene un nodo distinguido R, llamado raíz de T, y los restantes nodos de T forman un par ordenado de árboles binarios disjuntos T1 y T2. (Cairó y Guardati, 2006, p. 184)

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 es vacío, entonces su raíz se llama sucesor izquierdo de R; y análogamente, si T2 no es vacío, su raíz se llama sucesor derecho de R.

Obsérvese que:

a) B es un sucesor izquierdo y C un sucesor derecho del nodo A.

b) El subárbol izquierdo de la raíz A consiste en los nodos B, D, E y F, y el subárbol derecho de A consiste en los nodos C, G, H, J, K y L.



Representación Gráfica de Árbol Binario
Fuente: http://google.com/imagenes

* Cualquier nodo N de un árbol binario T tiene 0, 1 ó 2 sucesores. Los nodos A, B, C y H tienen dos sucesores, los nodos R y J sólo tienen un sucesor, y los nodos D, F, G, L y K no tienen sucesores. Los nodos sin sucesores se llaman nodos terminales.

* La definición anterior del árbol binario T es recursiva, ya que T se define en términos de los sub-árboles binarios T1 y T2. Esto significa, en particular, que cada nodo 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.

3.1.2. Recorridos.

Las dos maneras más usuales de representar un árbol binario en memoria son:

- Por medio de datos tipo puntero, también conocidos como variables dinámicas
- Por medio de arreglos

Los nodos del árbol binario se representan como registros. Cada uno de ellos contiene como mínimo tres campos. En un campo se almacenará la información del nodo. Los dos restantes se utilizarán para apuntar los subárboles izquierdo y derecho, respectivamente, del nodo en cuestión.

Dado el nodo T

IZQ	INFO	DER

En él se distinguen tres campos:

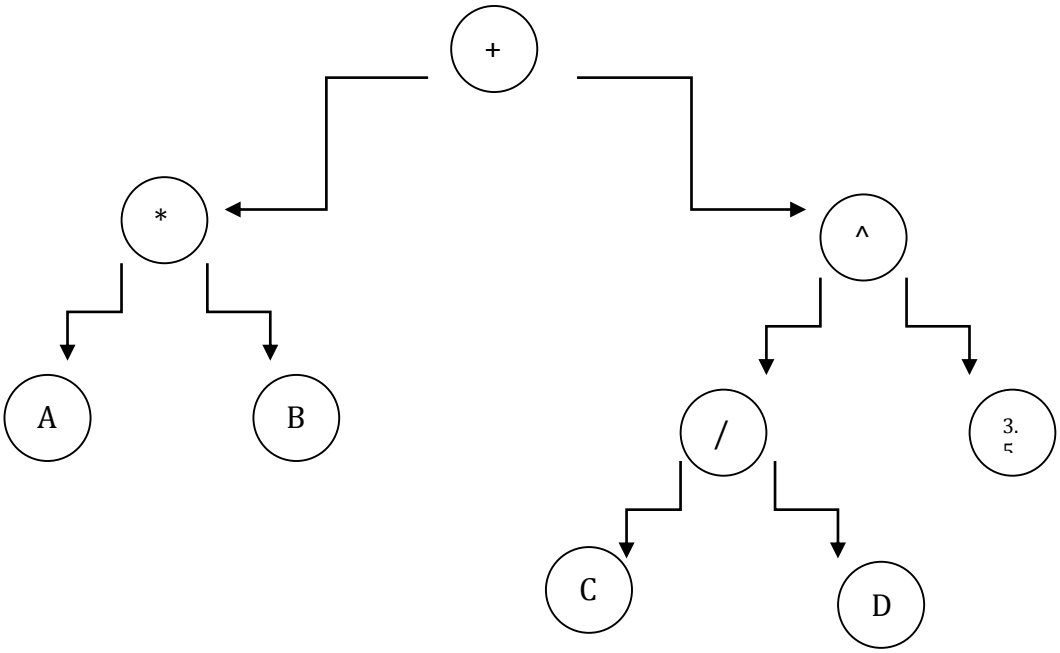
IZQ: es el campo donde se almacenará la dirección del subárbol izquierdo del nodo T.

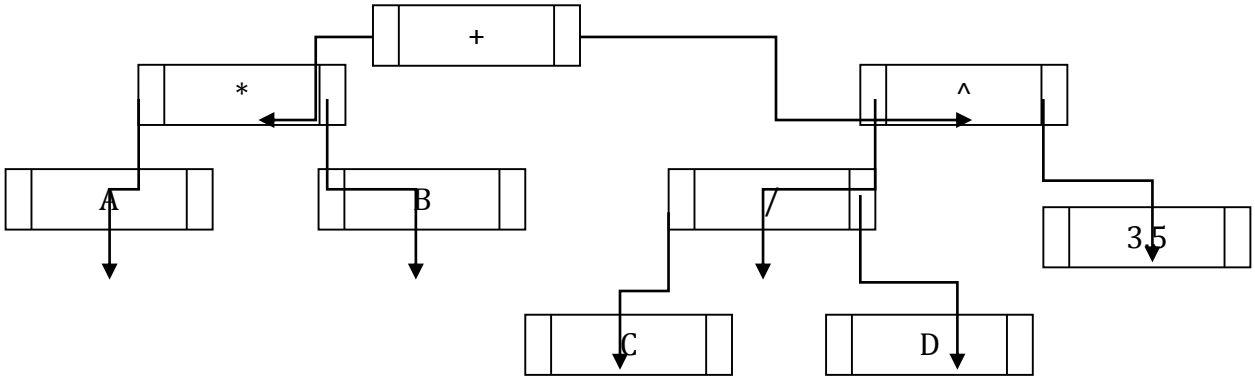
INFO: representa el campo donde se almacena la información del nodo. En este campo se pueden almacenar tipos simples o complejos de datos.

DER: es el campo donde se almacena la dirección del subárbol derecho del nodo T.

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, p. 196):





3.1.3. Operaciones con árboles binarios de búsqueda

Las operaciones de acuerdo con su empleo generarán Estructuras de Árboles de Búsqueda, Binarios, B+, equilibrados y degenerados (con una sola rama, una subrama y un nodo terminal) y de aquí comienza su recorrido, partiendo del Nodo Raíz para ir por las ramas, nodos internos, hasta llegar a los nodos Terminales. Estas operaciones son:

Insertar un elemento en un árbol ABB, eliminar un elemento de un árbol ABB, buscar un elemento en un árbol ABB, comprobar si un árbol está vacío, contar el número de nodos, calcular la altura de un árbol.

Implantación de un árbol binario de búsqueda (ABB)

A continuación, se propone la implementación en Lenguaje C de algunas de las operaciones fundamentales para árboles ABB. Para esta implantación, sólo necesitamos una estructura para referirnos, tanto a cualquiera de los nodos como al árbol completo. La declaración de tipos es la siguiente:

```
typedef struct _nodo {
  int dato;
  struct _nodo *derecho;
  struct _nodo *izquierdo;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Arbol;
```

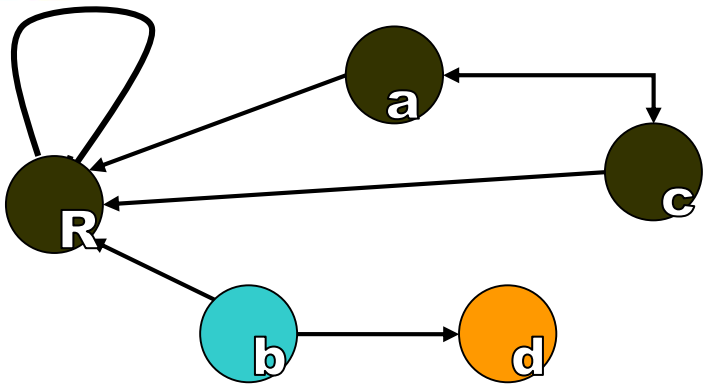
Como ejercicio personal, puedes integrar los códigos que te sugiere la siguiente página: http://c.conclase.net/edd/?cap=007c, para **insertar** un elemento y **eliminar** un elemento de un árbol. Posteriormente obtener toda la aplicación completa para comprender mejor su funcionamiento.

Con la implementación de estas funciones básicas para ABB pasamos al tema de **grafos**.

3.2. Grafos

“Un Grafo Simple G = (V,E) consiste en un conjunto de V de vértices y un conjunto posiblemente vacío E de aristas (edges), siendo cada arista un conjunto de dos vértices de V” (Drozdek, 2007, p. 376).

Los grafos “son Estructuras de Datos no lineales, en las cuales cada elemento puede tener cero o más sucesores y cero o más predecesores. Están formados por nodos (vértices: representan información) y por arcos (aristas: relaciones entre la información) (Guardati, 2007, p. 391).



Representación de un Grafo

Tipos de Grafos (véase, Sedgewick, 2000, pp. 515-547 y Guardati, 2007, pp. 393-446)

Definición del tipo de dato abstracto grafo

Los datos contienen, en algunos casos, relaciones entre ellos que no son necesariamente jerárquicas. Por ejemplo, supongamos que unas líneas aéreas realizan vuelos entre las ciudades conectadas por líneas, la estructura de datos que refleja esta relación recibe el nombre de grafo. (Véase, Cairó y Guardati, 2006, pp. 277-328).

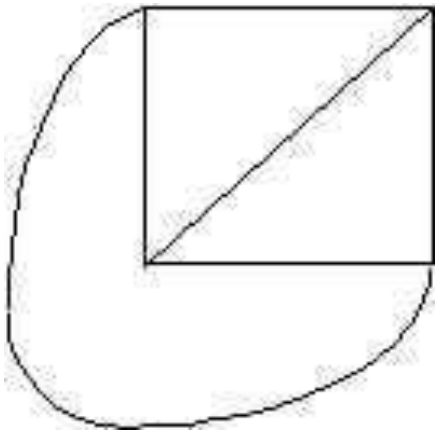
Se suelen usar muchos nombres al referirnos a los elementos de una estructura de datos. Algunos de ellos son “elemento”, “ítem”, “asociación de ítems”, “registro”, “nodo” y “objeto”. El nombre que se utiliza depende del tipo de estructura, el contexto en que usamos esa estructura y quién la utiliza.

En la mayoría de los textos de estructura de datos se utiliza el término “registro” al hacer referencia a archivos y “nodo” cuando se usan listas enlazadas, árboles y grafos.

También un grafo es una terna G = (V, A, j), en donde V y A son conjuntos finitos y j es una aplicación que hace corresponder a cada elemento de A un par de elementos de V. Los elementos de V y de A se llaman, respectivamente, "vértices" y "aristas" de G, y j asocia entonces a cada arista con sus dos vértices.

Esta definición da lugar a una representación gráfica, en donde cada vértice es un punto del plano, y cada arista es una línea que une a sus dos vértices.

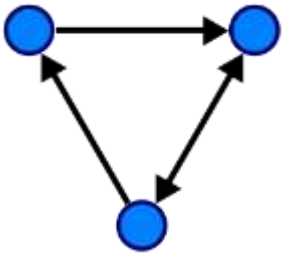
Si el dibujo puede efectuarse sin que haya superposición de líneas, se dice que G es un grafo plano. Por ejemplo, el siguiente es un grafo plano:



Enlaces equivalentes en un nodo del Grafo

3.2.1. Grafos dirigidos

En un grafo generalizado, su dirección puede estar especificada o no. Por el contrario, en el grafo dirigido o también conocido como **digrafo**, el conjunto de sus aristas tienen una dirección definida y está determinado por un par de conjuntos G=(V,A) donde V es un conjunto vacío llamado vértices o nodos y A es un conjunto de pares ordenados de elementos de V llamados aristas o arcos que van del primer al segundo nodo dentro del par.



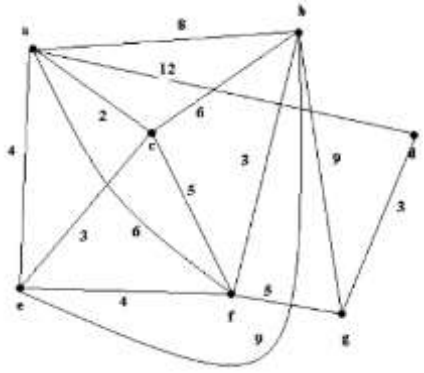
Véase en: http://commons.wikimedia.org/wiki/File:Directed.svg

3.2.2. Grafos ponderados

Los grafos ponderados son muy útiles para medir las distancias en un mapa. Por ejemplo 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.

Un grafo ponderado o grafo con valores o pesos es un grafo G (V, E), en el que a cada arista se le asigna un valor real no negativo o peso. Sobre el conjunto de aristas se

introduce una función peso (w: E->R^+) donde el peso de un subgrafo de un grafo ponderado es la suma de los pesos de todas sus aristas.



Véase:
http://docencia.udea.edu.co/regionalizacion/teoriaderedes/informaci%F3n/C3_minimos.pdf

3.2.3. Operaciones con grafos

Búsqueda en grafos

Para efectuar una búsqueda de los vértices de un grafo, se pueden emplear dos estrategias diferentes, (véase, Sánchez, 2008), búsqueda en profundidad o en anchura:

Búsqueda en profundidad (BEP)	Se comienza en cualquier vértice y en cada paso se avanza a un nuevo vértice adyacente siempre que se pueda. Cuando todos los adyacentes a X hayan sido visitados, se retrocede al vértice desde el que se alcanzó X y se prosigue. Así se consigue etiquetar (visitar) todos los vértices del componente conexo en que se encuentre el vértice inicial. Esta técnica se utiliza cuando necesitamos encontrar respuesta a un problema sobre un grafo sin condiciones de optimización.

	La idea en general de la búsqueda en profundidad comenzando en un nodo A es la siguiente: Primero examinamos el nodo inicial A. Luego examinamos cada nodo N de un camino P que comience en A; o sea, procesamos un vecino de A, luego un vecino de un vecino de A y así sucesivamente, hasta llegar a un punto muerto o final del camino P, y de aquí volvemos atrás por P hasta que podamos continuar por otro camino P´ y así sucesivamente. Este algoritmo es similar al del recorrido en orden de un árbol binario, y también a la forma en que se debe pasar a través de un laberinto. Observa que se hace uso de una pila en lugar de una cola, y éste es el detalle fundamental que hace la diferencia para realizar la búsqueda en profundidad.
	Implantación de un grafo Algoritmo para la búsqueda en profundidad: Este algoritmo realiza la búsqueda en profundidad el grafo G comenzando en un nodo A. 1. Inicializar todos los nodos al estado de preparado. (ESTADO=1) 2. Meter el nodo inicial A en la pila y cambiar su estado a estado de espera. (ESTADO=2). 3. Repetir los pasos 4 y 5 hasta que la pila esté vacía.

	4. Sacar el nodo N en la cima de la pila. Procesar el nodo N y cambiar su estado al de procesado. (ESTADO=3). 5. Meter en la pila todos los vecinos de N que estén en estado de preparados. (ESTADO=1) y cambiar su estado a estado de espera (ESTADO=2). [fin de bucle del paso 3] 6. Salir.
Búsqueda en anchura (BEA)	A diferencia con la BEP ahora se visitan todos los vecinos de un vértice antes de pasar al siguiente. Por tanto no hay necesidad de retroceder. Una vez etiquetados todos los vecinos de un vértice X, se continúa con el primer vértice alcanzado después de X en la búsqueda. Esta técnica se utiliza para resolver problemas en los que se pide hallar una solución óptima entre varias. En general la búsqueda en anchura comenzando de un nodo de partida A es la siguiente: Primero examinamos el nodo de partida A. Luego examinamos todos los vecinos de A y así sucesivamente. Con el uso de una cola, garantizamos que ningún nodo sea procesado más de una vez y usando un campo ESTADO que nos indica el estado actual de los nodos. Algoritmo para la búsqueda en anchura

	Este algoritmo realiza la búsqueda en anchura en un grafo G comenzando en un nodo de partida A. 1. Inicializar todos los nodos al estado de preparados. (ESTADO=1). 2. Poner el nodo de partida A en la COLA y cambiar su estado a espera. (ESTADO=2). 3. Repetir pasos 4 y 5 hasta que la COLA esté vacía. 4. Quitar el nodo del principio de la cola, N. Procesar N y cambiar su estado a procesado. (ESTADO=3). 6. Añadir a COLA todos los vecinos de N que estén en estado de preparados. (ESTADO=1) y cambiar su estado al de espera (ESTADO=2). [fin del bucle del paso 3] 7. Salir.

Caminos mínimos en grafos

Para lograr el propósito del recorrido mínimo dentro de un grafo G, es necesaria para nuestro caso en particular (puesto que no es la única técnica existente), la utilización del algoritmo de Warshall -para el camino mínimo-se expresa de la forma siguiente:

Sea G un grafo con m nodos, $\upsilon 1$, $\upsilon 2$, ..., υm supóngase que queremos encontrar la matriz de caminos P para el grafo G. Warshall dio un algoritmo para

este propósito que es mucho más eficiente que calcular las potencias de la matriz de adyacencia A y aplicar la proposición:

$$B_m = A + A^2 + A^3 + \ldots + A^m$$

donde sea A la matriz de adyacencia y P = Pij la matriz de caminos de un grafo G entonces, Pij = 1 si y solo sí hay un número positivo en la entrada ij de la matriz

Este algoritmo de Warshall se usa para calcular el camino mínimo y existe un algoritmo similar para calcular el camino mínimo de G cuando G tiene peso.

Véase en: http://www.monografias.com/trabajos/grafos/grafos.shtml
nota: tomado del libro Estructura de datos, serie schaum Mcgraw-Hill, pagina: 322, capitulo: 8 Grafos y sus aplicaciones, autor: Seymour Lipschutz.

Implantación de Grafos

Como vimos, un grafo es un conjunto de nodos o vértices que se encuentran relacionados con unas aristas. Estos vértices tienen un valor y, en ocasiones, las aristas también, conocido como *costo.*

Aprovechando la implementación gráfica de los **Applets** de Java en sus clases en la librería de java swing (**import javax.swing.*;**), desarrolla una aplicación que dibuje un grafo en el esquema de la Programación Orientada a Objetos implementando tres clases: **Principal** (será la clase principal), **PanelDibujo** (dibuja el grafo) y **Grafo** (almacena el valor de cada grafo) apoyándote del ejemplo que se propone en la siguiente página de internet:

Página: http://www.myjavazone.com/2010/12/estructura-de-datos-grafos.html

Las clases quedarán en un paquete (carpeta) llamado **Clases**.

```
// Clase Principal

package Clases;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Vector;

import javax.swing.*;

public class Principal extends JApplet{

 PanelDibujo pd;
 int max=0;
 JTextField valor;

 public void init(){

  pd=new PanelDibujo();
  add(pd);

  JPanel pdatos=new JPanel();

  JButton agregar=new JButton("Agregar Nodo");
  agregar.addActionListener(new ActionListener(){


   @Override
   public void actionPerformed(ActionEvent e) {
    if(max<10){
     try{
      Grafo gf=new Grafo(""+Integer.parseInt(valor.getText()));
      pd.getVgrafos().add(gf);
      pd.repaint();
      repaint();
      max++;
     }catch(NumberFormatException ne){
      JOptionPane.showMessageDialog(null, "Digite un numero valido");
     }
```

```
   }
  }
 });

 valor=new JTextField(5);
 pdatos.add(new JLabel("Valor Vertice" +    ""));
 pdatos.add(valor);
 pdatos.add(agregar);
 add(pdatos,BorderLayout.SOUTH);

 }
}
```

```
// Clase PanelDibujo

package Clases;
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.util.Vector;

import javax.swing.*;

public class PanelDibujo extends JPanel {
 int x=150;
 int y=150;
 int ancho=30;
 int alto=30;
 public Vector<Integer> xvs;
 public Vector<Integer> yvs;
 public Vector<Grafo> vgrafos;
 int indice=0;

 public PanelDibujo(){
  vgrafos=new Vector();  xvs=new Vector<Integer>();
  yvs=new Vector<Integer>();
  setDoubleBuffered(true);
 }

 public void paintComponent(Graphics grafico){
  super.paintComponents(grafico);
  Graphics2D g=(Graphics2D)grafico;
  if(vgrafos.size()!=0){
   g.setColor(Color.WHITE);
   g.fillRect(0, 0, getWidth(), getHeight());
```

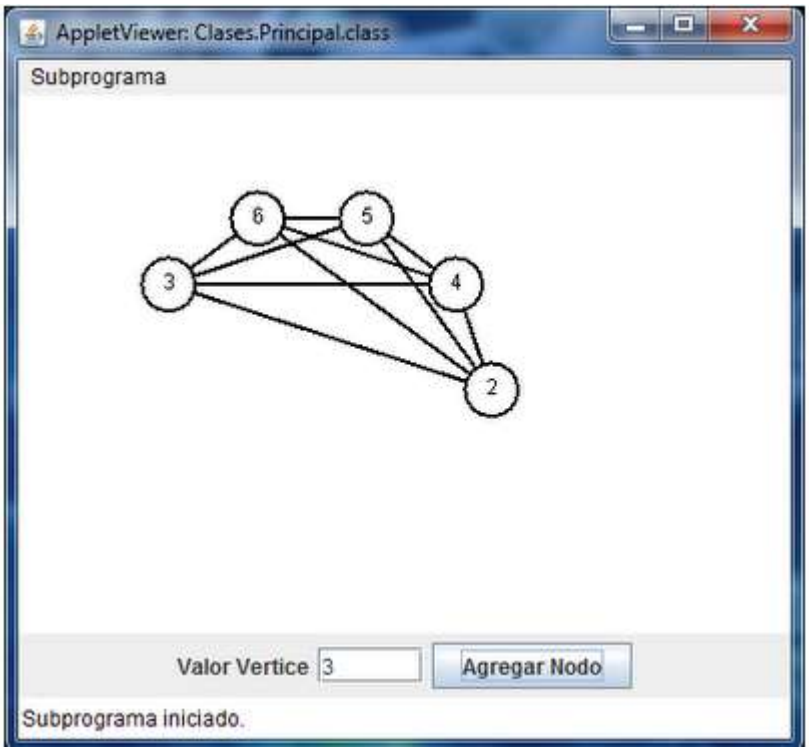
```
   g.setColor(Color.BLACK);
   int radio = 100;
   float angulo = 360/10;
   angulo = (float) Math.toRadians(angulo);
   for(int i=indice;i<vgrafos.size();i++){
    int xv=(int)(x+radio*Math.cos(i * angulo));
    int yv=(int) (y- radio * Math.sin(i * angulo));
    xvs.add(xv);
    yvs.add(yv);
    indice++;
   }
  }
  for(int i=0;i<vgrafos.size();i++){
   for(int j=0;j<vgrafos.size();j++){
    g.setStroke(new BasicStroke(2));
    g.setColor(Color.BLACK);
    g.drawLine(xvs.get(i)+15,yvs.get(i)+15,xvs.get(j)+15,yvs.get(j)+15);
    g.setColor(Color.WHITE);
    g.fillOval(xvs.get(i), yvs.get(i), ancho, alto);
    g.setColor(Color.BLACK);
    g.drawOval(xvs.get(i),yvs.get(i), ancho, alto);
    g.drawString(""+vgrafos.get(i).obtenerDato(),xvs.get(i)+((ancho/2)-3),
yvs.get(i)+((alto/2)+3));
    g.setColor(Color.WHITE);
    g.fillOval(xvs.get(j), yvs.get(j), ancho, alto);
    g.setColor(Color.BLACK);
    g.drawOval(xvs.get(j),yvs.get(j), ancho, alto);
    g.drawString(""+vgrafos.get(j).obtenerDato(),xvs.get(j)+((ancho/2)-3),
yvs.get(j)+((alto/2)+3));
   }
  }
 }
 public Vector<Grafo> getVgrafos() {
  return vgrafos;
 }
 public void setVgrafos(Vector<Grafo> vgrafos) {
  this.vgrafos = vgrafos;
 }
}
```

```
// Clase Grafo
package Clases;
import java.util.Vector;
```

```
public class Grafo {
 private String dato;
 public Grafo(String s){
  dato=s;
 }

 public String obtenerDato(){
  return dato;
 }
}
```

Véase en: http://www.myjavazone.com/2010/12/estructura-de-datos-grafos.html

Una imagen correspondiente a esta implementación sería la siguiente:



RESUMEN

En esta unidad hemos definido las estructuras de árbol e identificado sus elementos y los tipos de árboles ofrecidos por los compiladores más comunes, de igual modo, entendimos que un árbol binario T se define como un conjunto finito de elementos, llamados nodos, de forma que:

T es vacío (en cuyo caso se llama árbol nulo o árbol vacío)

ó

T contiene un nodo distinguido R, llamado raíz de T

Y los restantes nodos de T forman un par ordenado de árboles binarios disjuntos T1 y T2. Las Operaciones, de acuerdo con su empleo, generarán estructuras de árboles de búsqueda, binarios, B+, equilibrados y degenerados (con una sola rama, una subrama y una nodo terminal), y de aquí comienza su recorrido, partiendo del *nodo raíz* para ir por las ramas, nodos internos, hasta llegar a los nodos terminales.

Por otra parte, también abordamos el tema de los grafos, que se han definido como Estructuras de Datos no lineales, en las cuales cada elemento puede tener cero o más sucesores, y cero o más predecesores; están formados por nodos (vértices: representan información) y por arcos (aristas: relaciones entre la información). La teoría de Grafos se aplica hoy en día en muchos campos, tales como en Internet, redes sociales, etc., ya que cada computador es un vértice y la conexión entre ellos son las aristas; además se puede usar para hallar la ruta más corta en empresas de transporte y en muchas otras áreas.

BIBLIOGRAFÍA

SUGERIDA

Autor	Capítulo	Páginas
Lipschutz (1988)	8	337 y ss.
Guardati (2007)	7	--

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

OBJETIVO PARTICULAR

Al concluir la unidad, el alumno identificará los diferentes métodos para la clasificación de datos, identificará sus 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. Intercambio Directo (*Bubblesort*)
4.2. Intercambio Inverso
4.3. *Shaker Sort*
4.4. Inserción Directa
4.5. Selección Directa
4.6. *Shell*
4.7. *Quick Sort*
4.8. Criterios de selección de métodos de ordenamiento

INTRODUCCIÓN

Una de las operaciones más importantes en el manejo de datos es su ordenamiento. Los algoritmos de ordenamiento nos permiten, como su nombre lo dice, ordenar o clasificar la información. El *sort,* ordenación, es reagrupar un grupo de datos en una secuencia específica de orden ya sea de mayor a menor o menor a mayor. Muchas de estas operaciones las vemos en las hojas de cálculo como Excel, en páginas de Internet y Bases de Datos. Nos centraremos en los métodos más populares, analizando la cantidad de comparaciones que suceden, el tiempo que demora y ejemplificando con pseudocódigo o código en lenguaje de programación.

4.1. Intercambio directo (*Bubble Sort*)

El *bubble sort*, también conocido como ordenamiento burbuja, funciona de la siguiente manera: Se va comparando cada elemento del arreglo con el siguiente; si un elemento es mayor que el que le sigue, entonces se intercambian; esto producirá que en el arreglo quede como su último elemento, el más grande. Este proceso deberá repetirse recorriendo todo el arreglo hasta que no ocurra ningún intercambio. Los elementos que van quedando ordenados ya no se comparan. "Baja el más pesado".

Consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados este método también es conocido como intercambio directo.

Ejemplo:
Sea un arreglo de 6 números de empleados: {40,21,4,9,10,35}:

Primera pasada:

{21,40,4,9,10,35} <-- Se cambia el 21 por el 40.
{21,4,40,9,10,35} <-- Se cambia el 40 por el 4.
{21,4,9,40,10,35} <-- Se cambia el 9 por el 40.
{21,4,9,10,40,35} <-- Se cambia el 40 por el 10.
{21,4,9,10,35,40} <-- Se cambia el 35 por el 40.

Segunda pasada:

{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.

Ya están ordenados, pero para comprobarlo habría que acabar esta segunda comprobación y hacer una tercera.

Para su implementación generalmente definimos una función donde A=arreglo y N=tamaño como sigue:

```
int bubblesort(int A[],int N){
      int i,j,AUX;
      for(i=2;i<=N;i++){ //avanza
            for(j=N;j>=i;j--){ //retrocede
                  if(A[j-1]>A[j]){  //si a > p intercambio
                        AUX=A[j-1];
                        A[j-1]=A[j];
                        A[j]=AUX;
                     }
               }
         }
      return 1;
   }
```

En esta implementación de la función *bubblesort* utilizamos un *for* anidado con los índices i y j para ir comparando y ordenando los elementos durante el recorrido del arreglo.

Implementación completa en Lenguaje C++ del *BubbleSort*

```
// Ordenamiento por el método de Bubble Sort
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <conio.h>

 using namespace std; //inicializa librerias std (standares) en c++

int llenavector(int A[],int N){
        int c;
        int x;
         cout<<"Ingrese 10 numeros de empleados:"<<endl;
         for(c=1;c<=N;c++){
                cin>>x;
                A[c]=x;
           }
        return 1;
    }


int bubblesort(int A[],int N){
        int i,j,AUX;
        for(i=2;i<=N;i++){
              for(j=N;j>=i;j--){
                    if(A[j-1]>A[j]){
                          AUX=A[j-1];
                          A[j-1]=A[j];
                          A[j]=AUX;
                       }
                 }
           }
        return 1;
    }

int salida(int A[],int N){
        int c;
        for(c=1;c<=N;c++){
              printf("%d, ",A[c]);
           }
        return 1;
    }

int main()
```

```
{
	int A[10];
	llenavector(A,10);
	printf("ORDENAMIENTO BUBBLE SORT \n");
	printf("Numeros a ordenar: \n");
	salida(A,10);
	printf("\n\nNumeros ordenados: \n");
	bubblesort(A,10);
	salida(A,10);
	getch();
}
```

El programa lee un arreglo de 10 elementos, los muestra desordenados y al final los ordena invocando a la función llamada *bubblesort*.

Análisis de eficiencia del método de intercambio directo

Este método es el más fácil de implementar; sin embargo con mayor volumen de datos es el más ineficiente.

En el método *bubble Sort* tendremos consecutivas comparaciones entre pares de números de tal manera que en la primera pasada tendremos (n-1) comparaciones y en la segunda (n-2) y así sucesivamente hasta llegar a 2 y 1 comparaciones entre elementos dependiendo de tamaño del arreglo. Expresado en otros términos, esta operación se reduce a: BS = (n2 -n)/2 donde BS es el método *bubble Sort*. El número de elementos a ordenar dependen de si el arreglo está ordenado, desordenado o en orden inverso. Esto es: Para un arreglo ordenado = 0, para uno desordenado = 0.75 * (n2 - n) y para uno con orden inverso será = 1.5 * (n2 - n). El tiempo necesario para ejecutar el algoritmo *Bubble Sort* es proporcional a n2, donde *n* es el número de elementos del arreglo. Mientras más elementos contenidos en el arreglo, mayor será el número de comparaciones y menor la calidad de este método.

4.2. Intercambio inverso

El método por intercambio inverso es otra modalidad de intercambio, así tenemos que:

Ordenando los elementos del arreglo usando el método *bubblesort*, se transporta en cada pasada el elemento más pequeño a la parte izquierda del arreglo A de N elementos.

Un ejemplo de pseudocódigo para este proceso sería:

```
bubblesort1(A,N)
Inicio
        Declarar i,j,aux:entero
        Para i = 2 hasta N haga
                Para j = i hasta 2 inc (-1) haga
                        Si (A[j-1]>A[j]) entonces
                                Aux = A[j-1]
                                A[j-1] = A[j]
                                A[j] = aux
                        Fin si
                Fin para
        Fin para
Fin
```

Intercambio inverso contrario al anterior

El intercambio inverso es una mejora al método *bubble Sort* que consiste en la modalidad de que en cada pasada del arreglo el valor más grande se lleva a la derecha en lugar de la izquierda. Se tienen dos etapas, en la primera se trasladan los elementos más pequeños hacia la izquierda almacenando en una variable el último elemento intercambiado. En la segunda, se trasladan los elementos más grandes hacia la parte

derecha del arreglo almacenando en otra variable la posición del último elemento intercambiado.

Un pseudocódigo para este proceso sería:

```
bubblesort2(A,N)
Inicio
    Declarar i,j,aux:entero
    Para i = 1 hasta N-1 haga
        Para j = 1 hasta N-i haga
            Si (A[j]>A[j+1]) entonces
                Aux = A[j]
                A[j] = A[j+1]
                A[j+1] = aux
            Fin si
        Fin para
    Fin para
Fin
```

Traduce este pseudocódigo a un programa en lenguaje C/C++ llamado sortInverso.cpp en el compilador de tu preferencia.

4.3. *Shaker Sort*

La idea básica de este algoritmo consiste en mezclar las dos formas en que se puede realizar el método *bubblesort.*

El método *Shaker Sort*, también lo puedes encontrar como *"Sacudida"* o *"Cocktail"* y es una versión mejorada del método *Bubble Sort*, ya que utiliza un algoritmo de intercambio que le permite alterar ambos extremos del arreglo a la vez, con lo cual, en el primer paso ordena los datos del arreglo de derecha a izquierda y se trasladan los elementos más pequeños hacia la parte izquierda del arreglo, almacenando en una variable la posición del último elemento intercambiado. En el siguiente paso se efectúa un ordenamiento contrario y se trasladan los elementos más grandes hacia la parte derecha del arreglo, almacenando en otra variable la posición del último elemento intercambiado y así sucesivamente hasta ordenar la lista completa.

El código de la función *Shaker Sort* queda de la siguiente manera en lenguaje C.

```
void ShakerSort (itemType a[], int N)
{
 int i, j, izq, der, aux;
  izq = 2; der = N; j = N;
  do
  {
   for(i = der; i >= izq; i--)
      if (a[i - 1] > a[i])
       { swap (a, i, i -1);
         j = i;
```

```
      }
    izq= j + 1;
    for(i = izq; i <= der; i++)
     if (a[i -1] > a[i])
        { swap (a, i, i -1)
          j = i;
        }
     der= j - 1;
    } while (izq <= der);
}
```

Como ejercicio personal, desarrolla un programa en C/C++ que contenga esta función la cual puedes consultar en la siguiente ruta de Internet.

http://es.scribd.com/doc/40292011/Metodos-de-to-en-c

4.4. Inserción directa

El método de la baraja o inserción directa se basa en tomar el primer elemento de la parte izquierda desordenada del arreglo y busca la posición que le corresponde dentro de la sección ordenada. Se toma el primer elemento de la parte no ordenada y se almacena en una variable auxiliar (aux). Se compara empezando por el final de la parte ordenada, hasta que se encuentra un elemento menor. Entonces se desplaza una posición a la derecha, todos los elementos que han resultado mayores que el que queremos insertar y se coloca el valor de la variable auxiliar (aux) en el lugar encontrado.

A partir de éste código desarrolla un programa correspondiente al Método de inserción directa (InsercionDirecta.cpp) y pruébalo en el compilador de C/C++ de tu preferencia.

```
int array[N];
int i,j,aux;

// Define un arreglo de 10 números
// Dar valores a los elementos del arreglo

for(i=1;i<N;i++) // i contiene el número de elementos de la sublista.
{ // Se intenta añadir el elemento i.
  aux=array[i];
  for(j=i-1;j>=0;j--) // Se recorre la sublista de atrás a adelante para buscar
  { // la nueva posición del elemento i.
    if(aux>array[j]) // Si se encuentra la posición:
    {
      array[j+1]=aux; // Ponerlo
      break; // y colocar el siguiente número.
    }
    else // si no, sigue buscándola.
      array[j+1]=array[j];
  }
  if(j==-1) // si se ha mirado todas las posiciones y no se ha encontrado la correcta
   array[0]=aux; // es que la posición es al principio del todo.
}
```

4.5. Selección directa

El proceso de este algoritmo consiste en buscar el menor elemento en el arreglo y colocarlo en primera posición. Luego se busca el segundo elemento más pequeño del arreglo y se coloca en segunda posición. El proceso continúa hasta que todos los elementos del arreglo hayan sido ordenados. El método se basa en los siguientes principios:

1. Seleccionar el menor elemento del arreglo.
2. Intercambiar dicho elemento con el primero.
3. Repetir los pasos anteriores con los (n-1), (n-2) elementos y así sucesivamente hasta que sólo quede el elemento mayor.

Se requiere hacer el ordenamiento de tres claves de empleados con el método de selección directa: 44, 52, 11, 37, 49, por lo tanto, n=5 claves

1ª pasada: En esta pasada se busca entre los últimos elementos el menor de todos, y lo intercambiaremos con la primera posición.
44, 52, 11, 37, 49→ Para buscar el menor, necesitaremos una instrucción *for* que recorra los *n* últimos elementos.
44, 52, 11, 37, 49 → El menor es el 11, colocado en tercera posición.
44, 52, 11, 37, 49 → Lo intercambiamos con el de la primera posición.
11, 52, 45, 37, 49 → Ya tenemos uno en orden. Nos quedan los n-1 últimos.
2ª pasada: En la segunda pasada buscamos entre los últimos *n-1* (es decir, 4) elementos el menor de todos, y lo intercambiaremos con la segunda posición.
11, 52, 44, 37, 49 → Recorremos los cuatro últimos y el menor es el 37.
11, 37, 44, 52, 49 → Lo intercambiamos con la segunda posición y ya hay dos en orden.

3ª pasada: En esta pasada buscamos entre los últimos *n-2* (es decir, 3) elementos el menor de todos, y lo intercambiaremos con la tercera posición.

11, 37, 44, 52, 49 → El menor es el 44, en tercera posición.

11, 37, 44, 52, 49 → El 44 ya estaba en 3ª posición, así que al intercambiarlo con él mismo, se queda como está. Ya tenemos tres en orden.

4ª y última pasada: buscamos entre los últimos n-3 (es decir, 2) elementos el menor de todos, y lo intercambiaremos con la cuarta posición.

11, 37, 44, 52, 49 → El menor es el 49, en quinta posición.

11, 37, 44, 49, 52 → Lo intercambiamos con la cuarta posición. Ya hay cuatro en orden.

11, 37, 44, 49, 52 → El último está necesariamente en orden también.

Las claves quedarán ordenadas.

Vease: http://latecladeescape.com/algoritmos/1122ordenacion-por-seleccion-directa-selectionsort

Con base en lo anterior, podemos implementar en Lenguaje C o C++ la siguiente función:

```
void seleccion_directa(int n)
{
 int i,j,min,k;
 int cambio;
 for(i=0;i<n;i++)
  {
    min=arr[i];
    k=0;
    cambio=0;
    for(j=i+1;j<n;j++)
      {
       if (arr[j]<min)
         {
           min=arr[j];
           k=j;
          cambio=1;}
```

```
        }
      if (cambio )
        {
          arr[k]=arr[i];
          arr[i]=min;
        }
   }
}
```

4.6. Shell

A continuación comentamos el método *Shell sort*.

El método *Shell Sort* también lo puedes encontrar como método de inserción con incrementos decrecientes creado por Donald Shell y consiste en realizar múltiples recorridos al arreglo y en cada pasada ordena un número igual de elementos.
Cada elemento se compara con los que le siguen a su izquierda. Si el elemento a insertar es menor, se tiene que ejecutar muchas comparaciones antes de colocarlo en su lugar definitivamente. El método *Shell* agiliza los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño y con eso se obtiene una ordenación más rápida. El método se basa en tomar como salto *N/2* (siendo N el número de elementos) y luego reduciendo a la mitad el arreglo en cada repetición hasta que el salto o distancia vale 1.

Ejemplo: Supongamos que tenemos que ordenar un vector de claves de 6 empleados por el Método *Shell Sort*.

60, 11, 55, 22, 33, 44, 10

En el primer recorrido se realizan 3 saltos reordenando la lista como:
22, 11, 44, 55, 10, 33, 55, 60 haciendo los intercambios: (60,22), (55,44), (60,10)

En el segundo recorrido se realizan 3 saltos reordenando la lista como:
10, 11, 44, 22, 33, 55, 60 haciendo los intercambios: (22,10)

En el tercer recorrido se realizan 3 saltos reordenando la lista como:
10, 11, 44, 22, 33, 55, 60 haciendo ningún intercambio

En el cuarto recorrido se realizan 1 saltos reordenando la lista como:
10, 11, 22, 33, 44, 55, 60 haciendo los intercambios: (44,22), (44,33)

En el quinto recorrido se realizan 1 saltos reordenando la lista como:
10, 11, 22, 33, 44, 55, 60 haciendo ningún intercambio por lo tanto el vector quedará ordenado.

Referente a la eficiencia de este método podemos decir que, el tamaño del conjunto de datos utilizados en las comparaciones tiene un impacto significativo en la eficiencia del algoritmo. Algunas implementaciones de este algoritmo tienen una función que permite calcular el tamaño óptimo del *set* de datos para un arreglo determinado. Su implementación original, requiere O(n2) comparaciones e intercambios en el peor caso; sin embargo, el *Shell Sort* es una versión mejorada del ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada. Los pasos múltiples sobre los datos, se hacen con tamaños de espacio cada vez más pequeños. El último paso del *Shell Sort* simplemente es un ordenamiento por inserción; pero para entonces, ya está garantizado que los datos del arreglo están ordenados.

A partir de éste fragmento de código en C/C++ transfórmalo en una función llamada *ShellSort* e implementa el programa ShellSort.cpp para un vector de 10 claves de empleados y pruébalo en el compilador de C/C++ de tu preferencia.

```
int array[N];
int salto,cambios,aux,i;

for(salto=N/2;salto!=0;salto/=2) // El salto va desde N/2 hasta 1.
  for(cambios=1;cambios!=0;) // Mientras se intercambie algún elemento:
  {
    cambios=0;
    for(i=salto;i<N;i++) // se da una pasada
      if(array[i-salto]>array[i]) // y si están desordenados
      {
        aux=array[i]; // se reordenan
        array[i]=array[i-salto];
        array[i-salto]=aux;
        cambios++; // y se cuenta como cambio.
      }
  }
```

Véase en: http://es.scribd.com/doc/48004260/ESTRUCTURA-de-DATOS-III-Algoritmos-De-Ordenamiento

4.7. *Quick sort*

El método *Quick sort* o de ordenación rápida utiliza un algoritmo recursivo, es decir, implementa una función que se llama a sí misma. El *Quick sort* es un algoritmo del estilo *divide y vencerás*. El algoritmo de recursión consiste en una serie de cuatro pasos:

1. Si hay menos de un elemento por ordenar, retorna inmediatamente (termina).
2. Tomar un elemento del vector que sirve como “muestra”.
3. Dividir el arreglo en dos partes, una con los elementos mayores y una con los elementos menores a la muestra.
4. Repite recursivamente el algoritmo para las dos mitades del arreglo original hasta que queda ordenado.

Véase en http://es.scribd.com/doc/48004260/ESTRUCTURA-de-DATOS-III-Algoritmos-De-Ordenamiento

Ejemplo de la función *Quicksort* recursiva.

```
void Quicksort_Recursivo(int ini, int fin)
{
 int izq,der,x,aux;
 x=arr[ini];
 izq=ini;der=fin;
 while(izq<der)
  {
   while(arr[izq]<=x &&izq<der) izq++;
   while(arr[der]>x)
   der--;
    if (izq<der)
      {
        aux= arr[izq];
        arr[izq]=arr[der];
```

```
        arr[der]=aux;
       }
    }
     arr[ini]=arr[der];
     arr[der]=x;
     if(ini<der-1) Quicksort_Recursivo (ini,der-1);
      if(der+1 <fin) Quicksort_Recursivo (der+1,fin);
    }
   }

}
```

4.8. 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 lleva a cabo 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.

Por último, podemos considerar el cálculo de la eficiencia del método.

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 teniendo una complejidad O(n log2 n) siendo posiblemente el algoritmo más rápido.

BIBLIOGRAFÍA

SUGERIDA

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

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

OBJETIVO PARTICULAR

Al concluir la unidad, 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 por transformación de llaves (*Hash*)
  - 5.3.1. Funciones *Hash*
  - 5.3.2. Resolución de colisiones
- 5.4. Búsqueda en árboles binarios

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, y permite la recuperación de datos previamente almacenados. El tipo de búsqueda se puede clasificar como interna o externa, según el lugar en el que esté almacenada la información (en memoria o en dispositivos externos). 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.

Centrándonos en la búsqueda interna. Como principales algoritmos de búsqueda en arreglos tenemos la búsqueda secuencial, la binaria y la búsqueda utilizando tablas de *hash*.

5.1. Búsqueda secuencial

El método de búsqueda secuencial consiste en 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.

La implementación en lenguaje C de este algoritmo es la siguiente:

```
for(i=j=0;i<N;i++)
  if(array[i]==elemento)
  {
    buscado[j]=i;
    j++;
   }
```

Este algoritmo se puede optimizar cuando el arreglo está ordenado, en cuyo caso la condición de salida cambiaría a:

```
for(i=j=0;array[i]<=elemento;i++)
```

o cuando sólo interesa conocer la primera ocurrencia del elemento en el arreglo.

```
for(i=0;i<N;i++)
  if(array[i]==elemento)
    break;
```

En este último caso, cuando sólo interesa la primera posición, se puede utilizar un centinela (bandera); esto es, dar a la posición siguiente al último elemento de arreglo el valor del elemento, para estar seguros de que se encuentra el elemento, y no tener que comprobar a cada paso si seguimos buscando dentro de los límites del arreglo:

```
array[N]=elemento;
for(i=0;;i++)
  if(array[i]==elemento)
    break;
```

Si al acabar el bucle, i vale N es que no se encontró el elemento. El número medio de comparaciones que hay que hacer antes de encontrar el elemento buscado es de (N+1)/2.

5.2. Búsqueda binaria

La búsqueda binaria divide un arreglo por su elemento medio en dos subarreglos más pequeños y compara el elemento a buscar con este elemento medio. Si estos elementos coinciden, la búsqueda termina. Si el elemento es menor, entonces se busca en el primer subarreglo, y si es mayor se buscará en el segundo subarreglo.

El método de búsqueda binaria requiere que el arreglo de datos se encuentre ordenado y es más eficiente que el de búsqueda secuencial ya que este 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.
Ejemplo: Se requiere buscar la clave 33 de un empleado por el método de búsqueda binaria en un arreglo de 9 elementos. 10, 22, 33, 40, 55, 66, 77, 80, 99.
El primer paso toma el elemento central que es 55 y se divide en dos

El elemento buscado 33 es menor que el central 55, debe estar en el primer subarreglo: 10, 22, 33, 40

Se divide el subrrango en dos quedando el 22

Como el elemento buscado es mayor que el central, debe estar en el segundo subarreglo: 33, 40

Se vuelve a dividir este subrrango y como el elemento buscado coincide con el central 33, lo hemos encontrado.

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

Eficiencia de este método.

Este método es más eficiente que el de búsqueda secuencial o lineal para casos de grandes volúmenes de datos y para mejorar su velocidad se puede implementar en forma recursiva, siendo una función que divide el arreglo en dos más pequeños elementos.

El método realiza *log(2,N+1)* comparaciones antes de encontrar el elemento, o antes de descubrir que no está. Este número es muy inferior que el necesario para la búsqueda lineal para casos grandes.

Este método también se puede implementar de forma recursiva, siendo la función recursiva la que divide al arreglo en dos más pequeños.

A partir del ejemplo de la página:
http://mygnet.net/codigos/c/metodos_de_busqueda/busqueda_binaria_de_forma_recursiva_sobre_un_vector_ordenado_dot_muy_completo.2936, implementa un programa de Búsqueda binaria en forma recursiva sobre un vector ordenado en lenguaje C en el compilador de tu preferencia.

5.3. Búsqueda por transformación de llaves (Hash)

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 a cada elemento mediante una transformación de elementos, esto se hace mediante una función de conversión llamada función *hash*. Hay diferentes funciones para transformar el elemento y el número obtenido es el índice del elemento.

Elementos	índices	Transformación
José Luis	78	José Luis-5589343464
María	55	María-5577003423
Pedro	12	Pedro-5598334388
Raúl	42	Raúl-5589233390

Un ejemplo de una tabla Hash que asocia elementos con teléfonos.

En este ejemplo el nuevo índice obtenido nos permite el acceso a las personas por su número telefónico almacenados a partir de una clave especial generada.

5.3.1. Función Hash

La función *Hash* tiene como entrada un conjunto de cadenas que transforma en un rango de salida finito, normalmente cadenas de longitud fija. La idea básica de un valor

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.

Funciona transformando la clave con una función *hash* en un *hash*, un número que identifica la posición donde la tabla *hash* localiza el valor deseado.

La manera más simple de transformar el elemento es asignarlo directamente; es decir, al 0 le corresponde el índice 0, al 1 el 1, y así sucesivamente, pero cuando los elementos son muy grandes, empezamos a tener problemas con el manejo de la información, ya que se puede desperdiciar espacio de almacenamiento y la recuperación de datos se vuelve más lenta y compleja. La función de *hash* ideal debería ser biyectiva; esto es, que a cada elemento le corresponda un índice, y que a cada índice le corresponda un elemento; pero no siempre es fácil encontrar esa función, e incluso a veces es inútil, ya que se puede no saber el número de elementos a almacenar. La función de *hash* depende de cada problema y de cada finalidad, y se pueden utilizar con números o cadenas.

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. Aritmética modular; en esta función el índice de un número es resta de la división de ese número entre un número N prefijado, preferentemente primo. Mitad del cuadrado; Consiste en elevar al cuadrado la clave y coger las cifras centrales. Truncamiento; Consiste en ignorar parte del número y utilizar los elementos restantes como índice. Plegamiento; Consiste en dividir el número en diferentes partes, y operar con ellas (normalmente con suma o multiplicación). Estas funciones sólo son sugerencias y que con cada problema se puede implementar una nueva función hash que incluso se puede crear con base en un algoritmo.

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 asignación de una misma dirección a dos o más claves diferentes. Cuando sucede este conflicto entre claves, entonces se busca el mejor método de resolución de colisiones que lleve a la solución de este problema.

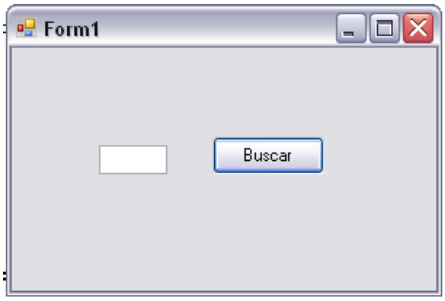
Hay diferentes maneras de resolución de colisiones, una de ellas es crear un arreglo de apuntadores, donde cada apuntador señale el principio de una lista enlazada. De esta manera, cada elemento que llega a incorporarse, se pone en el último lugar de la lista de ese índice con una clave nueva. El tiempo 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 al elemento. El proceso de búsqueda concluye cuando el elemento es hallado, o bien cuando se encuentra una posición vacía. Por ejemplo si la posición 97 ya estaba ocupada, el registro con clave 54405-97 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.

Implementación de Búsqueda por transformación de llaves Hash

La implementación utiliza el paradigma orientado a objetos, modo gráfico, en la versión de Visual Studio Lenguaje de Programación C Sharp o C#. Como podrás observar, en este ejemplo la sintaxis es muy similar a C y Java.

En el ambiente de programación de Visual Studio crea una ventanita con las siguientes características:



Name=textBox1 Text=Buscar

Da Clic en el botón e ingresa el siguiente código en C Sharp.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace Hash
{
   public partial class Form1 : Form
   {
      private int[] datos = new int[5] {1679, 4567, 1165, 0435, 5033 };
      private int[] hash = new int[7];
      private int[] enlace = new int[7];
      public Form1()
      {
         InitializeComponent();
         for (int b = 0; b <= 6; b++)
         {
            enlace[b] = -9999;
         }
         //Reacomodo por hash
         int r, aux=0;
         for (int i = 0; i <= 4; i++)
```

```
        {
            r= datos[i] % 7;
            if (datos[i] == 0)
            {
                hash[r] = datos[i];
            }
            else
            {
                for(int s=0;s<=6;s++)
                {
                    if(hash[s]==0)
                    {
                        aux=s;
                    }
                }
                hash[aux]=datos[i];
                enlace[r] = aux;
            }
        }
    }

    private void button1_Click(object sender, EventArgs e)      {
        int temp,r;
        temp = int.Parse(textBox1.Text.ToString());
        r = temp % 7;
        if (temp == hash[r])
        {
            MessageBox.Show("Se encuentra en \nel renglon:" + r.ToString(),
"Resultado");
        }
        else
        {
            while(enlace[r] != -9999)
            {
                if (temp == hash[enlace[r]])
                {
                    MessageBox.Show("Se encuentra en \nel renglon:" +
enlace[r].ToString(), "Resultado");
                    r = enlace[r];
                }
            }
        }
    }
  }
}
```

Ejecuta tu aplicación.

Imágenes del programa corriendo:

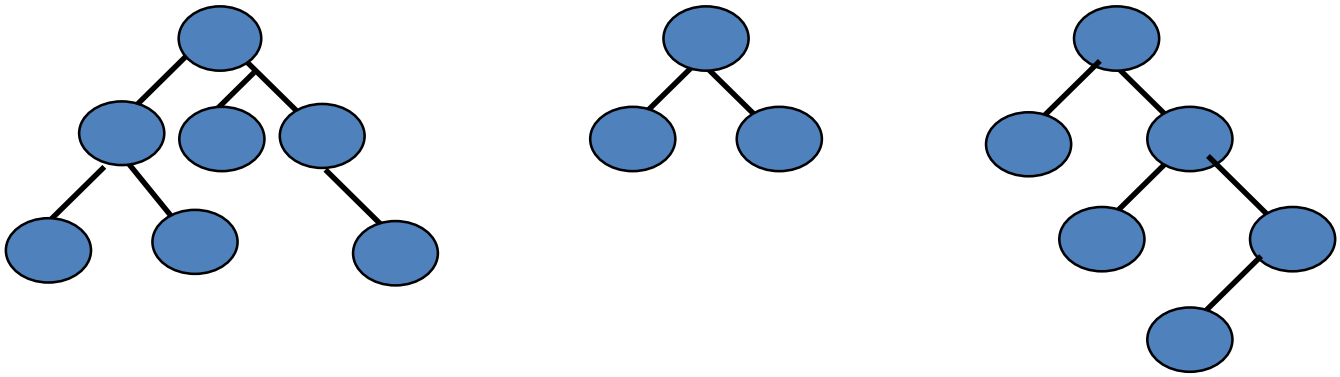
5.4. Búsqueda en árboles binarios

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

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.

Ejemplo:



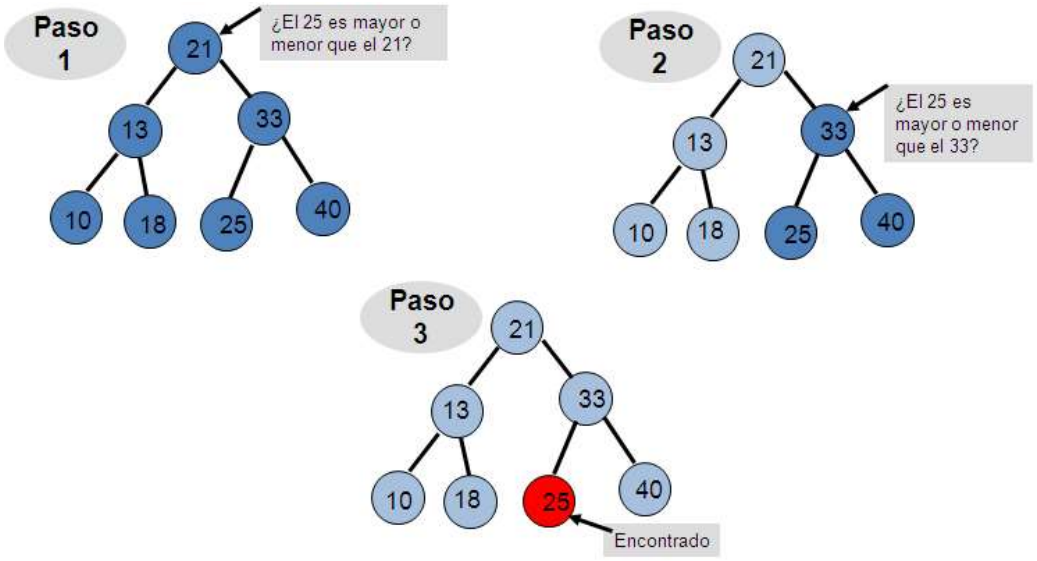
Árbol Binario de Búsqueda (ABB)

Este tipo de árbol permite almacenar información ordenada.

Reglas a cumplir:

- Cada nodo del árbol puede tener 0, 1 ó 2 hijos.
- Los descendientes izquierdos deben tener un valor menor al padre.
- Los descendientes derechos deben tener un valor mayor al padre.

Ejemplo de recorrido para buscar el nodo 25.



En el ejemplo, se requieren tres pasos para encontrar el nodo deseado. En caso de encontrar el elemento, se regresará el valor buscado dependiendo de si se encontró en el descendiente izquierdo o derecho. En caso contrario, se regresará *FALSE* para indicar que no se encontró el nodo, como se muestra en la función de C llamada **Buscar**.

```
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, 1990

----------. (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, Aaron M. (1993). *Estructuras de datos en C*. México: Prentice Hall.

Facultad de Contaduría y Administración
Sistema Universidad Abierta y Educación a Distancia