ESCUELA DE CIENCIAS BÁSICAS E INGENIERÍA

Ingeniería de Sistemas

ASIGNATURA: Estructura de Datos

CORPORACIÓN UNIVERSITARIA REMINGTON

DIRECCIÓN PEDAGÓGICA

Este material es propiedad de la Corporación Universitaria Remington (CUR), para los estudiantes de la CUR en todo el país.

2011

CRÉDITOS

El módulo de estudio de la asignatura Estructura de Datos del Programa Ingeniería de Sistemas es propiedad de la Corporación Universitaria Remington. Las imágenes fueron tomadas de diferentes fuentes que se relacionan en los derechos de autor y las citas en la bibliografía. El contenido del módulo está protegido por las leyes de derechos de autor que rigen al país.

Este material tiene fines educativos y no puede usarse con propósitos económicos o comerciales.

AUTOR

Edilberto Restrepo Restrepo Magíster en Educación Superior de la Pontificia Universidad Javeriana, Especialista en Gerencia de Proyectos de la Universidad del Tolima, Administrador de Empresas y Tecnólogo en Administración de Redes de Datos. Con 18 Años de experiencia como docente y Diseñador de asignaturas en las áreas de Lógica y Programación, Mantenimiento de computadores, Administración de Redes de Datos, asesoría de proyectos informáticos y productivos de emprendimiento. En instituciones como CESDE, Universidad San Martin, Universidad Cooperativa de Colombia, Comfenalco, Comfama y Microempresas de Antioquia. Actualmente coordinador de la Unidad de formación en la corporación Universitaria Remington. Con ponencias sobre procesos evaluativos, Cesde 2005, Aprendizajes colaborativos, Remington, 2011. Actualmente miembro del grupo de investigación AVICUR
mgedirre@gmail.com
formaciondocente.coordinador01@remington.edu.co

Nota: el autor certificó (de manera verbal o escrita) No haber incurrido en fraude científico, plagio o vicios de autoría; en caso contrario eximió de toda responsabilidad a la Corporación Universitaria Remington, y se declaró como el único responsable.

RESPONSABLES

Escuela de Ciencias Básicas e Ingeniería
Director. Mauricio Sepúlveda

Director Pedagógico
Octavio Toro Chica
dirpedagogica.director@remington.edu.co
Coordinadora de Medios y Mediaciones
Angélica Ricaurte Avendaño
mediaciones.coordinador01@remington.edu.co

GRUPO DE APOYO

Personal de la Unidad de Medios y Mediaciones
EDICIÓN Y MONTAJE
Primera versión. Febrero de 2011.

Derechos Reservados

Esta obra es publicada bajo la licencia CreativeCommons. Reconocimiento-No Comercial-Compartir Igual 2.5 Colombia.

TABLA DE CONTENIDO

1. MAPA DE LA ASIGNATURA 7

2. ARREGLOS O VECTORES 8

2.1. Mapa Conceptual 8

2.2. Arreglos o Vectores Unidimensionales 9

2.3. Arreglos Multidimensionales o Matrices 30

2.4. Registros y Arreglos de Registros 38

3. MANEJO DINÁMICO DE LA MEMORIA 44

3.1. Mapa Conceptual 45

3.2. Estructuras de Listas 46

3.3. Estructuras en Pilas o Lifo 73

3.4. Estructura de Colas o Fifo 78

4. ESTRUCTURAS EN ARBOL 83

4.1. Mapa Conceptual 84

4.2. Árboles en General 85

4.3. Árboles Binarios 90

4.4. Pistas de Aprendizaje 101

4.5. Glosario 102

4.6. Bibliografía 103

4.7. Fuentes digitales o electrónicas 103

1. MAPA DE LA ASIGNATURA

ESTRUCTURA DE DATOS

PROPÓSITO GENERAL DEL MÓDULO

Que el estudiante se apropie de las diferentes estructuras de almacenamiento de datos en memoria y las aplique de forma eficiente, óptima y efectiva en la solución de problemas mediante el desarrollo de software.

OBJETIVO GENERAL

Desarrollar destrezas para la implementación de soluciones óptimas y efectivas en el manejo de la información mediante el uso de estructuras de almacenamiento en memoria.

OBJETIVOS ESPECÍFICOS

- Inducir al estudiante para que apropie y maneje estructuras unidimensionales, multidimensionales y de registros para el almacenamiento de información.
- Promover en el estudiante proposiciones y formas de manejo dinámico de memoria, mediante el uso de listas, pilas y colas.
- Guiar el estudiante en la aplicación de los procedimientos para el uso de estructuras jerárquicas en árbol con el fin de automatizar los procesos que así lo requieran.

UNIDAD 1
ARREGLOS

Por medio de análisis de casos de la vida cotidiana que presenten necesidades de sistematización de información, el estudiante
Comprenderá y aplicara adecuadamente las formas de acceso y manejo de variables compuestas, arreglos de una, varias dimensiones y de registros para solución de necesidades diversas.

UNIDAD 2
MANEJO DINÁMICO DE LA MEMORIA

Usando grupos de trabajo colaborativo, que permitan la discusión y proposición de plantanteamientos lógicos para soluciones de desarrollo informático, los estudiantes Plantearan diseños efectivos para el manejo dinámico de la memoria, usando estructuras de almacenamiento en listas, pilas y colas para contextos diferentes.

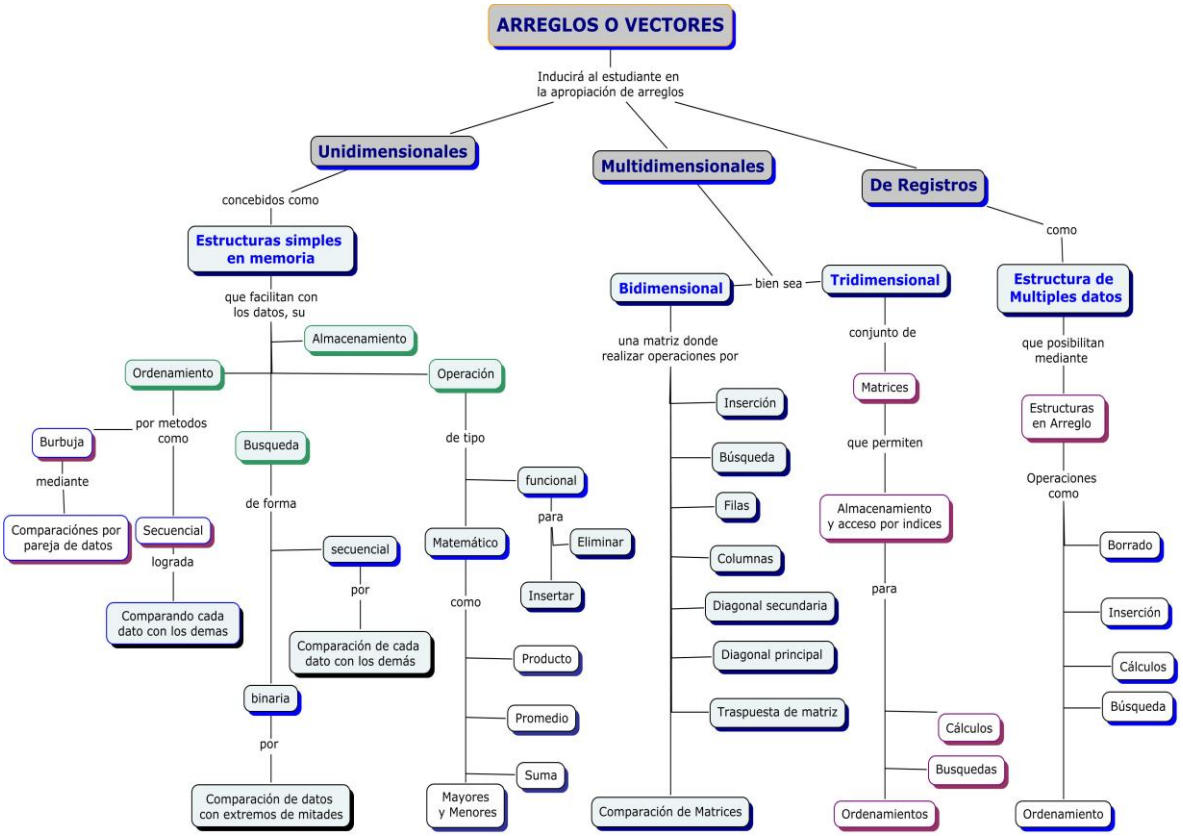
UNIDAD 3
ARBOLES

Con el estudio de casos los grupos de estudiantes, encontrarán situaciones cotidianas, familiares, comerciales e industriales, mediante las cuales se evidencie su capacidad en el manejo de estructuras de datos jerarquizadas en arboles de diferentes características y formas, supliendo exigencias diversas de los escenarios abordados.

2. ARREGLOS O VECTORES

En muchas situaciones es necesario procesar una colección de valores que están relacionados entre sí por algún método, por ejemplo, una lista de calificaciones, una serie de temperaturas medidas a lo largo de un mes, etc. el procesamiento de tales conjuntos de datos utilizando variables simples, puede ser extremadamente difícil, es por ello que la mayoría de los lenguajes de programación incluyen soporte para la implementación de estructuras de almacenamiento de datos que se basan en conceptos matemáticos de "vector" y "matriz". Un Arreglo (matriz, tabla, arreglo) es una secuencia de posiciones en memoria central a las que se puede acceder directamente, que contiene datos del mismo tipo y pueden ser seleccionados individualmente mediante el uso de subíndices. Este capítulo estudia el concepto de Arreglos unidimensionales y multidimensionales, como el procesamiento de los mismos.

2.1. Mapa Conceptual



OBJETIVO GENERAL

Inducir al estudiante para que apropie y maneje estructuras unidimensionales, multidimensionales y de registros para el almacenamiento de información en memoria y su aplicación en contextos diversos.

OBJETIVOS ESPECÍFICOS

- Adquirir destrezas en el manejo de datos en un arreglos unidimensional
- Manipular datos en estructuras de almacenamiento multidimensional
- Plantear soluciones de almacenamiento mediante el uso de arreglos de registros

Prueba Inicial

Teniendo en cuenta sus conceptos adquiridos en asignaturas previas a estructura de datos y orientadas al manejo de la lógica de programación y estructuras de almacenamiento en memoria, plantee algunas respuestas al siguiente cuestionario, con fines de autodiagnóstico.

1. ¿Qué entiende usted por un vector o arreglo de datos en memoria?
2. ¿En qué casos particulares del desarrollo de software es necesario o conveniente usar vectores?
3. ¿Qué concepción tiene usted sobre lo que es una matriz de almacenamiento de datos en memoria?
4. Crees que las matrices de 3 dimensiones tienen alguna aplicabilidad en la vida cotidiana?, explique su respuesta.
5. ¿Qué tipo de estructuras lógicas de programación crees, se deban utilizar para el manejo de una matriz?
6. Qué conceptos tienes acerca de las estructuras de almacenamiento basadas en registros.

2.2. Arreglos o Vectores Unidimensionales

Definición

Un arreglo unidimensional es una estructura conformada por una colección finita y ordenada de datos del mismo tipo. Es la estructura natural para modelar listas de elementos iguales. El tipo de acceso a los arreglos unidimensionales es el acceso directo, es decir, podemos acceder a cualquier elemento del arreglo sin tener que consultar a elementos anteriores o posteriores, esto mediante el uso de un índice para cada elemento del arreglo que nos da su posición relativa.

Un Arreglo (vector) es un conjunto finito y ordenado de elementos homogéneos. Ordenado significa que el elemento primero, segundo, tercero,..., enésimo de un arreglo puede ser identificado. Sus elementos son homogéneos, es decir, del mismo tipo de datos. De esta forma todos sus elementos pueden ser de tipo entero de tipo texto, etc. Los arreglos se conocen también como matrices o tablas

El tipo más simple de arreglos es el unidimensional o vector (matriz de una dimensión). Un vector de una dimensión denominado Datos que consta de N elementos se puede representar así.

Pedro	Juan	David	……..	…….	Olga	Claudia
Datos[1]	Datos[2]	Datos[3]	……..	……..	Datos[n-1]	Datos[N]

Y en memoria estaría reservándose una posición equivalente de casillas para almacenar dichos datos.

Cada Arreglo debe tener un **nombre** cualquiera, que identifique todas las casillas como un conjunto de datos y un **índice** que hace referencia a cada una de las posiciones del Vector y el **dato** que se refiere a la información existente en cada posición, bien sea de tipo entero, texto u otro.

Así: El Nombre del Arreglo es DATOS, los índices son los números de los paréntesis que pueden ser reemplazados por una variable que dentro de una estructura cíclica tome diferente valores secuenciales y el dato es cada uno de los nombres que aparece en las casillas respectivas a los índices.

De esta forma para referirnos a un dato que está en una posición especifica del vector, debemos hacer lo siguiente:

Nombre del Arreglo [posición en la que se encuentra] ósea, Datos [3] hace referencia al dato “David” y Datos [n-1] se refiere a dato “Olga”

Observe que para hacer referencia al índice o posición de cada casilla del arreglo usamos el corchete dentro del cual va el respectivo valor.

Ejemplo:
Sea Numeral un arreglo de 20 posiciones, para almacenar las 20 primeras letras del abecedario español. Lo representaríamos de la siguiente manera.

Nombre del vector o arreglo: NUMERAL [1…20] de text

Posición[1]	Posición[2]	Posición[3]	Posición [4]						Posición 20
A	B	C	D	E	F	…….		…….	S

Cada componente del arreglo o dato, se accede por medio de un indicie entre 1 y 20 que son las posiciones existentes de la estructura. Así:
Numeral [3] se refiere a la letra C
Numeral [1] se refiere a la letra A
Numeral [4] se refiere a la letra D

Numeral [20] se refiere a la letra S

Nota: Algunos lenguajes de programación admiten que los índices de acceso a los datos estén conformados no únicamente por secuencias numéricas sino también por secuencias de literales a, b, c, d,…….z, o secuencias conocidas como los meses ene, feb, mar,…….dic.

De esta forma podríamos acceder a los datos correspondientes de cada vector usando secuencias como índices de posición en el vector.

Posición[ene]	Posición[feb]	Posición[mar]	Posición [Abr]						Posición [dic]
A	B	C	D	E	F			……	L

Numeral [ene] se refiere a la letra A
Numeral [mar] se refiere a la letra C
Numeral [abr] se refiere a la letra D

Numeral [dic] se refiere a la letra L

1. Operaciones con Arreglos

a. Construcción del Arreglo

Para construir un Arreglo en memoria es preciso desarrollar un algoritmo que defina el nombre del Arreglo y el tipo de datos que almacenará el vector, así como su alcance o tamaño, es decir el número de posiciones o datos que soportará dicha estructura. Para ilustrar dicha construcción consideremos el siguiente algoritmo, que llamaremos meses y guardaremos los nombres de los meses en las posiciones del arreglo.

Algoritmo meses

Inicio
Meses[1..12]:string, i: entero, Mes: string
para i desde 1 hasta 12 con variación + 1
Lea mes
meses[i]=mes
fin(para)
Fin

Observe que:
Inicia el algoritmo

- Se definen las variables a utilizar entre ellos el arreglo con su cantidad de posiciones y se
- define también los tipos de datos que almacenarán cada posición del arreglo y cada variable.
- Se inicia el ciclo para que con el indicie variado por i pasará por cada posición del arreglo.
- Luego se pide que se lea mes, es decir que se ingres un mes cualquiera
- Para luego asignarlo a la posición i del vector mediante la variable mes.
- Luego se cierra el ciclo para y el vector quedará lleno con los meses según se han digitado.

Ejercicio:
Construye un vector para que almacene todos los nombres y otro para almacenar las edades de sus compañeros de clase.

b. Impresión Del Arreglo

Nota: Haremos referencia de aquí en adelante a los subprogramas como elementos a desde los cuales se realizan procedimientos cortos y repetitivos para un programa principal, de tal forma que deben recibirse y enviarse parámetros puntuales según la necesidad, pero ello no implica que cada conjunto de instrucciones no pueda ser ejecutada directamente como un programa principal independiente.

Ya hemos visto en nuestro algoritmo anterior que cuando se termina la construcción de un vector interesa conocer cuál es el valor almacenado en cada una de las posiciones del vector. Para efectuar esta tarea invocamos un subprograma denominado imprimeMeses, El cual presentamos a continuación:

```
ImprimeMeses(Meses,12)
i: entero
        para i desde 1 hasta 12 con variacion + 1
          imprima i, Meses[i]
        Fin(para)
Fin
```

El subprograma envía los parámetros son el nombre del vector y el tamaño del vector.
Solo se requiere la variable **i** local para recorrer las diferentes posiciones del vector.
Con el ciclo **para** nos movemos en el vector desde la posición 1 hasta la posición 12, imprimiendo el subíndice y el mes de dicha posición.

Ejercicio:
Construye un vector para imprimir todos los nombres y las edades de los compañeros de clase, almacenadas en los vectores anteriores.

c. Suma De Los Datos De Un Vector O Arreglo

Es también muy frecuente determinar el total de los datos almacenados en un vector. Para ello definimos un subprograma función, al cual llamamos Suma Pagos, que efectué esta suma y retorne el resultado de ella al programa principal o llamante:

Realizaremos las instrucciones necesarias para llenar un vector con pagos digitados al azar y luego las instrucciones para sumar sus contenidos. Llamaremos al arreglo pagos,

```
SumaPagos(Pagos,N)
N, i:integer  s, P: real
      Para i desde 1 hasta N variando +1
            Lea P
            Pagos[i]=P
       Fin(para)
 S=0
    Para i desde 1 hasta N variando +1
          S=s + pagos[i]
    Fin(para)
    Imprima S
Fin o Retorne
```

Ejercicio:
Recorra el arreglo edades que creo anteriormente y calcule el promedio de las edades.

d. Hallar El Menor Y El Mayor Valor De Un Arreglo

Ya hemos visto que en un arreglo se almacenan los datos correspondientes a un conjunto de situaciones o eventos. Podemos conocer cuál es el valor mayor o cual es el valor menor almacenado en el arreglo.

Para conocer cuál es el valor mayor existente en el arreglo y cuál es su posición podemos aplicar el siguiente algoritmo.

En busca del mayor valor y su posición

```
MayorPago(Pagos)
Variables: mayor: real , p, i: entero
INICIO
    Mayor= 0
    Para i desde 1 hasta N con variación + 1
        SI Pagos[i]>Mayor
            Mayor= Pagos[i]
            P=i
        FinSi
    FinPara
    Imprima Mayor, P
FIN
Retorne(mayor, P)
```

El algoritmo inicia con el nombre del subprograma en caso que se desee invocar desde un programa principal, el cual enviará el nombre del vector pagos y su número de posiciones N.

- Se definen las variables: mayor para guardar el valor que va quedando de mayor en cada comparación, P que guardará la posición del arreglo donde se encuentra el mayor e i que es la variable que incrementará el ciclo.
- Dentro del ciclo para ante cada valor tomado por la variable i se pregunta si el arreglo en la posición actual es mayor que mayor, que inicio en 0, si lo es se graba dicho valor en la variable mayor y la posición en la variable P. este procesos se repita hasta terminar de recorrer y comparar todas las posiciones del arreglo.

- Se termina la estructura SI y el ciclo para.
- Se imprime la variable mayor con el dato que haya quedado y e igual forma la variable P.
- Se termina el programa y en caso que sea un subprograma se retornan los parámetros.

Ejercicio

En grupos de 4 estudiantes en trabajo colaborativo, se debe proponer que diseñen un programa que invoque subprogramas diferentes para llevar a cabo las siguientes tareas.

Recorrer el arreglo edades que se creó anteriormente y localice e imprima la edad menor y la posición donde se encuentra en el vector. Luego imprima todas las posiciones del vector con su respectiva posición para verificar lo realizado.

Localice un dato específico y su posición en un arreglo.

Multiplique los datos de dos Arreglos A y B uno a uno llevando el resultado a la respectiva posición de en un arreglo C.

Limpie las posiciones de un arreglo, es decir elimine sus datos, bien sea para un arreglo numérico o tipo texto.

e. Insertar Un Dato En Un Arreglo Ordenado Ascendentemente

Para buscar donde insertar un nuevo dato en un vector ordenado ascendentemente debemos recorrer el vector e ir comparando el dato de cada posición con el dato a insertar. Como los datos están ordenados ascendentemente, cuando se encuentre en el vector un dato mayor que el que se va a insertar esa es la posición en la cual deberá quedar el nuevo dato.

Si el dato a insertar es mayor que todos los datos del arreglo, entonces el dato a insertar quedara de último. Para ello basta con recorrer todo el vector comparando cada dato con el dato a insertar y al llegar a la posición n asignar al arreglo en posición N+1, el dato a insertar, de igual forma si el dato es tipo texto.

Consideremos la situación en la cual se debe insertar el 10 en el siguiente vector ordenado y con posiciones libres al final. Le llamaremos **orden**
Procedemos así:
Observe que el arreglo tiene datos ordenados ascendentemente, y el último dato está en la posición **7** y la dimensión del arreglo es **n**

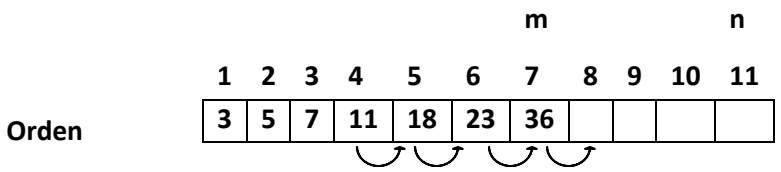


Figura 28.1

El algoritmo deberá iniciar buscando la posición donde deberá insertarse el dato 10, sin que se pierda el orden del arreglo, de esta forma.

```
Inicio
   Dato, m,n,i,j,p: entero
   Dato=10; n=11
  Para i desde 1 hasta n variando +1
       Si Orden[i]>=dato   entonces
       P=i
  FinPara
  M=7
  Para j desde m hasta P variando -1
     Orden [j+1]= Orden[j]
  Fin para
 Orden[p]=dato
 M=m+1
Fin
```

- El algoritmo inicia definiendo las variables necesarias y guardando el dato a insertar.
- El ciclo para, inicia variando i desde 1 hasta n u 11 con incremento de 1
- Dentro del ciclo se pregunta si el dato en cada posición del arreglo es mayor o igual al dato a insertar.
- En el momento en que no sea igual o mayor, se captura la posición donde se encuentra, el valor de i y se termina el Para.
- Con m=7 representando la posición donde está el último dato del arreglo.
- Inicia el nuevo Para que va desde j=m hasta P=i disminuyendo en 1 cada vez
- Dentro del Para se inicia al correr una posición a la derecha cada dato, hasta llegar a la posición P, donde debe insertare el dato
- Finaliza el Para
- Se asigna el dato al arreglo en la posición P.
- y el último dato ahora queda en la posición m+1
- Fin del algoritmo

Información complementaria

Operaciones con arreglos multidimensionales, su forma de acceso, ordenamiento y codificación de sus algoritmos propios. http://es.scribd.com/doc/42404009/Arreglo-Multidimensional, Pagina accedidas el 23 de septiembre de 2011.

Ejercicio:

Implementa un algoritmo para insertar un dato en una posición leída de un arreglo ordenado descendentemente.

Un algoritmo para eliminar una posición específica con su dato, de un arreglo

Para limpiar las posiciones de un arreglo, es decir eliminar sus datos, bien sea para un arreglo numérico o tipo texto.

f. Búsqueda Binaria

Parte de un arreglo con datos ordenados en forma ascendente. La primera comparación se hace con el dato de la mitad del vector. Si no es el dato que se está buscando es porque el dato que se está buscando es mayor o menor que el dato de la mitad del vector. Si el dato que se está buscando es mayor que el dato de la mitad del vector, significa que si el dato se halla en el vector, está a la derecha del dato de la mitad, o sea que no abra necesidad de comparar el dato que se está buscando con todos los datos que están a la izquierda del dato de la mitad del vector. Luego de que hemos descartado la mitad de los datos, la siguiente comparación se hará con el dato de la mitad, de la mitad en la cual posiblemente este el dato a buscar. Así continuamos con este proceso hasta que se encuentre el dato, o se detecte que el dato no está en el vector. Si tuviéramos un millón de datos, con una sola comparación estamos ahorrando 500.000 comparaciones, con la segunda comparación se ahorran 250.000 comparaciones, con la tercera se ahorran 125.000 comparaciones y así sucesivamente. Como se podrá ver la reducción del número de comparaciones es notable. He aquí una implementación e C++
http://www.ingenieriasw.com/busquedabinaria

El algoritmo para lograr este tipo de búsqueda es el siguiente.

1	Entero BusquedaBinaria(Binaria, n, d)
2	Variables: i, j, m: numéricas enteras
3	Inicio
4	i = 1
5	J = n
6	Mientras i <=j
7	m =(i + j)/ 2
8	SI Binaria[m] = d
9	Retorne(m)
10	Fin(SI)
11	SI Binaria[m]<d
12	i = m + 1
13	de_lo_contrario
14	j = m – 1
15	Fin(SI)
16	Fin(MIENTRAS)
17	Retorne(n + 1)
18	Fin
19	Fin (BusquedaBinaria)
20	

1. Se define el subprograma búsqueda_binaria, cuyos parámetros son: **Binaria**, en el cual hay que efectuar la búsqueda; **n**, el tamaño del vector; y **d**, el dato a buscar.

2. Se definen las variables de trabajo: **i**, para guardar la posición inicial del rango en el cual hay que efectuar la búsqueda; **m**, para guardar la posición de la mitad del rango entre **i** y **j**.

4 y 5. Se asignan los valores iníciales a las variables i y j. estos valores iníciales son 1 y n, ya que la primera vez el rango sobre el cual hay que efectuar la búsqueda es desde la primera posición hasta la última del vector.

6. Se plantea el ciclo MIENTRAS, con la condición de que **i** sea menor o igual que **j**. es decir, si el límite inferior (**i**) es mayor o igual que el límite superior (**j**), aún existen posibilidades de que el dato que se está buscando se encuentre en el vector. Cuando **i** sea mayor que **j** significa que el dato no está en el vector y retornara **n** +**1**. Si esta condición 6 es verdadera se ejecutan las instrucciones del ciclo.

7. Se calcula la posición de la mitad entre **i** y **j**. llamamos a esta posición **m**.

8. se compara el dato de la posición m con el dato d. si el dato de la posición m es igual al dato d que se está buscando (Binaria[m] == d), ejecuta la instrucción 9: termina el proceso de búsqueda retornando m, la posición en la cual se halla el dato d en el vector. En caso de que el dato de la posición es verdadero, significa que si el dato está en el vector, se halla a la derecha de la posición m; por consiguiente, el valor inicial del rengo en el cual se debe efectuar la búsqueda es m + 1, y por tanto ejecuta la instrucción 12 en el cual a la variedad i se le asigna m + 1. Si el resultado de la comparación de la instrucción 11 es falso, significa que si el dato está en el vector, se halla a la izquierda de la posición m; por consiguiente, el valor final del rango en el cual se debe efectuar en la búsqueda es m – 1, y por tanto ejecuta la instrucción 14 en la cual a la variable j se le asigna m – 1. Habiendo actualizando el limite inicial o el limite final, se llega a la institución 16, la cual retorna la ejecución a la instrucción 6, donde se efectúa de nuevo la comparación entre i y j. cuando la condición sea falsa se sale del ciclo y ejecuta la instrucción 17, la cual retorna n + 1, indicando que el dato no se halla en el vector.

Fíjese que la única manera de que se ejecute la instrucción 17 es que no haya encontrado el dato que se está buscando, ya que si lo encuentra ejecuta la instrucción 9, la cual termina el proceso.

Consideremos, como ejemplo, siguiente arreglo, y que se desea buscar el número 38 en dicho vector:

															n
	2	3	1	4	5	6	7	8	9	10	11	12	13	14	15
Binaria	6	7	3	10	12	18	22	28	31	37	45	52	60	69	73

Al ejecutar nuestro algoritmo de búsqueda binaria, y cuando se esté en la instrucción 7 por primera vez en la situación en el vector, se ve así:

	i ↓							**m** ↓							**j** ↙ **i**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Binaria	3	6	7	10	12	18	22	28	31	37	45	52	60	69	73

Al ejecutar la instrucción 8, la cual compara el dato de la posición **m (Binaria[m] == 28)** con **d (d == 38)**, se determina que el dato **d**, si está en el vector, se halla a la derecha de la posición **m**, por consiguiente el valor de **i** deberá ser 9. En constancia, al ejecutar las instrucciones 11 y 12, el límite

inicial del rango sobre el cual hay que efectuar la búsqueda es 9, y el rango sobre el cual hay que efectuarla es entre 9 y 15. Quedando de la siguiente forma:

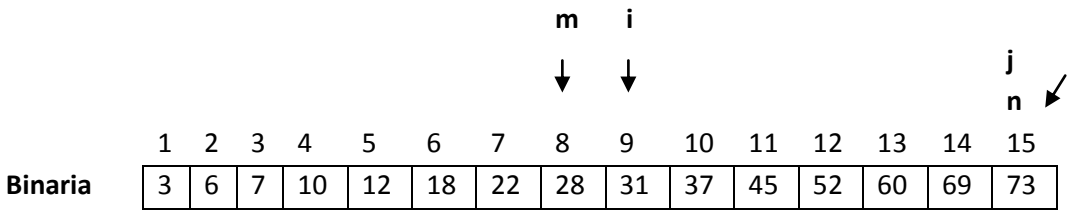
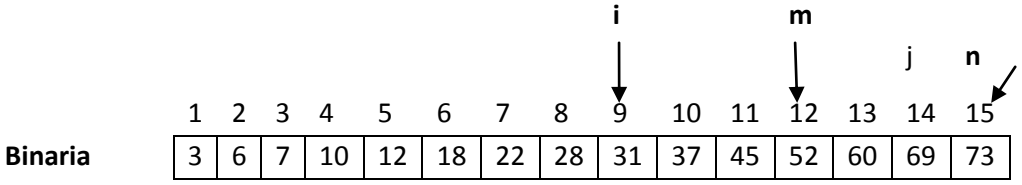
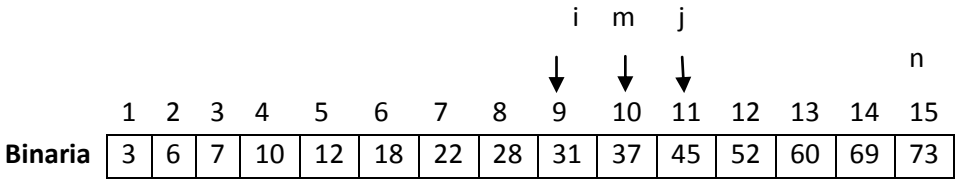


Figura 30.3

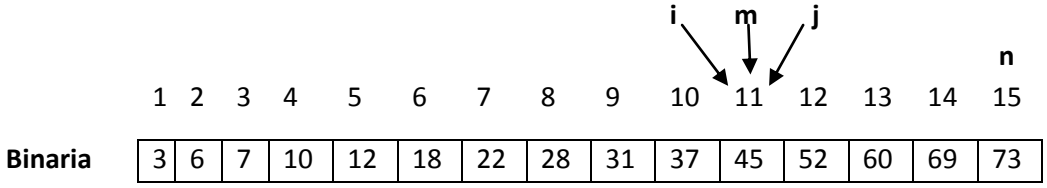
Al ejecutar el ciclo por segunda vez el valor de **m** será 12, y la situación en el vector se muestra así:



Luego, al comparar **d (d == 38)** con **Binaria[m] == 52),** se determina que el dato que se está buscando, si está en el vector, debe hallarse a la izquierda de **m**, es decir, entre las posiciones 9 y 11, por consiguiente el valor de **j** sea 11 (**j == m – 1**).
Al ejecutar el ciclo por tercera vez, la situación del vector es la siguiente:



En esta pasada se detecta que el dato debe estar a la derecha de **m**, por consiguiente el valor de **i** deberá ser 11, y en consecuencia el valor de **m** también. La situación en el arreglo es así:



En este punto la comparación de **d** con Binario**[m]** indica que si el dato **d** está en el vector, debe estar a la izquierda de **m**, por tanto el valor de **j** será 10, y al regresar a la institución 6 y evaluar la

condición del ciclo (**i<= j**), por consiguiente termina el ciclo y retorna en **m** el valor de 16 (**n + 1**), indicando que el 38 no se encuentra en dicho vector.

Otro ejemplo Ilustrativo

```
Datos de entrada:
  vec: vector en el que se desea buscar el dato
  tam: tamaño del vector. Los subíndices válidos van desde 0 hasta tam-1 inclusive.
  dato: elemento que se quiere buscar.

Variables
  centro: subíndice central del intervalo
  inf: límite inferior del intervalo
  sup: límite superior del intervalo

inf = 0
sup = tam-1

Mientras inf <= sup:
  centro = ((sup - inf) / 2) + inf // División entera: se trunca la fracción
  Si vec[centro] == dato devolver verdadero y/o pos, de lo contrario:
   Si dato < vec[centro] entonces:
    sup = centro - 1
   En caso contrario:
    inf = centro + 1
Fin (Mientras)
Devolver Falso
```

Implementado en C

```
#include <iostream>
#include <vector>
bool busqueda_dicotomica(const vector<int> &v, int principio, int fin, int &x){
   bool res;
   if(principio <= fin){
      int m = (principio + fin)/2;
      if(x < v[m]) res = busqueda_dicotomica(v, principio, m-1, x);
      else if(x > v[m]) res = busqueda_dicotomica(v, m+1, fin, x);
      else res = true;
   }else res = false;
   return res;
}
/*{Post: Si se encuentra devuelve true, sino false}*/
```

Información complementaria sobre ordenamiento y búsqueda, puede encontrar en las siguientes páginas. Pseudocódigo, http://barranquillo.ucaldas.edu.co/rgarcia/programacion20031/alordbus.htm, o codificación directa de los mismos. http://eztigma.brinkster.net/algoritmos.html , Paginas accedidas el 11 de octubre de 2011.

Ejercicio:
Implementa un algoritmo y codifícalo en cualquier lenguaje, para localizar por búsqueda binaria un dato leído en cualquier posición.

a. Ordenamiento Secuencial o Por Selección

Se utiliza cuando el vector no está ordenado o no puede ser ordenado previamente. Consiste en buscar el elemento comparándolo secuencialmente, con cada elemento del arreglo hasta encontrarlo, o hasta que se llegue al final. La existencia se puede asegurar cuando el elemento es localizado, pero no podemos asegurar la no existencia hasta no haber analizado todos los elementos del arreglo.

1	Void ordenamiento_ascendente-selección(V,m)
2	Variables: i, j, k: enteras
3	inicio
4	para i desde 1 hasta m – 1 con _variacion + 1
5	K = i
6	para j desde i + 1 hasta m con_variacion + 1
7	si v [j]<v[k]
8	K=j
9	**Fin(SI)**
10	Fin(PARA)
11	Intercambiar(V, k, i)
12	Fin(PARA)
13	FIN
14	Fin(ordenamiento_ascendente_seleccion)

1. Define el subprograma con sus parámetros: V, variable en la cual se almacena el vector que desea ordenar, y m, el número de elementos a ordenar. V es un parámetro por referencia.

1. Define las variables necesarias para efectuar el proceso de ordenamiento. La variable **i** se utiliza para identificar a partir de cual posición es que faltan datos por ordenar. Inicialmente el valor de **i** es 1, ya que inicialmente faltan todos los datos por ordenar y los datos comienzan en la posición 1. Cuando el contenido de la variable **i** sea 2, significa que faltan por ordenar los datos que hay a partir de la posición 2 del vector; cuando el contenido de la variable **i** sea 4, significa que faltan por ordenar los datos que hay a partir de la posición 4; cuando el contenido de la variable **i** sea **m,** significa que estamos en el último elemento del vector, el cual obviamente estará en su sitio, pues no hay más datos con los cuales se puede comparar. Esta es **la** razón por la cual en la instrucción 4 se pone a variar la **i** desde 1 hasta **m – 1.**

2. Se plantea el ciclo de la variable **i,** que varían desde 1 hasta **m -1,** tal como explicamos en el párrafo anterior.

3. Se le asigna a k el contenido de la variable i. la variable k se utiliza para identificar la posición en la que se halla el menor dato. Inicialmente suponemos que el menor dato se encuentra en la posición i del vector, es decir, suponemos que el primero del conjunto de datos que faltan por ordenar.

4. Se plantea el ciclo con el cual se determina la posición en la cual se halla el dato menor del conjunto de datos que faltan por ordenar. En este ciclo se utiliza la variable **j**, que tiene un valor inicial de **i + 1** y varia hasta **m.**

5. Se compara el dato que se halla en la posición j del vector con el dato que se halla en la posición k. si el contenido de la posición j es mayor que el contenido de la posición k, se remplaza el contenido de la variable k por j. de esta manera en la variable k siempre estará la posición en la cual encuentra el dato.

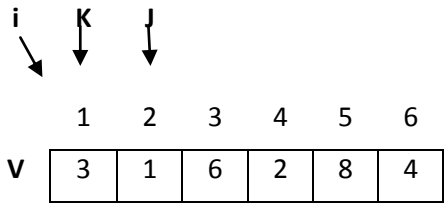
6. Al terminar la ejecución 11 se intercambia el dato que se halla en la posición **k** con el dato que se halla en la posición **i,** logrando de esta manera ubicar el menor dato al principio del conjunto de datos que faltan por ordenar.

Al llegar a la instrucción 12 continúa con el ciclo externo incrementado que es a partir de esa posición que faltan dos por ordenar.

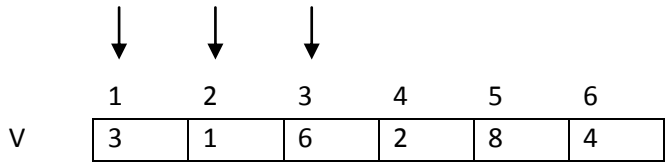
Ejemplo:

1	2	3	4	5	6
3	1	6	2	8	4

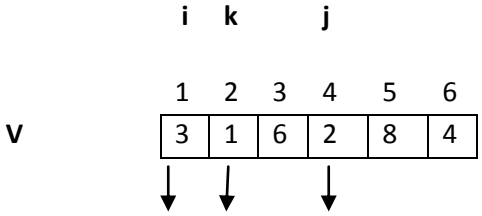
Al ejecutar por primera vez las instrucciones 4, 5 y 6 los valores de **i, k** y **j** son 1 y 2, respectivamente, lo cual indica: faltan por ordenar los datos a partir de la posición 1; el menor dato que faltan por ordenar, se halla en la posición 1 del vector; se va a comenzar a comparar los datos del vector, a partir de la posición 2. Gráficamente se presenta esta situación así:



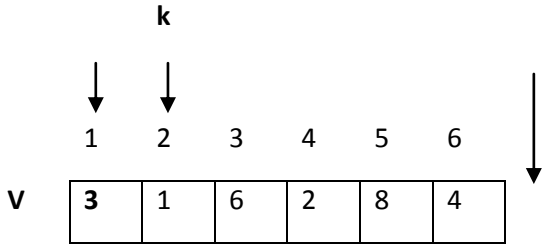
Al ejecutar la instrucción 7, por primera vez, comparara el dato de la posición 2 **(j == 2)**Con el dato de la posición 1 **(k == 1)**, obteniendo como resultado que la condición es verdadera; por tanto, ejecuta la instrucción 8, la cual asigna a **k** ya es 2, indicando que el menor dato se halla en la posición 2 del vector. Al ejecutar la instrucción 10, incrementa el contenido de **j** en 1, quedando **j** con el valor de 3. Esta nueva situación se presenta así:



Se ejecuta de nuevo la instrucción 7 y compara el dato de la posición 3 con el dato de la posición 2, obteniendo como resultado que l condición es falsa; por tanto, no ejecuta la instrucción 8, y continúa en la instrucción 10, es decir, incrementa nuevamente el contenido de **j** en 1, la cual queda valiendo 4. Esta nueva situación se presenta así:

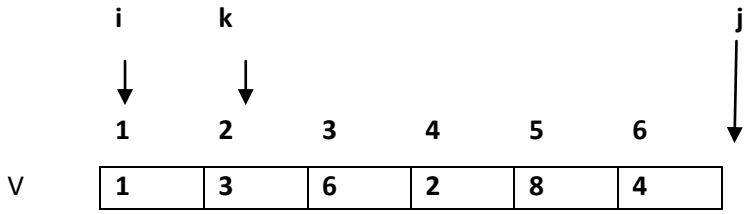


Continua ejecutando las instrucciones del ciclo interno (instrucciones 7 a 10), comparando el contenido de la posición **j** del vector con el contenido de la posición **k,** obteniendo siempre falso como resultado, hasta que el contenido de la variable **j** es 7. Cuando esto sucede, pasa ejecutar la instrucción 1, en la cual se intercambia el dato que se halla en la posición **i** con el dato que se halla en la posición **k**. en este momento la situación del vector se presenta así:



Al ejecutar la instrucción 11 el dato que se hallaba en la posición k queda en la posición i, y el dato que se hallaba en la posición i queda en la posición k.

Nuestro vector queda hasta éste paso de la siguiente manera, y de igual forma debe precederse para logar organizar los datos que faltan.



Otro algoritmo de ejemplo

```
Datos de entrada:
  vec: vector en el que se desea buscar el dato
  tam: tamaño del vector. Los subíndices válidos van desde 0 hasta tam-1 inclusive.
  dato: elemento que se quiere buscar.

Variables
  pos: posición actual en el arreglo

pos = 0
Para pos desde 0 hasta tam
 Si vec[pos] == dato devolver "verdadero" y/o "pos", de lo contrario
 Devolver "falso"
Fin Para
```

Ejercicio:
Implementa un algoritmo y codifícalo en cualquier lenguaje, para ordenar por el método de selección. Los elementos de un arreglo de 20 posiciones, el cual contiene los nombres de igual número de productos de una tienda de abarrotes.

b. ordenamiento Por Burbuja

Este método consiste en comparar dos datos consecutivos y ordenarlos ascendentemente, es decir, si el primer dato es mayor que el segundo se intercambia dichos datos, de lo contrario se dejan tal cual están. Cualquiera de las dos situaciones que se hubiera presentado se avanza en el vector para comparar los siguientes dos datos consecutivos. En general, el proceso de comparaciones es: el primer dato con el segundo, el segundo dato con el tercero, el tercer dato con el cuarto, el cuarto con el dato quinto, y así sucesivamente, hasta comparar el penúltimo dato con el último.

Como resultado de estas comparaciones, y los intercambios que se hagan, el resultado es que en la última posición quedara el mayor dato en el vector.

La segunda vez se compara nuevamente los datos consecutivos, desde el primero con el segundo, hasta que se comparan el antepenúltimo dato con el penúltimo, obteniendo como resultado que el segundo dato mayor queda de penúltimo.

Es decir, en cada pasada de comparaciones, de los datos que faltan por ordenar, se ubica el mayor dato en la posición que le corresponde, o sea de ultimo en el conjunto de datos que faltan por ordenar.

Algoritmo de ejemplo:

1	Vold ordenamiento_ascendente_burbuja(V, n)
2	Variables: i, j: numéricas enteras
3	inicio
4	Para i desde 1 hasta n-1 con _variacion +1
5	Para j desde 1 hasta n –i con_variacin +1
6	SI V[j]>V[j + 1]
7	Intercambiar(V,j, J+1)
8	Fin(SI)
9	Fin(PARA)
10	Fin(PARA)
11	FIN
12	Fin(ordenamiento_ascendente_burbuja)

Asumamos el vector siguiente como ejemplo

1	2	3	4	5	6
1	3	2	6	4	8

Nuestro interés es ordenar los datos de dicho vector en formas ascendente, utilizando el método denominado burbuja.

1	2	3	4	5	6
3	**1**	6	2	8	4
1	**3**	**6**	2	8	4
1	3	**6**	**2**	8	4
1	3	2	**6**	**8**	4
1	3	2	6	**8**	**4**

Primera pasada: cinco comparaciones

Las cincos comparaciones que se efectúan en la primera pasada. La comparación correspondiente a cada pasada se resalta con los bordes más gruesos. A cada comparación la llamaremos **parte** con el fin de evitar redundancias.

En la parte 1. Se compara el dato de la posición 1 con el dato de la posición 2. Como el dato de la posición 1 es mayor que el dato de la posición 2, se intercambian dichos datos, quedando el vector como en la parte 2.

En la parte 2 se compara el dato de la posición 2 con el dato de la posición 3. Como el dato de la posición 2 es mayor que el dato de la posición. 3 los datos se dejan intactos.
En la parte 3 se compara el dato de la posición 3 con el dato de la posición 4. Como el dato de la posición 3 es mayor que el dato de la posición 4 se intercambian dichos datos, quedando el vector como en la parte 4.

En la parte 4 se compara el dato de la posición 4 con el dato de la posición 5. Como el dato de la posición 4 es menor que el dato de la posición 5, los datos permanecen intactos.

En la parte 5 se compara el dato de la posición 5 con el dato de la posición 6. Como el dato de la posición 5 es mayor que el dato de la posición 6, se intercambian dichos datos.

Obteniendo hasta aquí el siguiente orden en los datos., observe que el dato mayor ya está en la última posición que le corresponde.

1	2	3	4	5	6
1	3	2	6	4	**8**

Figura 31.3

En la segunda pasada se harán nuevamente comparaciones de datos consecutivos desde la posición 1 hasta la posición 5, ya que la posición 6 ya está en orden.
Las comparaciones e intercambios correspondientes a esta segunda pasada se presenta así:

1	3	2	6	4	8
1	2	3	4	5	6
1	3	2	6	4	8
1	2	3	6	4	8
1	2	3	**6**	**4**	8

Segunda pasada: cuatro comparaciones

Como resultado de esta pasada el vector queda así: los dos valores mayores ya están en su sitio.

1	2	3	4	5	6
1	2	3	4	**6**	**8**

Debemos continuar con las mismas pasadas, pero en cada pasada las posiciones finales que ya están ordenadas no entran en el ciclo, hasta lograr por completo el ordenamiento del arreglo. Continúa hasta lograr ordenar todo el vector.

Otro algoritmo ejemplo Ordenamiento por BURBUAJA

```
INICIO
NELEM = Número de elementos del arreglo
ARREG = Arreglo unidimensional a ordenar
IMPLICIT NONE
INTEGER NELEM
REAL ARREG(*)
INTEGER I,J
REAL AUX
IF (NELEM.LT.2) RETURN
      DO I=1,NELEM-1
                DO J=1,NELEM-I
                    IF (ARREG(J).GT.ARREG(J+1)) THEN
                              AUX = ARREG(J)
                              ARREG(J) = ARREG(J+1)
                              ARREG(J+1) = AUX
                     ENDIF
                ENDDO
          ENDDO
ENDIF
RETURN
```

Ejercicio:
En grupos de n estudiantes, se deberá
Implementar un algoritmo y codificarlo en cualquier lenguaje, para ordenar por el método de Burbuja. Los elementos de un arreglo de 20 posiciones, el cual contiene los nombres de igual número de productos de una tienda de abarrotes.
Luego deberán argumentase posiciones personales con respecto a cuál de las formas anteriores de ordenamiento es más eficiente, funcional, recursiva y fácil de implementar y porque.
Termina la actividad con la puesta en común de los argumentos más fuertes de cada grupo de trabajo.

2.3. Arreglos Multidimensionales o Matrices

1. Arreglos Bidimensionales

El arreglo bidimensional se puede considerar como un vector de vectores. Es, por consiguiente, un conjunto de elementos del mismo tipo, en el cual el orden de los componentes es significativo y en el que se necesita especificar dos subíndices para poder identificar cada elemento de la matriz.

En una matriz de 30 elementos, cada uno de ellos tiene el mismo nombre. Sin embargo, un subíndice no es suficiente para especificar un elemento de un arreglo bidimensional; por ejemplo, si el nombre del arreglo es M, no se puede indicar M [3], ya que no sabemos si es el tercer elemento de la primera fila o de la primera columna. Es por ello que los elementos de un arreglo bidimensional deben referenciarse con dos subíndices: el primero refiere la fila y el segundo refiere la columna. Así M [2,3] se refiere al elemento de la segunda fila, tercera columna.

Matriz **M**

Fila 1.					
Fila 2.			M[2,3]		
Fila 3.					
Fila 4.					M[4,5]
Fila 5.					
Fila 6.					
	Columna 1	Columna 2	Columna 3	Columna 4	Columna 5

2. Recorrer Las Posiciones Y Almacenar Datos En Una Matriz

Para ello es necesario utilizar dos ciclos, una que mueva las filas y otro que muevas las columnas, de igual forma podría colocarse invertido, es decir, primero se muevan las columnas y después las

filas, como ya sabemos los ciclos internos se agotan por cada variación del ciclo externo. Veamos el algoritmo para almacenar los datos ingresados a cada posición de la matriz anterior.

Variables: Matriz(m,n), i, j: enteras
M=6, n=5
inicio
para i desde 1 hasta m con_variacion + 1
para j desde 1 hasta n con_variacion + 1
Lea (matriz[i][j])
Fin(PARA)
Fin(PARA)
FIN

1. Se inicia definiendo las variables necesarias para el incremento de los ciclos, i,j
2. Se inicializan las variables con el tamaño correspondiente a la filas m=6 y las columnas n=5.
3. Inicia el algoritmo
4. El primer ciclo controlando por i hasta m, moverá el flujo por las filas.
5. El segundo ciclo controlado por j hasta n, moverá las columnas.
6. Dentro de los ciclos, se lee cada dato en la posición actual de i,j, quedándose i en la misma posición hasta que j sea igual a n luego se repetirá todo el recorrido de j por cualquier cambio de i.
7. Se cierra el ciclo interno y luego el externo y la matriz habrá almacenado cada dato ingresado. Supongamos que la matriz con los datos quedo así

Fila 1.	50	12	10	66	15
Fila 2.	23	44	20	55	15
Fila 3.	12	22	30	44	13
Fila 4.	34	32	40	33	19
Fila 5.	56	67	49	22	29
Fila 6.	76	78	54	11	38
	Columna 1	Columna 2	Columna 3	Columna 4	Columna 5

Ejercicio
Escriba un algoritmo para imprimir la matriz anterior por columnas, y luego codifíquelo en cualquier lenguaje para ver su ejecución y resultados.

C. Sumar Los Datos De Cada Columna De La Matriz

Para ello asumiremos la matriz que se llenó en el ejercicio anterior, en este caso podemos utilizar el mismo algoritmo con algunas variaciones sencillas. Veamos.

Variables: Matriz (m,n), i, j: enteras Totcolumna:real
M=6, n=5,
inicio
para i desde 1 hasta n con_variacion + 1
TotColumna=0
para j desde 1 hasta m con_variacion + 1
totcolumna=totcolumna+ (matriz[i,j])
Fin(PARA)
Imprima totcolumna
Fin(PARA)
FIN

Matrices rectangulares son aquellas en las que el número de filas es diferente del número de columnas, y matrices cuadradas son aquellas en las que el número de filas es igual al número de columnas.

3. Sumar La Diagonal Principal De Una Matriz Cuadrada

El siguiente algoritmo sumará aquellas posiciones de la matriz donde la fila sea igual a la columna.

Variables: i, s: enteras; matriz(N,N)
inicio
s = 0
Para I Desde 1 Hasta N Con_Variacion + 1
S = s + matriz[i][i]
Fin(PARA)
Imprima S
FIN

- Definimos nuestras variables de trabajo; **i**, variable con la cual recorremos los elementos de la diagonal principal, y **s**, la variable en al cual almacenamos la suma de dichos elementos.
- Iniciamos el proceso

- Inicializamos s en cero
- El ciclo para recorrerá la matriz avanzando por filas y columnas, una vez que van a la par.
- Desde uno hasta n
- Dentro del ciclo se suma el contenido donde la fila sea igual a la columna
- Se cierra el ciclo y se imprima la variable S.
- Terminamos el programa
- Supongamos que la matriz sumada fue la siguiente, los números sombreados son su diagonal principal y fueron los sumados.

								n
		1	2	3	4	5	6	7
	1	**3**	1	6	2	8	4	5
	2	9	**7**	7	4	1	2	4
	3	5	6	**2**	8	5	4	7
	4	3	5	5	**4**	8	9	3
	5	8	4	5	8	**8**	7	6
	6	6	3	5	9	8	**4**	2
n=	7	2	5	7	5	4	4	**1**

Ejercicio:
Proponga un algoritmo que permita calcular el promedio de los datos ubicados en la diagonal secundaria de la matriz anterior.

D. Ordenamiento De Los Datos De Una Matriz De M Filas Por N Columnas

Para este proceso utilizaremos un Vector auxiliar de tamaño MxN al cual llevaremos los datos de la matriz, leída por columnas, según queremos que nos quede ordenada la matriz. Luego ordenamos los datos en el vector y los volvemos a llevar a la matriz.

```
Paso 1: Copiar matriz en vector
NV:= MxN;
For i := 1 to M do
  For j := 1 to N do
    Begin
      NV := NV + 1;
    B[ NV ] := A[ i, j ]
  Enddo
```

```
Enddo
Paso 2: Ordenar el vector
For i := 1 to NV-1 do
    For j := i + 1 to NV do
        If B[ i ] > B[ j ] then
            Begin
                Aux := B[ i ];
                B[ i ] := B[ j ];
                B[ j ] := Aux;
            Enif;
    Endo
Enddo
Paso 3: Copiar vector en matriz
NV := 0;
  For i := 1 to M do
          For j := 1 to N do
            Begin
             NV := NV + 1;
                A[ i, j ] : = B[ NV ];
          End;
  Enddo
Enddo
```

Tomado de: Computación II , María Beatriz Serrano

4. Suma De Dos Matrices Llevando Los Resultados A Otra Matriz

Para sumar dos matrices A y B equivalente en filas y columnas, y llevar el resultado a otra matriz C, basta con iniciar el recorrido de las matrices con un par de ciclos, que accedan a cada fila y columna, a la par en el ciclo interno se va expresando la operación y se descarga el resultado en la matriz C. Veamos el algoritmo para ello.

La suma de dos matrices **A** y **B** consiste en crear una nueva matriz **C** en la cual cada elemento de **C** es la suma de los correspondientes elementos de las matrices **A** y **B.** por tanto, para poder sumar dos matrices, estas tienen que tener las mismas dimensiones. Simbólicamente la suma de dos matrices es expresa así: es realmente sencillo.

```
Variables: i, j: enteras A(M,N), B(M,N), C(M,N)
   Inicio
         para i desde 1 hasta n con_variacion + 1
           para j desde 1 hasta m con-variacion + 1
                c[i][j] = a[i][j] + b[i][j]
          Fin(para)
       fin (para)
   fin
```

- Inicia la definición de las variables y matrices con sus respectivos tamaños.
- Un ciclo externo que accederá a las filas de la matriz, desde 1 hasta n
- Un ciclo interno que accederá a las columnas de la matriz, desde 1 hasta m.
- Dentro del ciclo más interno, se asigna a cada posición de la matriz C los resultados de los datos correspondientes en las matrices A y B.
- Cerramos los ciclos y el programa.

De igual forma podrías calcular cualquier operación con los datos equivalente de dos matrices y almacenarlas en otra diferente.

1. **Transpuesta de una Matriz**

La transpuesta de una matriz es aquella matriz resultante de tomar la matriz **A** con **m** filas y **n** columnas, y llevar sus datos a una matriz **B** donde su número de filas serán las columnas de **A** y su número de columnas serán las filas d **B**

Cada elemento de la fila **i** columna **j** de la matriz **A** queda ubicado en la fila **j** columna **i** de la matriz

B. El dato que en la matriz **A** se hallaba en la fila 2 columna 5 queda en la fila 5 columna 2, así sucesivamente. Observe la ilustración.

A=

	1	2	3	4	5	6	n 7
1	3	1	6	2	8	4	9
2	7	7	2	5	1	5	4
3	4	1	3	8	1	2	8
4	6	3	2	1	5	5	3
m5	4	1	2	2	1	3	7

B=

	1	2	3	4	n 5
1	3	7	4	6	4
2	1	7	1	3	1
3	6	2	3	2	2
4	2	5	8	1	2
5	8	7	1	5	1
6	4	5	2	5	3
m7	9	4	8	3	7

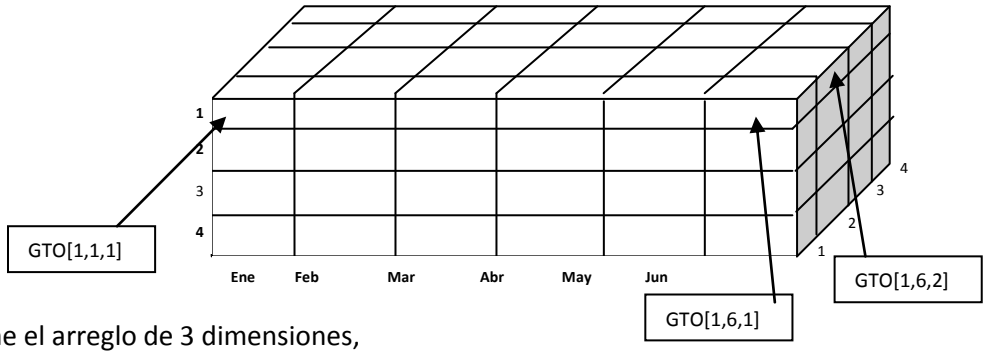
Ejercicio:
Implemente un algoritmo que a partir de la matiz A genere e imprima la matriz B.

Información complementaria
Puede ayudarte observar algunos procesos con arreglos de multidimensionales, en su operación de cálculo de su transpuesta y diagonal principal y segundaria. http://www.youtube.com/watch?v=qdY4vK1V0Nk, accedida en septiembre 27 de 2011

2. Arreglos Tridimensionales

Un arreglo puede ser definido de tres dimensiones, cuatro y hasta de N dimensiones. Los conceptos de rango de subíndices y número de elementos se pueden ampliar directamente desde arreglo de una dimensión hasta de orden más alto. En general, un arreglo de n-dimensiones requiere que los valores de los n subíndices puedan ser especificados a fin de identificar un elemento individual del arreglo.

El arreglo A de N dimensiones puede representarse así:
A[1..3, 1..2, 1..3] de tres dimensiones 3,2,3, su el límite inferior es diferente de 1, entonces los índices se calculan así: A[LS_1-LS_1+1, LS_2-LS_2+1, LS_3-LS_3+1]

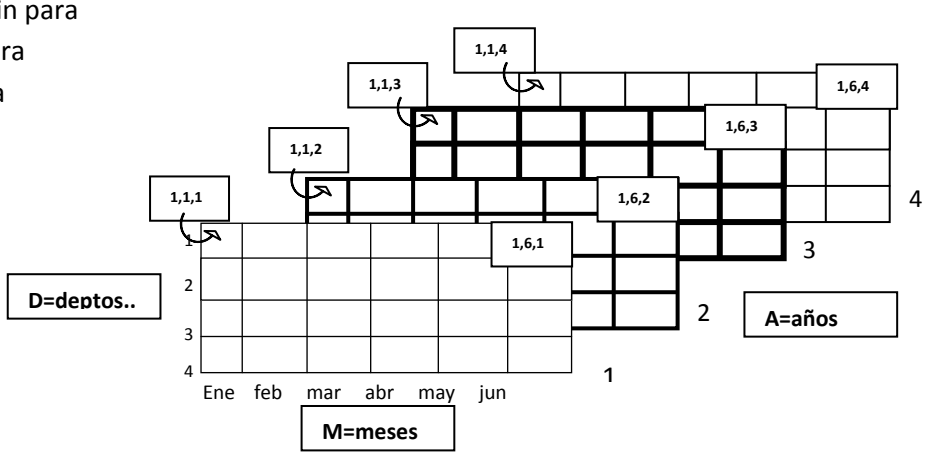


Se tiene el arreglo de 3 dimensiones,
Identificando los índices así:

GTO[4,6,4]
Donde se tienen gastos de
4 departamentos, Por 6 meses en
4 años

Tendríamos almacenados los gastos correspondientes a los meses de enero a junio, por cada departamento durante 4 años.
El algoritmo para llenar los datos de la matriz sería el siguiente:

```
Inicio
Para A=1 hasta 4 incrementando  1
  Para D=1 hasta 4 incrementando 1
        Para M=1 incrementando 1
          Lea Gto[D,M,A]
        Fin para
  Fin para
Fin para
```



3. Ilustración más acorde con la realidad de un arreglo en tres dimensiones

Observamos entonces que: Los departamento serán controlados por la letra o índice **D**, los meses se controlan con la letra o índice **M** y los años con la letra **A**. las direcciones en el conjunto de matrices se acceden primero con el número de la fila, luego el número de la columna y al final se referencia el número de la matriz en la cual esta´. Por ello la referencia queda Gto[1,1,1] hace referencia a la fila 1, columna 1 en la matriz 1. Gto[1,6,3] se refiere a la posición fila 1, columna 6 matriz 3.

Por tanto para el acceso a las posiciones de un arreglo tridimensional deberá utilizar 3 ciclos, uno para acceder a las diferentes matrices, otro para moverse en cada matriz por las filas y otro para moverse por las columnas.

Ejercicio:
En un trabajo colaborativo, el grupo de estudiantes deberá proponer un algoritmo y codificarlo en un lenguaje comercial o simulador, que permita almacenar en una matriz tridimensional, la información correspondiente a 10 buses que deberán registrar su número de pasajeros diarios por cada viaje, durante los siete días de la semana. Además deberá generar los siguientes informes:
Número total de viajes y pasajeros por bus y por semana.
Promedio de pasajeros por viaje y por día.
Bus y día de mayor y menor número de pasajeros.

Al finalizar la actividad se deberá valorar la interacción y participación que se observó en cada uno de los integrantes, desde sus aportes, potencialidades y limitantes evidenciadas.

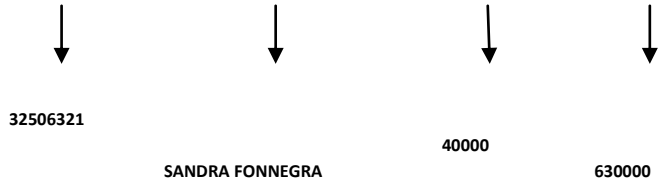
2.4. Registros y Arreglos de Registros

1. Registros

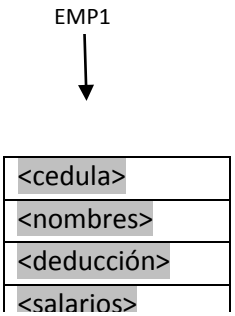
Un registro es una estructura de datos compuesta. Se puede decir que un registro es un conjunto de campos variables relacionados que, en general, puede pertenecer a tipos de datos diferentes, llamados componentes del tipo registro, donde todas las componentes pueden manipularse bajo un solo nombre de variable. Por ejemplo, si se tiene un registro de datos compuestos por los campos: cedula, nombres, deducción y salario, podemos representarlo de la siguiente forma: Si la información la tratamos de forma individual necesitaríamos los nombres de variables Cc, Nom, Deduc y Sal, lo que haría generar campos o posiciones diferentes en memoria con estos nombres.

Si la información del empleado se fuera a tratar en forma individual y los nombres de variables seleccionados para cada campo fuera: CC, NOM, DEDUC y SAL, al ejecutar la instrucción:

LEA: CC, NOM, DEDUC, SAL
Ocurrirá en memoria lo siguiente:



La información de la empleada está dispersa en la memoria. La idea de usar registro es agrupar toda la información en una misma área de memoria bajo un solo nombre. Si se toma la Determinación de que EMP1 es una variable de *tipo registro,* o sea, almacenar en el área asignada toda la información de un empleado, gráficamente se puede mirar de la siguiente manera.

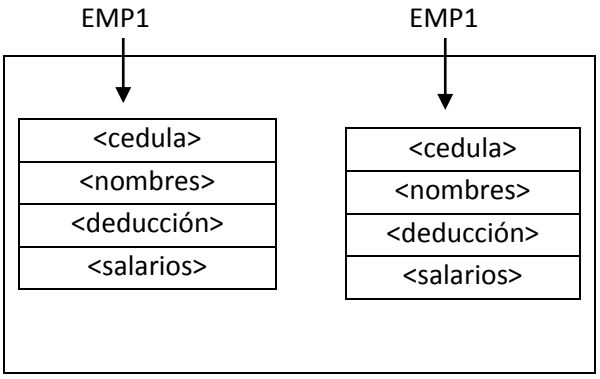


Para que EMP1 sea tomada como una variable tipo registro, el programador debe definirla como tal, según el lenguaje utilizado; es decir, el tipo de dato registro no está predefinido, ejemplo, en C++ se definirá así:

```
Struct empleado
        {
            Long cc;
             Char nom[36]
              Doublé deduc;
             Doublé sal;
        };
```

Lo anterior quiere decir que el programador ha definido un tipo de dato registro llamando empleado y que a cualquier variable que se le asigne el tipo de dato empleado, se puede almacenar los valores de las componentes: cc, nom deduc y sal.
Empleado emp1, emp2;

En la anterior instrucción se define a emp1 y emp2 como variables tipo registro, por lo tanto, pueden almacenar todos los valores de los campos que conforman al tipo de dato empleado, y su representación interna seria:



Las variables tipo registro emp1 y emp2 son variables compuestas; para referenciar a cada una de sus componentes hay que clasificarlas, o sea, decir en forma explícita a cuál de las componentes del registro se quiere uno referir. La cualificación se hace así

Variables_tipo_regristro.componente.

Para referirse a cada uno de los datos que componen cada registro, se procede así:

Emp1.cc	Emp2.cc
Emp1.nom	Emp2.nom
Emp1.deduc	Emp2.deduc
Emp1.sal	Emp2.sal

Manejando cada referencia como si fuese una variable independiente, adscrita al nombre del registro, en memoria pueden definirse diversos registros y cada uno con diferente cantidad y tipo de datos.

Para definir una variable de tipo registro o un registro en memoria se procede así.

Tipo registro: **Cliente**
    Cadena: Nombre
    Cadena: Teléfono
    Real: Saldo
Fin Tipo

Este registro determina un registro de nombre cliente, con los campos Nombre de tipo cadena, teléfono de tipo cadena y saldo de tipo real.

De igual forma podremos tener:

Tipo registro: Venta	Tipo registro: Producto
Real:Descuento	Cadena: descripción
Real:PcentajeUtilidad	Real:precioCosto
Real:PrecioVenta	Entero: cantidad
Entero:cantidad	Real:Vlrcompra
Real:VlrVenta	
Fin tipo	Fin tipo

Operaciones:
Producto.vlrcompra=producto.preciocosto*producto.cantidad
Venta.precioventa=producto.preciocosto+producto.preciocosto*venta.pcentajeutilidad
Venta.vlrventa=(venta.precioventa-venta.descuento)*venta.cantidad

Note que se pueden realizar operaciones con los campos de cada registro con solo referirlos mediante el nombre del registro punto y el campo específico.

2. Arreglo de Registro

Es una estructura de datos de gran utilidad e importancia en la programación de aplicaciones, ya que muchas veces es deseable, desde el punto de vista de la lógica del programa en particular, mantener disponible en memoria principal una serie de registro con información del mismo tipo para su proceso; dichos registros no estarán dispersos; harán parte de un arreglo, facilitando su manejo.

Un arreglo de registros es muy parecido a un archivo de datos, se diferencian en que los arreglos de registros residen en memoria principal y los archivos en memoria auxiliar, por lo tanto, estos últimos tiene existencia permanente.

Otra importancia que tienen los arreglos de registros es que cada uno de sus elementos está formado por varios campos pertenecientes a diferentes tipos de datos, a diferencia de los otros tipos de arreglos sus elementos son de un solo tipo.

Un arreglo de registros es aquel donde cada casilla o posición del arreglo puede ser un Registro, es decir puede contener diferentes tipos de datos. Para definirlo como tal podemos proceder así:

Definimos el registro
Tipo registro: Cliente

- Cadena: Nombre
- Cadena: Teléfono
- Real: Saldo

Fin Tipo
Luego definimos el arreglo con su nombre, límite inferior, límite superior o tamaño del mismo y lo establecemos de tipo registro.

Arreglo [0...10] de Cliente
De esta forma ya tenemos definido un arreglo de 10 posiciones y en cada una de las posiciones se define un registro con cada uno de los campos definidos para él.

Para referirnos a un campo de registro en una posición específica lo hacemos de esta forma:

Arreglo [1].nombre
Arreglo [1].teléfono
Arreglo [1].saldo

Y así sucesivamente en cada una de las posiciones del Arreglo, para ello como ya sabemos por procedimientos con arreglos, debemos colocar un ciclo que nos permita acceder a cada posición del arreglo y por ende a cada uno de sus campos.

Para los procesos de ordenamiento y búsqueda se procede de igual forma como se hizo con los arreglos unidimensionales una vez que al hacer referencia solo a la posición del arreglo ya se está involucrando todo el registro de dicha posición.

Ejercicio:
Se puede coordinar un trabajo colaborativo, que plantee un programa principal, en algoritmo o código de lenguaje, donde cada persona aporte un subprograma que realice una de las siguientes tareas y:
Plantee una solución al siguiente escenario: se requiere almacenar en un arreglo de registros, los datos: documento, nombre, sexo, fecha nacimiento, edad, teléfono, y dirección de sus profesores de carrera, la edad se debe calcular teniendo como base la fecha de nacimiento y la fecha actual, y elaborar informes que muestren Promedio de edad de los profesores, los datos del profesor mayor y del menor, y que ordene el arreglo alfabéticamente según el nombre.
La actividad termina con la venta del programa desarrollado al resto del grupo, argumentando sus fortalezas funcionales, de rendimiento y evidenciando las bondades del trabajo colaborativo

Actividades de Afianzamiento

1. Implementa un algoritmo para resolver un problema cotidiano, donde sea pertinente y adecuado el uso de 3 vectores con manejo paralelo. Piense que informes podrían pedirse a cerca de dicha información almacenada, aplique luego un método de ordenamiento a los datos de cada vector sin perder la sincronía de sus datos.

2. Para administrar los procesos de venta de la compañía los Lirio.S.A. Es necesario almacenar la siguiente información: código del Vendedor, y los nombres de cada uno de los meses de Enero a junio. Usted deberá plantear la forma de almacenar la información correspondiente a 15 vendedores durante estos 6 meses del año, y presentar informes, referentes a: Total, promedio, máxima y Mínima venta hecha por mes, por vendedor y en todo el semestre. Debe implementarse con arreglos o vectores bidimensionales.

3. El metro de Medellín necesita llevar un conteo del número de pasajeros que mueve cada uno de sus 20 vagones a la semana, para ello se tienen la siguiente información: el metro inicia labores a las 5 de la mañana hasta las 11 de la noche, cada vagón toma 60 minutos para su recorrido. Deberá almacenarse en una arreglo bidimensional por cada vagón el número de pasajeros que transporta en cada viaje, el vagón controla sus pasajeros por viaje registrando cada ingreso. Se debe vaciar luego a otro arreglo bidimensional el total de pasajeros que movió cada vagón por cada uno de los días de la semana, que permita elaborar un informe sobre: promedio de pasajeros por viaje, promedio de pasajeros por vagón, día y vagón con mayor y menor número de pasajeros. Total pasajeros movilizados por el metro en la semana.

4. Deberás investigar sobre una situación problémica

5. cotidiana de la ciudad en cualquier ámbito comercial o industrial que requiera plantear una solución informática utilizando arreglos de tres dimensiones, aplicando las operaciones básicas con dicha estructura de almacenamiento para generar informes funcionales y pertinentes a la necesidad a suplir.

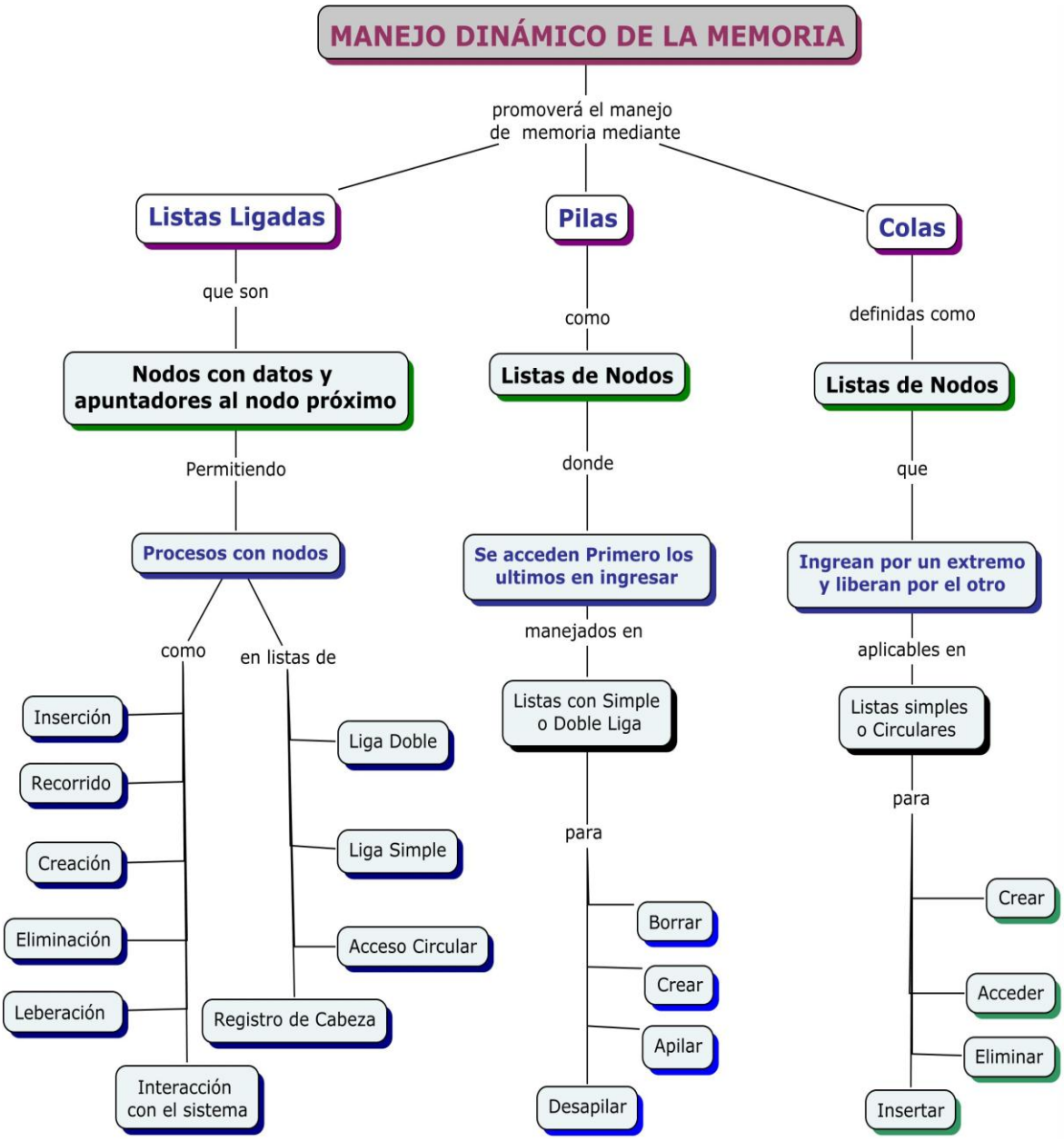
3. MANEJO DINÁMICO DE LA MEMORIA

Las estructuras básicas o arreglos tienen una limitación, no pueden cambiar de tamaño durante la ejecución. Los arreglos están compuestos por un determinado número de elementos, número que se decide antes de empezar a utilizarlos en la fase de diseño, antes de que el programa se ejecute.

Pus bien en muchas ocasiones se necesitan estructuras que puedan cambiar de tamaño durante la ejecución del programa. Por supuesto, podemos hacer arreglos dinámicos, pero una vez creados, su tamaño también será fijo, y para hacer que crezcan o disminuyan, debemos reconstruirlas desde el principio.

Las estructuras dinámicas nos permiten crear estructuras de datos que se adapten a las necesidades reales a las que suelen enfrentarse nuestros programas. Además nos permitirán crear estructuras de datos muy flexibles, ya sea en cuanto al orden, la estructura interna o las relaciones entre los elementos que las componen.

3.1. Mapa Conceptual



OBJETIVO GENERAL

Promover en el estudiante proposiciones y formas de manejo dinámico de memoria, mediante el uso de listas, pilas y colas, frene a situaciones reales que demanden su aplicación.

OBJETIVOS ESPECÍFICOS

- Apropiar al estudiante del manejo de las estructuras de almacenamiento en listas
- Inducir al estudiante en el manejo óptimo de la memoria implementando estructuras de almacenamiento con Pilas
- Llevar el estudiante a plantear soluciones efectivas mediante la propuesta de estructuras de almacenamiento ante el uso de las Colas.

Prueba Inicial

Con esta prueba se pretende que usted antes de iniciar la temática correspondiente a ésta unidad, mida sus conocimientos previos, para tener un punto de partida en su asimilación y poder confrontarlo al final de la unidad, determinando así su avance y progreso con respecto a la asignatura.

1. ¿Qué entiende usted por manejo dinámico de la memoria?
2. ¿Qué relación encuentra usted entre un arreglo de registros y una lista ligada?
3. ¿En qué aspectos considera usted, puede favorecer a un sistema el hecho de utilizar listas ligadas, colas o pilas en lugar de usar únicamente arreglos de registros?
4. ¿Es usted capaz de enunciar las diferencias funcionales entre las pilas, las colas y las listas?
5. ¿Por qué se dice que las listas, pilas y colas no son estructuras de almacenamiento propiamente definidas?
6. ¿Qué elementos de un algoritmo son comunes en el manejo de cualquier estructura de almacenamiento en memoria? Porque?

3.2. Estructuras de Listas

Una lista es una colección de elementos llamados generalmente nodos, El orden entre los nodos se establece por medio de punteros, es decir, direcciones o referencia a otros nodos. Los cuales constan esencialmente de dos partes. Una que contiene la información o datos de cualquier tipo y un capo liga de tipo puntero, que se utiliza para establecer el enlace con otra lista, si el campo es el último de la lista su campo liga sería igual a Nil(vacio). De esta forma no es necesario almacenar los nodos de forma continua

El primer nodo de la lista recibe el apuntador de la variable P, de tipo puntero que almacena la dirección del primer nodo. El último nodo con su apuntador Nil indica que ya no existen más nodos en la lista

En este nodo que llamaremos DATO, para referirnos al dato diremos DATO(X) o sea 3.1416 y si indicamos LIGA(x) | 3.1416 | 7 | hacemos referencia a 7.

3. Operaciones con las Listas

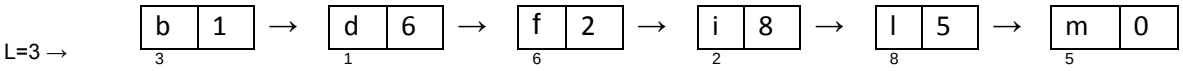
a. Creación de la Lista

Se define las variables de tipo apuntador P y Q, P para guardar la dirección de prime nodo.

Se crea el primer nodo de la lista CREA (P)
Lee el primer dato para el nodo creado Lea P^.información
Asigna la liga del primer nodo Hacer P^.LIGA= Nil
Se inicia el ciclo de repetición Repetir

Dentro del ciclo las instrucciones para crear otro nodo CREA (Q)
De igual forma para asignar el dato al nodo Lea Q^.información
Como ya hay dos nodos deben enlazarse Hacer Q^.LIGA=P; P=Q
Hasta que ya no haya más información.

b. Recorrer La Lista Ligada



Se define una variable auxiliar **p**.

- **p** le asignamos el valor de **L**. p queda valiendo **3**.
- Para imprimir el contenido del campo de **dato** del registro **p** escribimos **write(dato(p))** y escribe el dato **'b**'.
- Para avanzar sobre la lista, es decir, trasladamos al siguiente registro escribimos: **p = liga(p),** es decir, a la variable **p** le asigna lo que hay en el campo de liga del registro **p**. **p** queda valiendo **1.**
- Para imprimir el contenido del campo de **dato** del registro **p** escribimos **write(dato(p))** y escribe el dato **'d**'

- Variable **p** le asigna lo que hay en el campo de liga del registro **p**. **p** queda valiendo **6.**
- Para imprimir el contenido del campo de **dato** del registro **p** escribimos **write(dato(p))** y escribe el dato **'f'**
- Variable **p** le asigna lo que hay en el campo de liga del registro **p**. **p** queda valiendo 2**.**
- Imprimimos el dato del registro p y continuamos repitiendo el proceso hasta que p sea cero.

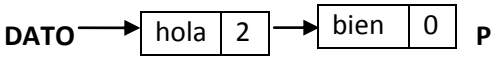
Un algoritmo que ejecuta esta tarea es:

```
Sub_programa recorre_lista(L)
        p = L
         while p <> 0 do
                write (dato(p))
                p = liga(p)
        End(while)
fin(recorre_lista)
```

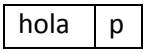
C. Inserción Al Inicio De La Lista

El nuevo nodo se coloca al inicio de la lista, para ello procedemos así:

Definimos Q de tipo puntero.
CREAT (Q) creamos el nodo Q
Hacer Q^.información =DATO o sea asignamos el dato a la parte de información de Q. La parte de liga de Q la hacemos igual P Q^.LIGA=P, que tiene la dirección inicial del primer nodo, y P la hacemos igual a Q , P=Q

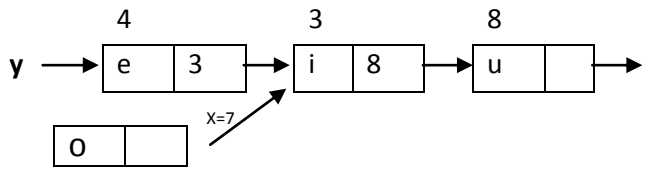


Nodo **Q** a insertar



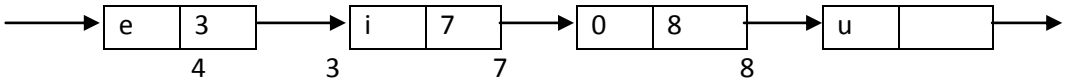
Ejercicio
Según lo anterior deduce cual sería el algoritmo para insertar el nodo al final de la lista.

Para insertar un registro en una lista ligada se debemos conocer en cual registro haremos la inserción. Llamemos **y** este registro, y llamemos **x** el registro a insertar.



Insertar el registro **x (x = 7),** a continuación del registro **y (y = 3)** implica que cuando llegamos al registro **y** debemos trasladarnos hacia el registro **x,** o sea que el campo de liga del registro **y** debe quedar valiendo 7, y cuando estemos en el registro **x** debemos trasladarnos hacia el registro 8, es decir, que el campo de liga del registro **x** debe quedar valiendo 8. Para lograrlo debemos proceder así:

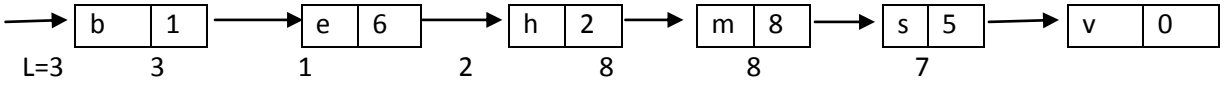
liga(x)=liga(y)
liga(y) = x
y la lista queda:



Ahora que pasa si tenemos una lista ligada, se leyó un dato y hay que insertarlo en la lista.

- Buscar donde insertar el dato leído, es decir, determinar **y.**
- Conseguir un registro **x,** guardar el dato leído en **x** y luego insertarlo a continuación del registro **y.**

Ocupemos del primer paso. Consideremos la siguiente lista:



Sea **d**= 'j' el dato leído.

Para buscar donde insertarlo debemos recorrer la lista comparando el dato de cada registro con el dato leído hasta encontrar un registro cuyo dato sea mayor que **d,** el dato leído.

El nuevo registro debe insertarse antes de ese dato, o sea, a continuación del registro anterior aquel que tiene un dato mayor. Esto significa que si utilizamos una variable **p** para hacer el recorrido e ir efectuando las comparaciones, debemos utilizar otra variable que permanentemente apunte hacia el registro anterior a **p.** llamemos esa variable **y**.

Inicialmente a **p** le asignamos el valor de **L**, y como este es el primer registro y no tiene anterior, el valor inicial de **y** es cero.

Para efectuar esta tarea utilizaremos la siguiente función:

```
Function buscar_donde_insertar)L, d)
        p = L
        y = O
        while p <> 0 and dato(p) < d do

                y = p
                p = liga(p)
        end(while)
return(y)
fin(buscar_donde_insetar)
```

L y **d** son parámetros de entrada, **y** es el dato de retorno.

Es bueno hacer notar en este algoritmo que si el dato a insertar debe quedar de primero en la lista, el valor retornado en **y** es cero; si el dato a insertar debe quedar de ultimo en la lista, entonces **y** quedara apuntando hacia el último registro de la lista ligada.

A continuación presentamos el algoritmo de inserción. Observe que dicho algoritmo no requiere mover datos para poder insertar un dato en la posición apropiada. Esto hace que el orden de magnitud de dicho algoritmo sea constante, es decir, **O(1).**

```
Sub_programa inseratr(L, y, d)
        conseguir_regristro(x)
        dato(x) = d
         if y = 0 then
                liga(x) = L
                 L = x
         else
                 liga(x) = liga(y)
                  liga(y) = x
         end(if)
fin(insertar)
```

Y y **d** son parámetros de entrada, **L** es parámetro de entrada y salida.

En este algoritmo utilizamos una instrucción **conseguir_registro(x)**, la cual, invoca un sub_programa que interactúa con el sistema operativo solicitándole un registro, cuya dirección la retorna en la variable **x.**

Información complementaria
Información que puede darte otras ideas al respecto o complementar la abordado sobre listas. http://www.youtube.com/watch?v=ohb9bjPX7Vw&feature=relmfu, Acceso Oct. 2011

Ejercicio:
Poniendo a prueba su capacidad de indagación, consulta y creatividad proponga la lógica y el algoritmo necesario para borrar un registro leído en una lista ligada.

D. Interacción con El Sistema Operativo

Conseguir registro(x) y **liberar registro(x)**. Parte de las funciones del sistema operativo de un computador es administrar la memoria. Lo que incluye conocer los registros libres y ocupados, para que al invocarse en forma dinámica algún procedimiento su interacción con el sistema operativo sea correcta y eficiente. Dicha interacción consiste en que el sistema operativo le asignara memoria al programa cuando este lo solicite y recibirá memoria cuando el programa lo libere.

Conseguir registro(x): el sistema operativo asigna memoria al programa que hizo la solicitud. La dirección del registro o bloque asignado retorna en el parámetro **x.** para poder efectuar una asignación correcta, el sistema operativo debe conocer cuales registros de memoria están libres y asignar uno de ellos.

Liberar registro(x): da la orden al sistema operativo que disponga del registro o bloque **x** y lo marque como libre. Una vez que el sistema operativo maneja una lista ligada con los registros disponibles.

Si un programa invoca conseguir registro(x) entonces el sistema operativo asignara el primer registro de la lista ligada en la cual maneja los registros libres, al programa que hizo la solicitud. Esta lista ligada consta de todos los registros disponibles, y una vez asignado al programa, lo borrara de la lista de disponibles, es decir, lo desconecta de dicha lista**.**

Si se invoca **liberar registro(x),** el sistema operativo inserta un registro al principio de la lista de registros disponibles.

Veamos y analicemos los algoritmos para construir y liberar un registro

```
Sub_programa conseguir_registro(x)
        if disp = 0 then
                write("no hay registros disponibles")
                 stop
        end(if)
        x = disp.
        disp = liga(disp)
fin8conseguir_registro)
Sub_programa liberar_registro(x)
       liga(x) = disp.
        disp. = x
fin(liberar_registro)
```

De aquí en adelante, por simplicidad, el subprograma **conseguir registro(x)** lo seguiremos invocando **new(x)** y el subprograma **liberar registro(x)** lo invocaremos **free(x)**

Sean **A** y **B** los apuntadores a dos listas ligadas, cada una de las cuales tiene datos únicos ordenados ascendentemente. El objetivo es construir una nueva lista ligada intercalando los datos de las listas **A** Y **B** sin quedar datos repetidos en la lista resultado.

Para efectuar dicha tarea debemos recorrer las listas **A** y **B** simultáneamente, comparando los datos de ellas. Cuando el dato de un registro en la lista resultado con los datos correspondientes al registro de la lista **A** y avanzados sobre la lista **A;** si el dato menor corresponde al registro de la lista **B** entonces añadimos un registro a la lista resultado con los datos correspondientes al registro de la lista **B** y avanzamos sobre la lista **B;** en caso de que los datos sean iguales añadimos un registro a la lista resultado con ese dato y avanzamos sobre ambas listas.

Para recorrer una lista ligada se requiere de una variable auxiliar. Como aquí se requiere recorrer dos listas simultáneamente necesitaremos dos variables auxiliares. Llamémoslas **p** y **q.**

Ejemplo:
El algoritmo tiene tres parámetros. Los dos primeros son parámetros por valor y son los apuntadores a las listas que se desean intercalar, el tercer parámetro es por referencia y en el cual se retornara el apuntador hacia el primer registro de la lista resultado.

```
Sub_programa intercala(A, B, C)
    1. New(x)
    2. L =x
    3. Ultimo = x
```

```
4.  P = A
5.  q = B
6.  while p <> 0 and q <> 0 do
7.  casos
1.  :dato(p) < dato(q):
2.  Añadir_registro(dato(p), ultimo)
3.  P = liga(p)
4.  :dato(p) < dato(q):
5.  Añadir_registro(dato(q), ultimo)
6.  q = liga(q)
7.  :dato(p) = dato(q)
8.  Añadir_ergistro(dato(p), ultimo)
9.  P = liga(p)
10. Q = liga(q)
11. Fin(casos)
12. End(while)
13. Whiel p <> 0 do
14. Añadir_registro(dato(p), ultimo)
15. = liga(p)
16. End(while)
17. While q <> 0 do
18. Añadir_registro(dato(q9,ultimo)
19. Q = liga(q)
20. End(while)
21. Liga(ultimo) = 0
22. X = L
23. L = liga(L)
24. Free(x)

 Fin(intercala)
Sub_programa añadir_registro(d, u)
        new(x)
        dato(x) = d
        liga(u) = x
        u = x
fin(añadir_registro)
```

En dicho sub_programa utilizamos una variable **último** ya que la forma de construir la lista resultado es añadiendo registros siempre el final de la lista. Utilizamos además un sub_programa

llamado **añadir_regristro,** el cual tiene como parámetros un campo de dato y el campo llamado **último.** El campo de dato es parámetro por valor y el campo **último** es parámetro por referencia.

Como una lista ligada se puede terminar de recorrer mientras que en la otra lista aún faltan registros, los ciclos de las instrucciones 20 y 24 controlan esas situaciones. Al final del algoritmo, instrucciones 28 y 30, borramos el primer registro de la lista ligada resultado, ya que este es un registro auxiliar que se utilizó para construir la lista más fácil y eficientemente. Cuando expliquemos el sub_programa **añadir_registro** trataremos este caso.

El sub_programa **añadir _registro** consigue un registro, le asigna el dato recibido, lo conecta con el ultimo y dice que ese es el nuevo último.

Ahora, para que utilizamos un registro auxiliar en la construcción de la lista resultado.

Si no utilizamos registro auxiliar, las tres primeras instrucciones del sub_programa **intercala** habría que reemplazarlas por una sola instrucción que fuera **último = 0,** y la primera vez que se invoque el sub_programa **añadir_registro** el parámetro **u** entraría valiendo 0, lo cual implica que el sub_programa **añadir_termino** quedaría así:

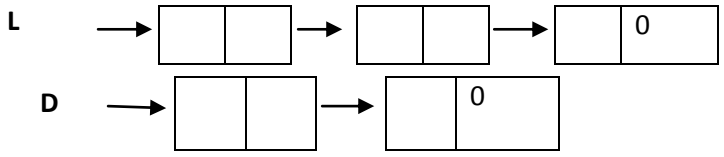
```
Sub_programa añadir_registro(L, d, u)

        new(x)
        dato(x) = d
       if u = 0 then
              L = x
       else
              liga(u) = x
       end(if)
        u = x
fin(añadir_registro)
```

Y la pregunta **if u = 0** solo verdadera la primera vez que se llame el sub_programa, todas las demás veces el resultado de la pregunta es falso, por tanto, hacer esta pregunta es ineficiente, además que hay que codificar más instrucciones. Como se puede observar, con la utilización de la variable auxiliar, al principio de la lista obviamos la pregunta en **añadir registro.**

E. Proceso Para Liberar Listas Ligadas

Tenemos a **L** el apuntador hacia el primer registro de la lista que se desea liberar; **disp.** El apuntador hacia el primer registro de la lista ligada de registros disponibles.



El siguiente algoritmo retorna todos los registros de la lista L a la lista de disponibles, uno por uno.

```
Sub_programa liberar_lista(L)
    while L <> 0 do
        x = L
        L = liga(L)
        free(x)
    end(while)
fin(liberar_lista)
```

Así, que si la lista tiene **n** registros el algoritmo será de orden **O(n).** Por eficiencia liberaremos todos los registros de una sola vez.

Si dispongo de la lista **L** y quiero llevar sus regiones a la lista de disponibles **(disp),** sin tener que hacerlo uno por uno, debe recorrerse la lista **l** para encontrar su último registro y ponerlo apuntar hacia el primer registro de la lista **disp.** Luego la nueva lista **disp.** Sera **L** y tendrá la misma dirección de memoria que **L.**

```
Sub_programa liberar_lista(L)
        p = L
        while liga (p) <> 0 do
                p = liga(p)
        end(while)
        liga(p) = disp.
        disp. = L
         L = 0
fin(liberar_lista)
```

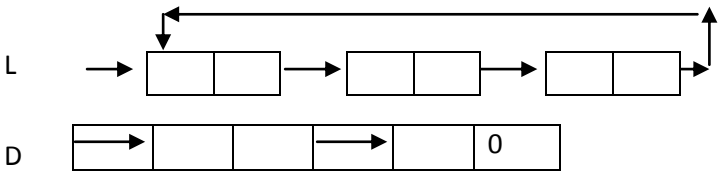
Este algoritmo devuelve todos los registros de la lista L a la lista de disponibles. El orden de magnitud es O(n) ya que de todas formas tenemos que recorrer la lista ligada para identificar el último registro.

Aunque los dos algoritmos tienen el mismo orden de magnitud, es más eficiente el segundo ya que dentro del ciclo solo hay una instrucción, y se omite el llamado al sub_programa **liberar registro(x).**

Sin embargo, para procesos que se presentan con mucha frecuencia, lo ideal es tener un algoritmo con orden de magnitud **O(1).** Para obtener esto, basta con hacer una pequeña modificación a la representación de la lista ligada: hacerla circular, o sea que el campo de liga del último registro ya no será cero sino que apuntara hacia el primer registro de la lista ligada.

El sub_programa de liberar una lista en representación de lista ligada circular tendrá orden de magnitud **O(1).**

El proceso consiste en encadenar **disp.** (Lista de disponibles) al primer registro de la lista **L** y asignarle a **disp.** El valor del segundo registro d ela lista **L.**



Para lograr dicho objetivo se guarda en una variable **x** el segundo registro de la lista **L.** la variable **disp.** Apuntara hacia este registro y el campo liga del registro **L** apuntara hacia el registro que inicialmente era **disp.**

```
Sub_programa liberar_lista_circular(L)
        x = liga(L)
        liga(L) = disq
        disp = x
         L = 0
         x = 0
fin(liberar_lista_circular)
```

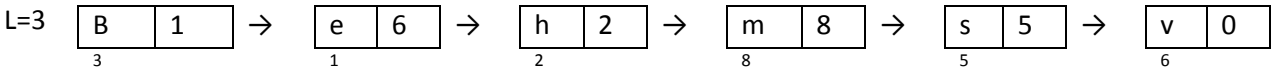
Información complementaria

En el siguiente link, podrás encontrar más información de ayuda para tu aprendizaje y aplicación de las listas, pilas y colas.

http://www.calcifer.org/documentos/librognome/glib-lists-queues.html

Ejercicio:
Según los conceptos analizados de interacción con el sistema operativo, de manera gráfica utilizando los elementos que usted consideré más expresivos, traduzca el proceso de interacción del sistema operativo con las listas que se dan en su memoria y los subprogramas que las ejecutan. Para confrontar con sus compañeros su percepción conceptual al respecto.

F. **Listas Simplemente Ligadas**



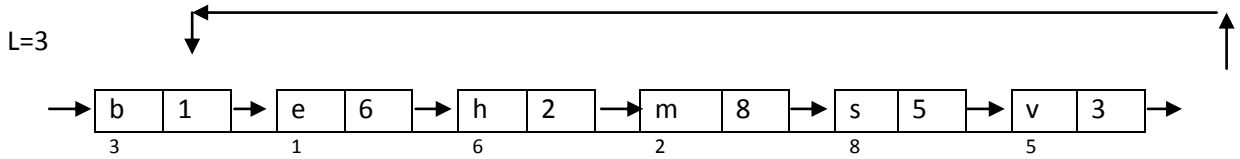
Recorda el algoritmo de recorrido.

```
Sub_programa recorrer_lista(L)
        p = L
        while p <> 0 do
                write(dato(p))
                p = liga(p)
    end(while)
fin(recorrer_lista)
```

El algoritmo recorre e imprime el contenido de cada registro de la lista. Aquí se controla la situación de la lista vacía, **L = 0,** con la misma instrucción con que se controla la terminación de recorrido de la lista: **while p <> 0.** Es decir, con una sola instrucción estamos controlando dos situaciones. Resumiendo, cuando tenemos una lista simplemente ligada, las características de lista vacía y recorrido son:

Lista vacia**: L = 0**
primer registro: p = L
Ultimo registro: liga(p) = 0
Termiancion de recorrido: p = 0

Lista simplemente ligada circular



Para recorrer la lista no podemos elaborar un algoritmo como en el caso de la lista simplemente ligada porque cuando tenemos la lista circular ningún registro tendrá campo de liga cero (0), por tanto la situación de terminación de recorrido no se podrá controlar con la instrucción **while p <= 0.**

Cuando tenemos la lista circular el recorrido se termina cuando la variable auxiliar para recorrido, es decir **p,** vuelva a ser igual a **L.** plantear el ciclo con la instrucción **while p <> L** no sería correcto porque la primera instrucción es signarle a la variable **p** el

Contenido de la variable **L,** y entonces al preguntar si **p** es diferente de **L** daría como resultado falso y nunca encontraría el ciclo.

Por consiguiente debemos plantear un ciclo que permita ejecutar primero las instrucciones del ciclo y después preguntar por la condición de si **p** es diferente de **L.** para ello disponemos de la instrucción del ciclo **do – while**. El algoritmo quedaría así:

```
Sub_programa recorre_lista_circular (L)
      p = L
      do
          write(dato(p))
         p = liga(p)
     while p <> L
fin(recorre_lista_circular)
```

En caso de que el parámetro **L** entre valiendo 0, la primera vez **p** queda valiendo 0, entra al ciclo y cuando vaya a ejecutar **write(dato(p))** nuestro programa cancela.

Para que el programa NO cancele debemos añadir instrucciones, para preguntar si **p <> 0**, antes del ciclo **do – while,** lo cual, implica que en todas las situaciones donde haya que recorrer una lista haba que controlar dicha situación con más instrucciones, lo cual va en detrimento del programa.

¿Qué paso entonces?: que por querer hacer eficiente la operación **liberar_lista,** estamos desmejorando todos los algoritmos en los cuales haya que recorrer alguna lista, porque necesitamos instrucciones diferentes para controlar las situaciones de lista vacía y de terminación de recorrido de la lista, además las operaciones de inserción y borrado tendrán orden de magnitud **O(n).** Observemos.

```
Function buscar_donde_insertar(L, d)
      if L= 0 or d < dato(L)
         return(0)
      end(if)
      y = L
      p = liga(L)
      while p <> L and dato(p) < d do
            y = p
            p = liga(p)
     end(while)
     return(y)
fin(buscar_donde_insertar)
```

En el algoritmo anterior controlamos la situación de lista vacía y decoración del primer dato con instrucciones adicionales. Nuestro algoritmo para insertar es:

```
Subprograma insertar(L, y, d)
    new(x)
    dato(x) = d
    casos
         :L  =  0:                          // inserta en lista vacía
                L = x
                liga(x) = x
          :Y = 0:                          // inserta en principio
                u = ultimo(L)
                 liga(x) = L
                 liga(u) = x
                 L = x
           :else:                         // inserta en sitio intermedio
                liga(x) = liga(y)
                liga(y) = x
      fin(casos)
fin(insertar)
```

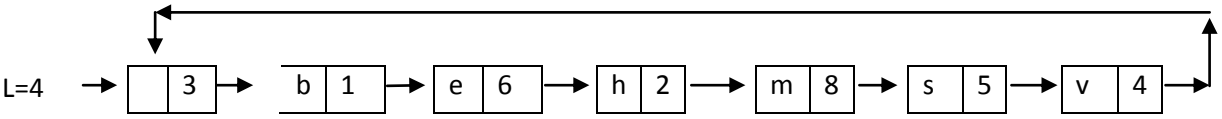
En el anterior algoritmo cuando el registro a insertar vaya a quedar al principio habrá que buscar el último registro de la lista, algoritmo que tiene orden de magnitud O(n), siendo n el número de registros de la lista. Este algoritmo lo presentamos a continuación.

```
Función ultimo(L)
    if L = 0   then
        return(0)
    end(if)
    p = L
    while liga(p)  <>  L do
            p = liga(p)
    end(while)
    return(p)
fin(ultimo)
```

Nuestro objetivo es poder tener una lista en la cual liberar lista sea eficiente y además controlar las situaciones de lista vacía y de terminación de recorrido con una sola instrucción.

Dicho objetivo lo logramos añadiendo un registro al principio de la lista, el cual llamaremos **registro cabeza.**

Lista Simplemente Ligada Circular Con Registro Cabeza



El registro ¿cabeza es un registro que, por lo general, no tendrá información correspondiente al objetivo que se esté representando en la lista ligada, siempre será el primer registro de la lista ligada y facilita todas las operaciones sobre la lista: construir la lista, recorrer la lista, insertar un registro, borrar un registro y liberar la lista.

El algoritmo de recorrido de las listas es:

```
Sub_programa recorre_lista(L)
        p = liga(L)
        while p <> L do
                write(dato)(p))
                p = liga(p)
        end(while)
```

fin(recorrer_lista)

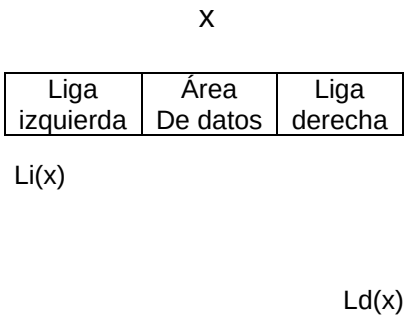
Para recorrer la lista se comienza en el segundo registro, es decir, a **p** se le asigna **liga(L).** La lista se termina de recorrer cuando **p** sea igual a **L.** Si la lista está vacía, inicialmente **p** queda valiendo **L** y no entrar al ciclo de recorrido.

Listas Probablemente Ligadas

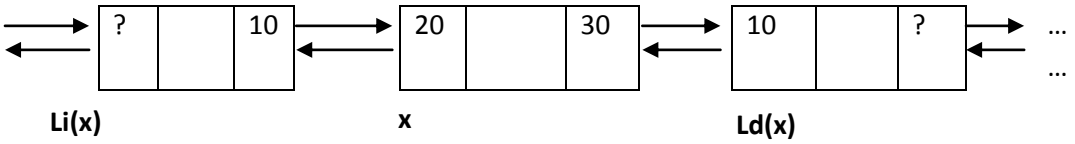
Hasta aquí hemos tratado con líneas ligadas que solo se pueden recorrer en un solo sentido: de izquierda a derecha.

Existen cierto tipo de problemas que exige que las listas ligadas se puedan recorrer en ambas direcciones: de izquierda a derecha y de derecha y a izquierda.

Para lograr esto debemos diseñar registros (nodos) con dos campos de liga: uno que llamamos **Li** y el otro que llamaremos **Ld.** Un esquema general de dicho nodo es:



Li(x): apunta hacia el registro anterior a **x,**
Ld(x): apunta hacia el registro siguiente de **x.**
Consideremos el siguiente segmento de lista doblemente ligada para determinar lo que llamaremos la propiedad fundamental de las listas doblemente ligadas.



El registro anterior a **x** es el registro **Li(x)** y el registro posterior a **x** es el registro **Ld(x).** La propiedad fundamental de las listas doblemente ligadas es:

Ld(Li(x)) = x = Li(Ld(x))

El campo liga derecha del registro liga izquierda de x es el mismo registro x, y, el campo liga izquierda del registro liga derecha de x es el mismo registro x.

Al manejar listas doblemente ligadas, las operaciones básicas son las mismas que en las listas simplemente ligadas: recorrer la lista, insertar un registro y borrar un registro.

Recorridos sobre la lista

En listas doblemente ligadas se tiene propiedad de que la lista se puede recorrer en ambas sentidos: de izquierda a derecha y de derecha a izquierda.

Recorrido de izquierda a derecha

El algoritmo es idéntico al de una lista simplemente ligada, la diferencia es que para avanzar sobre la lista lo haremos utilizando el campo de liga derecha.

```
Sub_programa recorre_izq_der (L)
        p = L
        while p <> 0 do
                write(dato(p))
                 p = Ld(P)
         end(while)
fin(recorre)
```

Recorrido de derecha a izquierda

Para recorrer la lista de derecha a izquierda debemos ubicarnos inicialmente en el último registro de la lista. Para ello utilizaremos una función, la cual llamaremos **último,** y a partir de este registro nos desplazaremos sobre la lisa utilizando el campo de liga izquierda.

```
Sub_programa recorre_der_izq(L)
        p = ultimo(L)
        while p <> 0 do
               write(dato(p))
                p = Li(p)
        end(while)
fin(recorre_izq_der)
```

Consideremos entonces la función **última** utilizada en dicho algoritmo. Para identificar el último registro en una lista doblemente ligada basta recorrer la lista buscando un registro cuyo campo de liga derecha sea cero.

```
Function ultimo(L)
```

```
        if L = 0 then
            return(0)
        end(if)
        p = L
        while  Ld(p) <> 0 do
                p = Ld(p)
        end(while)
        return(p)
fin(ultimo)
```

Inserción en una lista doblemente ligada

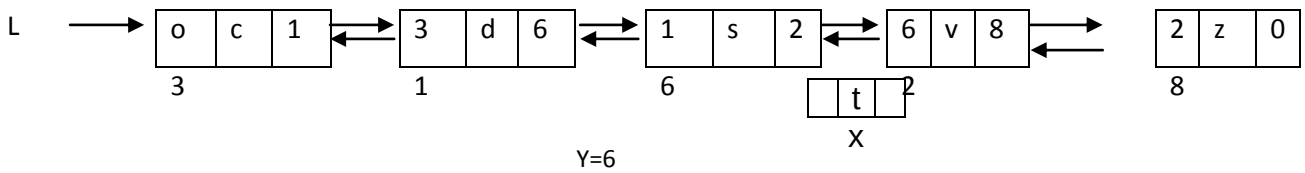
El proceso de inserción consta de dos partes: buscar donde insertar e insertar.

Buscar donde insertar

El subprograma buscar donde insertar es idéntico al de las listas simplemente ligadas. Se recorre la lista comparando el dato de p con el dato a insertar y se avanza con el campo de liga derecha. Se retorna también el registro a continuación del cual hay que insertar un nuevo registro con el dato d. nuestro algoritmo es:

```
Function buscar_donde_insertar(L, d)
        p = L
        y = 0
        while p <> 0 and dato(p) < d  do
                y = p
                p = Ld(p)
        end(while)
        return(y)
fin(buscar_donde_insertar)
```

Insertar un registro x a continuación de un registro y en la siguiente lista:



Si deseamos insertar la letra **"t"** en ella, primero ejecutamos buscar donde insertar y dicho algoritmo retornara **y = 6,** es decir, debemos insertar un nuevo registro con el dato **"t"** a continuación del registro 6.

Las instrucciones serían:
Ld(y) = x
Ld(x) = Ldy)
Li(x) = y
Li(Ld(x)) = x

Las instrucciones 1 y 2 preparan el registro **x** para que funcione adecuadamente cuando se conecte a la lista. Las instrucciones 3 y 4 conectan el registro **x** con la lista.

Si el dato a insertar fuera a quedar de ultimo la instrucción 3 fallaría, ya que estaría tratando de asignarle al registro **Ld(x),** el cual vale cero, el registro **x.**

Por consiguiente, cuando el registro a insertar vaya a quedar de último las instrucciones son:

Li(x) = y
Ld(x) = 0
Ld(y) = x

Las instrucciones para insertar un registro al principio de la lista son:

Ld(x) = L
If L <> 0 then
Li(L) = x
End(if)
L = x

En las instrucciones 2 y 3 se controla la eventualidad de que la lista **L** en la cual hay que efectuar la inserción este vacía.

Considerando las tres situaciones presentadas, nuestro algoritmo para insertar un registro **x** a continuación de un registro **y** en una lista doblemente ligada es:

```
Subprograma insertar(L, y, d)
      conseguir_registro(x)
       dato(x) = d
       LI(x) = y
      casos
          :y = 0
               ld(x) = L
               if L <> 0 then
                    Li(L) = x
              end_if)
              L =  x
        :Ld(y) = 0
              Ld(x) = 0
              Ld(y) = x
         :else:
              Ld(x) = Ld(y)
              Li(ld(x)) = x
              Ld(y) = y
    fin(casos)
fin(subprograma)
```

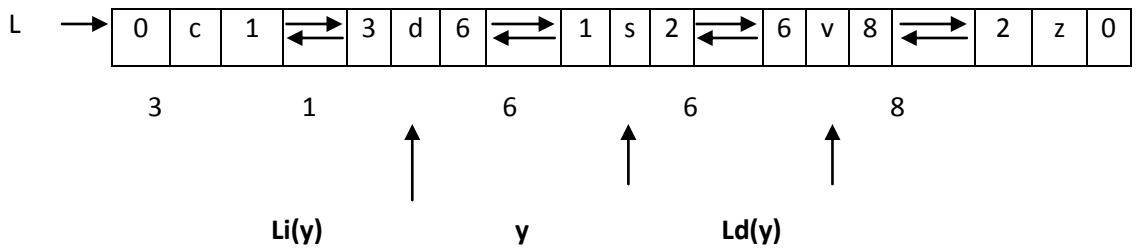
Borrado de un registro de la lista doblemente ligada

El proceso de buscar el dato a borrar es más sencillo que en las listas simplemente ligadas, ya que para cada registro conocemos fácilmente el registro anterior y el registro siguiente.

```
Function buscar_dato(L, d)
      if L = 0
           return(0)
       end(if)
       y = L
       while y <> 0 and dato(y) <> d do
            y = Ld(y)
       end(while)
        return(y)
fin(buscar_dato)
```

Nuestro algoritmo **buscar_dato(L, d)** retorna **y**: dirección del registro a borrar. La variable **y** será como si el dato buscando no se encuentra. Llamemos **y** el registro a borrar.

Recuerde que borrar un registro de una lista ligada, simplemente consiste en desconectar el registro de la lista.



Si queremos borrar el registro y = 6, aprovechando cuando que la lista es doblemente ligada, tenemos que el registro anterior a y es el registro Li (y), (Li(y)= 1) y el registro siguiente a y es el registro Ld(y), (Ld(y)=2.

Para desconectar el registro **y** en sentido de izquierda a derecha que hay que hacer es que cuando estemos en el registro **Li(y)** debemos pasarnos hacia el registro **Ld(y),** o sea brincarnos el registro **y** la instrucción para efectuar esta operación es :

Ld(Li(y)) = Ld(y)

Si estamos recorriendo de derecha a izquierda, cuando estemos ubicados en el registro Ld(y), debemos pasarnos hacia el registro Li(y), o sea brincarnos el registro y la instrucción para efectuar esta operación es :

Li(Ld(y)) = Li(y)

Si el registro a borrar es el último, es decir, Ld(y) = 0, la instrucción Li(Ld(y)) = Li(y) fallara ya que nos estaríamos al campo liga izquierda del registro Ld(y), el cual es cero.

Por consiguiente, cuando el registro a borrar sea último la única instrucción para efectuar el borrador es:

Ld(Li(y)) = Ld(y)

Si el registro a borrar es el primero, es decir**, Li(y)** = 0. La instrucción **Ld(Li(y))** = **Ld(y)** fallara, ya que el registro Li(y) es cero.

Cuando el registro a borrar sea el primero, el registro que debe quedar de primero es el que estaba de segundo, y la instrucción para lograr esto es **L = Ld(y)** y como el nuevo primero debe tener cero en el campo de liga izquierda, le asignaremos cero al campo de liga izquierda de **L: Li(L) = 0**

Sin embargo, como la lista pudo haber tenido solo un registro, cuando ejecutamos la instrucción **L=Ld(L), L** queda valiendo cero y la instrucción **Li(L)=0** fallara. Por consiguiente la instrucción **Li(L)=0** debemos condicionarla al hecho de que **L** sea diferente de cero.

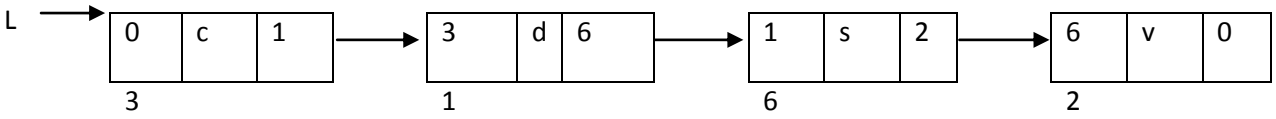
Considerando estas situaciones, el algoritmo completo para borrar un registro de una lista doblemente sería:

```
Sub_programa borrar (L,y)
        if y = 0 then
                write('dato a borrar no existe')
                return
        end(if)
        casos
                :L(y) = 0                       // borra el primer registro
                        L = Ld(y)
                        if L <> 0 then
                                li(L) = 0
                        end(if)
                :Ld(y) = 0:                     // borra el último registro
                        Ld(Li(y)) = 0
                :else:                          / borra registro intermedio
                        Li(Ld(y)9 = Li(y)
                        Ld(Li(y)) = Ld(Y)
        Fin(casos)
        free(y)
fin(borrar)
```

Ejercicio:
Idee una situación de la cotidianidad comercial o industrial donde sea útil la implementación de un software que mediante los tipos de listas que acabamos de identificar se pueda dar solución a una necesidad específica. y plantee un algoritmo con la forma de aplicación de dichas estructuras de almacenamiento.

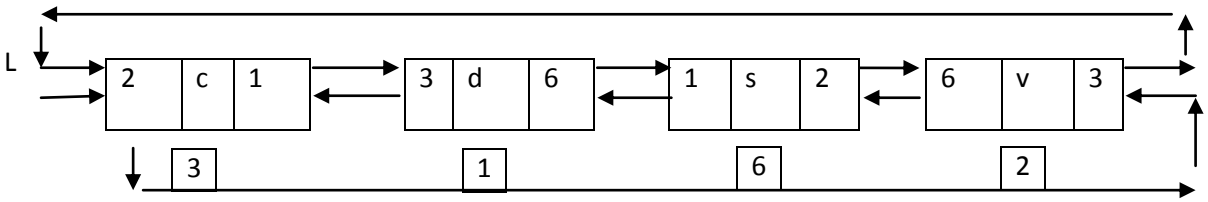
G. Diferentes Tipos de Listas Doblemente Ligadas

Listas doblemente ligadas



El campo de liga izquierda del registro es cero porque el primer registro no tiene anterior; y el campo de liga derecha del último registro también vale cero porque el último registro no tiene siguiente. Los algoritmos para las operaciones básicas ya fueron representados con anteriormente

Listas doblemente ligadas circulares



En la instrucción doblemente ligadas circulares el campo de liga de derecha del último registro apunta hacia el primer registro de la lista y el campo de liga izquierda del primer registro apunta hacia el último registro de la lista.

En las listas doblemente ligadas circulares se presentan los mismos problemas que se tenían con las listas simplemente ligadas circulares; las situaciones de lista vacía y de fin de recorrido se deben controlar con situaciones diferentes.

Algoritmos para recorridos

```
Sub_progrma recorre:izq_der(L)
      if L = 0 then
            return
      end(if)
      p = L
      do
          write(dato(P))
          p = Ld(P)
       while p <> L
fin(recorre_izq_der)
Sub_programa recorre_der_izq(L)
      if L = 0 then
```

```
            return
        end(if)
        p = Li(L)
        do
            write(dato(p))
            p = Li(p)
        while p <> Li(L)
fin(recorrer_der_izq)
```

Como hemos estado tratando, el proceso de inserción consta de dos partes: buscar donde insertar e insertar.

```
Function buscar_donde_insertar(L, d)
    if L = 0 then
        return(0)
    end(if)
    if d < dato(L) then
        return(0)
    end(if)
    y = L
    p = Ld(L)
    while p <> L and dato(p) < d do
        y = p
        p = Ld(p)
    end(while)
    return(y)
fin(buscar_donde_insertar)
```

Nuestro algoritmo **buscar_donde_insertar(L, d)** retornara también un registro **y,** el cual apunta hacia el registro a continuación del cual hay que insertar un nuevo registro. La variable **y** valdrá cero si el registro a insertar debe quedar de primero o si la lista está vacía.

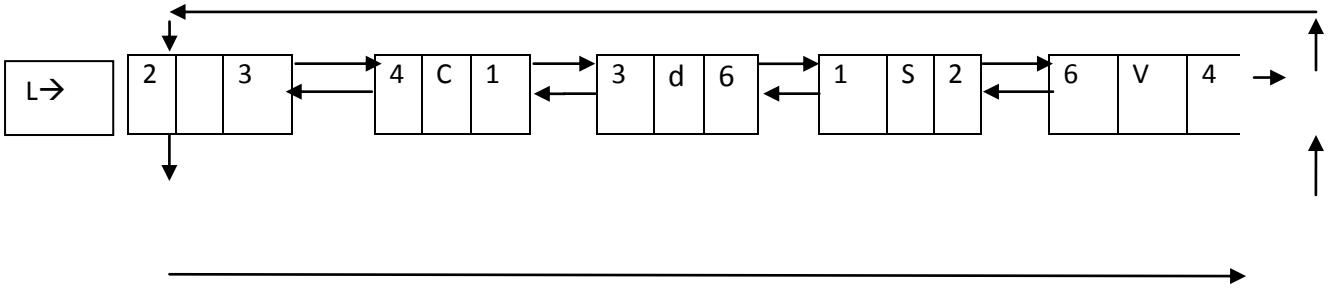
Subprograma insertar (L, y, d)

1. new(x)
2. dato(x)
3. if L = 0 then
4. Li(x) = x
5. Ld(x) = x
6. L = x
7. Return

8. End(if)
9. Sw = 0
10. If y = 0
11. Y = Li(L)
12. Sw = 1
13. End(if)
14. Ld(x) = Ld(y)
15. Li(x) = y
16. Li(Ld(x)) = x
17. Ld(y) = x
18. If sw = 1 then
19. L = x
20. End(if)
21. Fin(insertar)

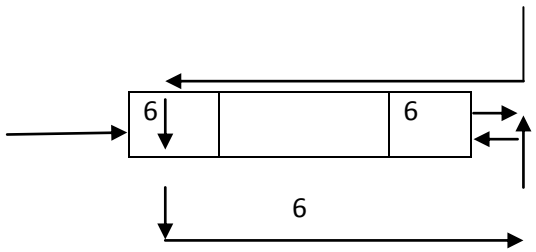
Instrucciones 3 a 8 controlan la situación de la lista vacía. Instrucciones 9 a 13 controlan el dato a insertar quede primero. Dado que las instrucciones para insertar un registro en cualquier parte de la lista son las mismas, debemos controlar la situación en la cual hay que insertar el nuevo registro al principio de la lista **(y = 0).** Para ello manejamos un swiche **(sw)** el cual inicializamos en cero y lo convertimos en uno si el registro a insertar debe quedar primero. En instrucciones 18 a 20 consideramos esta situación y actualizamos **L** si el registro a insertar será el primero.

Listas doblemente ligadas circulares con registro cabeza



En las listas doblemente ligadas circulares con registro cabeza se tiene la ventaja de que la representación de la lista vacía tiene como mínimo el registro cabeza:

Lista vacia: L

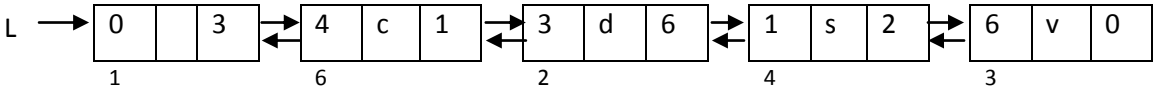


Veamos para esta nueva representación la forma como se facilitan y se hacen más eficientes los algoritmos para las operaciones básicas sobre listas ligadas.

```
Sub_programa recorre_izq_der(L)
        p = Ld(L)
        whiele P <> L do
                write(dato(p))
                p = Ld(p)
        end(while)
fin(recorre_izq_der))
Sub_programa recorre_der_izq(L)
        p = Li(L)
        while  p <> L do
                write(dato(p))
                p = Li(L)
        end(while)
fin(recorre_der_izq)
function buscar _donde_insertar(L, d)
      Y = L
      p = Ld(L)
      while    p <> L and dato(p) < d do
            y = p
            p = Ld(p)
       end(while)
        return(y)
fin(buscar_donce_insertar)
Subprograma insertar(L, y, d)
          new(x)
      dato(x) = d
       Ld(x) = Ld(y)
       Li(x) = y
       Li(Ld(x)) = x
        Ld(y) = x
fin(insertar)
```

```
Function buscar_dato(L, d)
        p= Ld(L)
        while p <> L and dato(p) <> d do
                p = Ld(p)
         end(while)
          return(p)
fin(buscar_dato)
Sub_programa borrar(L, x)
       if x = L then
           write('dato no existe')
            return
       end(if)
       Ld(Li(x)) = Ld(x)
       Li(Ld(x)) = Li(x)
fin(borrar)
```

Listas doblemente ligadas no circulares con registro cabeza



Dejemos al estudiante la tarea de desarrollar los algoritmos básicos para esta otra variación de lista doblemente ligada.

Ejercicio:
Mediante un trabajo colaborativo en un grupo de n estudiantes, coordine el desarrollo de un subprograma para dada uno de los siguientes planteamientos o requerimiento.
Escribir un algoritmo que dada una lista ligada, intercambie el primer registro con el último, el segundo con el penúltimo, el tercero con el antepenúltimo y así sucesivamente. Considere todos los tipos de lista.

Escribir algoritmo para determinar el promedio de los datos de una lista ligada. Considere todos los tipos de lista.

Dada una lista con sus datos ordenados en forma ascendente. Puede haber datos repetidos. Escriba un algoritmo que elimine los registros que tengan datos repetidos. Considere todos los tipos de lista.

Al Finalizar el trabajo, cada grupo deberá demostrar porque su programa es eficiente, funcional y que se logró desde la interacción del trabajo colaborativo.

3.3. Estructuras en Pilas o Lifo

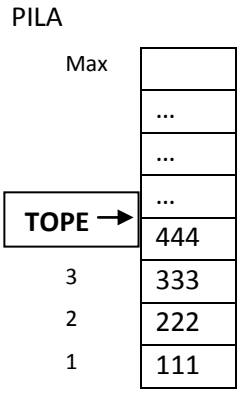
Una pila es una lista ordenada de elementos en la cual la adición o borrado de elementos es posible solo por los extremos de la lista. Por ende los elementos de la pila serán eliminados en orden inverso al que se ingresaron. O sea el último elemento que se ingresa a la pila será el primero en eliminarse, por ello recibe esta estructura el nombre de LIFO (last input firs output)

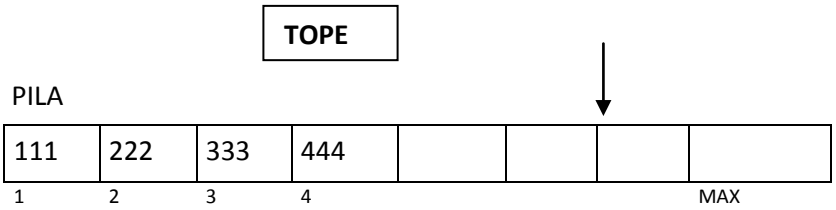
Funciona similar a una pila de platos recién lavados, es decir el cocinero arruma los platos en orden de lavada, el primero que lava quedará debajo de la pila, y el último en la parte superior, así al momento de utilizar un plato limpio tomará el de la parte superior, el último que lavo.

Representación de Pilas

Las pilas no son estructuras de datos definidas sino que se representan mediante arreglos o listas enlazadas.

Si usásemos arreglos debemos definir el tamaño máximo de la pila y además una variable auxiliar denominada TOPE que será el apuntador al último dato de la pila, estas son dos posibles ilustración es de una pila.





Algoritmo para definir una estructura de PILA

Funciones:

crear()→pila
apilar(elemento,pila)→pila
desapilar(pila)→pila
tope(pila)→elemento
esvacia(pila) → lógico

Axiomas:

para todo E pilas, i E elementos
Esvacia(crear) ::= verdad
esvacia(apilar(i, O)) :: =falso
desapilar(crear) ::= crear
desapilar(apilar(i, P)) ::= P
tope(crear) ::= error
tope(apilar(i, P)) :: i

- Crear, es la función constante que crea la pila vacía.
- Apilar, es l función que permite incluir un elemento en una pila si tenemos una pila con 3 elementos R, S, T la estructura seria:

Apilar (T, apilar(S, apilar(R, crear)))

Según la forma general apilar (i, P), el elemento i es T y P =Apilar(S,Apilar(R,Crear))

Desapilar extrae el último elemento de la pila y deja una nueva pila la cual queda con un elemento menos.

Tope da información sobre el último elemento que se halla en la pila sin extraerlo es vacía chequea que haya elementos en la pila.

a. Algoritmo para representación de PILA

La forma más simple es utilizar un arreglo de una dimensión y una variable, que llamaremos **Tope,** que indique la posición del arreglo en la cual se halla el último elemento de pila. La función **crear n seria** simplemente definir un vector y una variable, que llamaremos **Tope,** la cual inicialmente tendrá el valor cero.

```
Dimensión pila(100)
tope = 0
```

La función **Esvacia** consiste simplemente en preguntar por el valor de **Tope:**

```
If tope = 0 then
    verdad
else
    falso
end(if)
```

La función **Tope,** que da información sobre el último elemento de la pila es:

```
If tope = 0 then
   error
```

```
    else
        return(pila(tope))
    end(if)
```

Como se puede observar, estas funciones son tan sencillas y tan cortas que no se justifica elaborar algoritmos independientes para ellas. En su defecto se codificaran en el proceso que se esté efectuando, cuando sea necesario. Los algoritmos para apilar y desapilar se presentan en los siguientes subprogramas:

```
Sub_programa apilar(pila, n, tope, dato)

     if tope = n then
         pila_llena
     else
        tope = tope + 1
        pila(tope) = dato
    end(if)
fin(apilar)
```

n y dato son parámetros por valor y **tope** y **pila** parámetros por referencia
pila es un vector con capacidad de **n** elementos **tope** es la variable que indica la posición del vector en la cual se halla el último elemento de la pila.
dato es la variable que lleva la información a guardar en la pila.

Ejemplo:

	1	2	3	4	5	6	7	8	9
pila	A	B	C	D					

En nuestro ejemplo el vector se llama **Pila, n** es 9 y **tope** es 4, el subprograma PILA_LLENA invocara cuando **tope** sea 9. Dicho subprograma sacara un mensaje apropiado y detendrá el proceso.

```
Sub_programa desapilar(pila, tope, dato)
   if  tope = 0 then
         pila_vacia
   else
       dato = pila(tope)
        tope = tope – 1
   end(if)
end(sub_programa)
```

Pila, tope y dato son parámetros por variable. dato es la variable que retorna la información sacada de la pila.

a. Representación de pilas como lista ligada

Siempre que se desee representar un objetivo como lista ligada lo primero que se debe hacer es definir la configuración del registro.

	Dato	Liga
X		

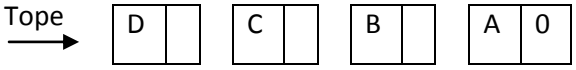
Dos campos: dato y liga

Dato(x) = se refiere al campo dato en el registro **x.**
liga(x) = se refiere al campo liga del registro **x,** es el campo que apunta hacia otro registro. Representemos como lista ligada la siguiente pila:

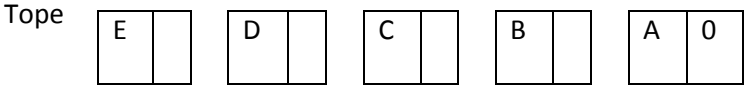
Pila

D
C
B
A

Dibujémosla de la forma como es común hacerlo con listas ligadas



Apilar: consistente en insertar un registro al principio de una lista ligada.




```
Sub_programa apilar(tope, d)
        nex(x)
        dato(x) = d
        liga(x) = tope
        tope = x
fin(apilar)
```

Se observa que es más fácil y más eficiente que manejando la pila en vectores, ya que en listas ligadas no hay necesidad de controlar pila llena antes de apilar.

Desapilar

Consiste en borrar el primer registro de una lista ligada. Para desapilar es necesario saber si la pila está o no vacía. Si está vacía no se podrá desapilar. La pila está vacía cuando tope = 0

```
Sub_programa desapilar(tope, d)
    if tope = 0 then
        pila_vacia
    else
        d = dato(tope)
        x = tope
        tope = liga(tope)
        devolver_registro(x)
fin(desapilar)
```

Ejercicio de Aplicación:
Distribuya las diferentes posibilidades de manejo de pilas entre varios estudiantes, cada uno deberá proponer una actividad con pilas. Luego conformando grupos pequeños, deberán conseguir en el salón los subprogramas que permitan desde un programa principal realizar satisfacer las siguientes necesidades considerando todas las posibilidades.

Dibuje los distintos estados de una estructura tipo pila si se llevan a cabo las siguientes operaciones. Muestre con cómo va quedando la pila y el puntero al tope de la misma. Considere que la pila está inicialmente vacía (TOPE = 0).

a. Insertar (PILA, X)
b. Insertar (PILA, X)
c. Eliminar (PILA, X)
d. Insertar (PILA, X)
e. Eliminar (PILA, X)
f. Insertar (PILA, X)

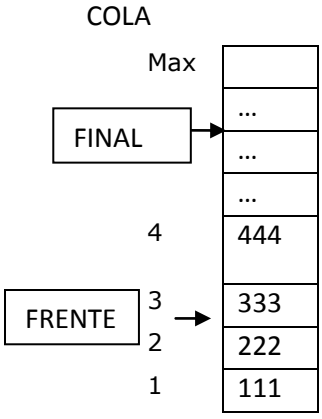
Con cuantos elementos quedó la pila?
Hubo algún caso de error (Desbordamiento o Subdesbordamiento)? Explique.

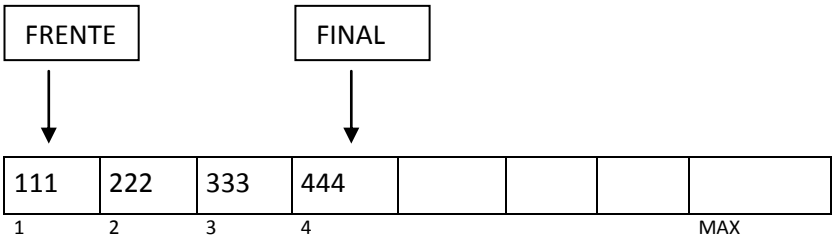
3.4. Estructura de Colas o Fifo

Una cola es una lista de elementos que se introducen por un extremo y se eliminan por el otro, es decir se eliminan en el mismo orden que se insertan, por ello el primero que entra será el primero en salir. Son por ello llamadas FIFO, en la vida real existen muchos casos de colas, ejemplo: la cola para ingresar a un evento o transporte. Ahí la primera en iniciar la fila será la primera en tomar el servicio.

1. Representación de Colas

Al igual que las pilas las colas no existen como estructuras de datos, por ello se representan mediante arreglos o listas enlazadas. De igual manera si usamos arreglos debemos definir el tamaño máximo de la cola y dos variables auxiliares. Una de ellas para guardar la posición del primer elemento de la cola (FRENTE) y el otro para la posición del último elemento (FINAL), gráficamente se verían así:





2. Insertar un elemento en una cola

Las inserciones se llevan a cabo por el final de la cola, mientras que la eliminación se hará por el frente una vez que el primero en entrar es el primero en salir.

El algoritmo sería el siguiente:
Función: InsertarCola(Cola, Max,Frente, Final, Dato)
Este algoritmo inserta el elemento Dato al final de la Cola, Frente y Final son los punteros que indican respectivamente el inicio y fin de la Cola, Max es el máximo número de elementos que puede almacenar la cola.

1. Si Final < Max ----verificando que haya espacio libre
2. Entonces
3. Final=Final + 1 ------ actualiza el final
4. Cola[Final]=Dato
5. Si Final=1 entonces ----------se insertó el primer elemento de cola
    a. Frente=1
6. FinSi
7. Si No
8. Imprima "Desbordamiento"
9. FinSi

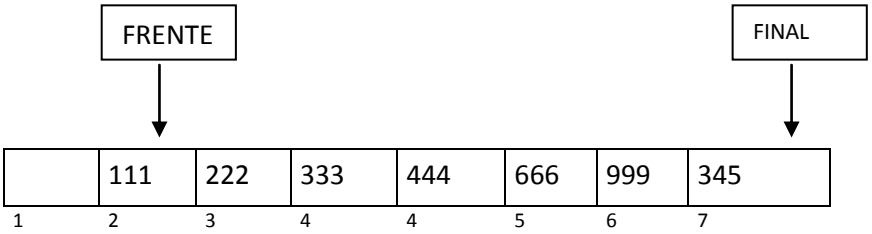
Eliminar un elemento de una cola
Función: EliminarCola(Cola, Frente, Final, Dato)

Este algoritmo elimina el primer de la Cola y lo almacena den Dato, Frente y Final son los punteros que indican respectivamente el inicio y fin de la Cola.

1. Si Frente # 0 ----verificando que la cola no este vacía
2. Entonces
3. Dato=Cola[Frente]
4. Si Frente=Final ----------sí hay un solo elemento
    a. Entonces
    b. Frente=0 ----------Indica Cola vacía
    c. Final =0
5. Si No
6. Frente =Frente +1
7. FinSi
8. SiNo
9. Imprima "Desbordamiento"
10. FinSi

3. Colas circulares

Para un uso más eficiente de la memoria, podemos tratar las colas como una estructura circular. Es decir el elemento anterior al primero es el último



A medida que se insertan o eliminan elementos se actualizan los punteros Frente y Final. En esta lista de cola, se tiene Frente=2 y Fina=8, si eliminamos 111 y 222 Frente quedaría igual a 4 y al insertar un elemento en la cola como Final = Max, se llevaría el apuntador a la primera posición que estaba vacía haciendo Final=1, así se da un mejor aprovechamiento de la memoria, ya que al eliminar un elemento dicha posición queda disponible para futuras inserciones

InsertarCircular(ColaCir, Max, Frente, Final, Dato)

Se insertará el elemento Dato al final de la ColaCir, Frente y Final, son punteros indicadores de inicio y fin de la cola, Max es el tamaño de la colacir.

1. Si ((Final=Max) y(Frente=1) o ((Final+1)=Frente) Entonces
2. Imprima “Desbordamiento” -----La cola está llena
3. Si No
4. Si Final=Max entonces
5. Final=1
6. Sino
7. Final=Final+1
8. FinSi
9. ColaCir[Final]=Dato
10. Si Frente=0 entonces
11. Frente=1
12. FinSi
13. FinSi

EliminarCircular(ColaCir, Max, Frente, Final, Dato)

Se eliminará el primer elemento de la ColaCir y lo almacena en Dato, Frente y Final, son punteros indicadores de inicio y fin de la cola, Max es el tamaño de la Colacir.

Si Frente=0 entonces -------verificando si la cola está vacía
    Imprima “Subdesbordamiento”

1. SiNo
2. Dato=ColaCir[Frente]
3. Si Frente=Final entonces --------- por si hay un solo elemento
4. Frente=0, Final=0
5. SiNo
6. Si Frente=Max entonces
7. Frente+1
8. SiNo
9. Frente=Frente+1
10. FinSi
11. FinSi
12. FinSi

Información complementaria

Documento propio para la conceptualización, lógica y codificación de las estructuras de datos conformadas por lista, pilas, colas, las

www.lcc.uma.es/~lopez/modular/tads/pilas_colas.pdf, acceso septiembre 10 de 20011

http://www.programacionfacil.com/estructura_de_datos:colas

http://www.youtube.com/watch?v=5Y6itIz2M8A

Ejercicio de Aplicación

Se tiene un cola Circular C de 6 elementos, inicialmente la cola está vacía, es decir que Frente=Final=0, ilustre gráficamente el estado de C, después de realizar las siguientes operaciones.

1. Insertar los elementos AAA, BBB, CCC
2. Eliminar el elemento AAA
3. Insertar los elementos DDD, EEE,FFF
4. Insertar el elemento GGG
5. Insertar el elemento HHH
6. Eliminar los elementos BBB, CCC

Con cuantos elementos quedó C?
Hubo algún caso de error, desbordamiento o Subdesbordamiento? Explique

Escriba un algoritmo que inserte un elemento en una cola circular y otro para eliminar un elemento de una cola circular, considere todos los casos.

Mediante el uso de una cola simule el movimiento de clientes en una cola de espera de un banco.

Usted deberá indagar en su entorno, social, industrial, comercial o educativo, que escenarios o circunstancias Problemicas del manejo de la información, son aptos para plantear soluciones basadas en Listas, Pilas y Colas, y proponer al menos una solución con cada uno de estos Componentes.
Propio para trabajo en grupos uno con cada tipo de estructura, con confrontación de potencialidades y limitantes de cada aplicación, dadas la estructura utilizada.

4. ESTRUCTURAS EN ARBOL

Hasta el momento solo se han estudiado estructuras lineales estáticas y dinámicas de datos: a un elemento solo sigue otro elemento. Al analizar la estructura árbol se introducen ramificación entre nodos.

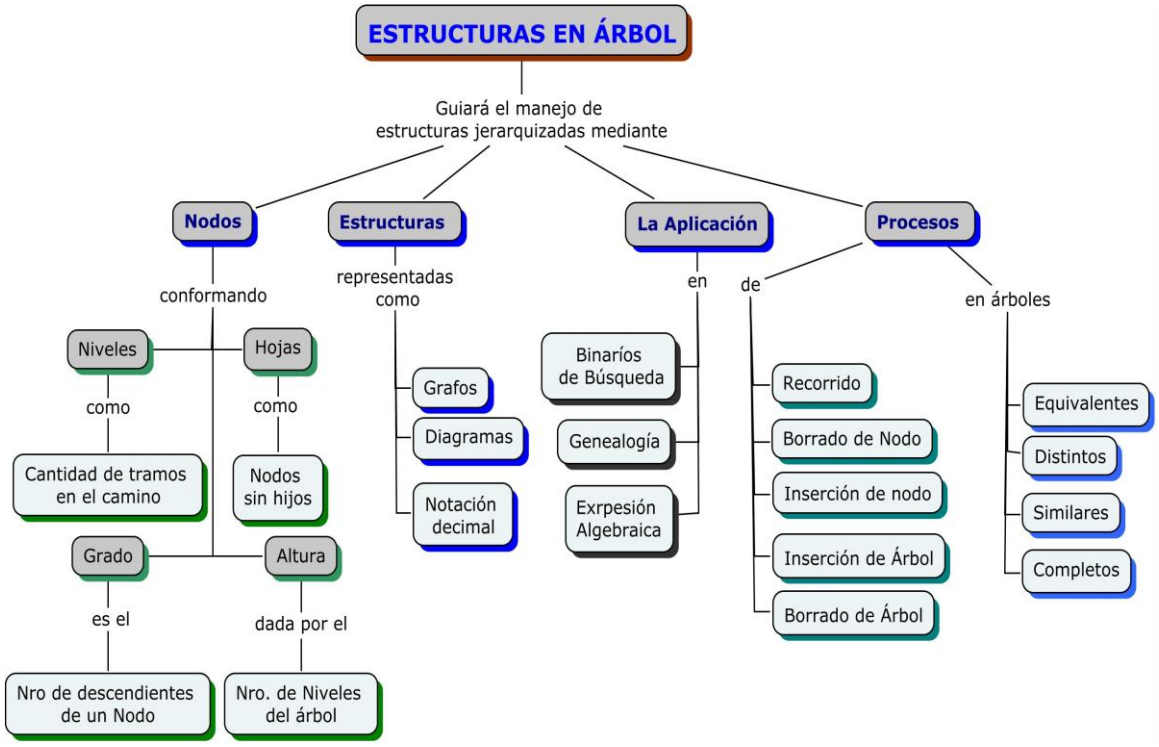
Los árboles representan las estructuras no-lineales y dinámicas de datos más importantes en computación. Dinámicas, puesto que la estructura árbol puede cambiar durante la ejecución de un programa. No lineales, puesto que a cada elemento del árbol puede seguirle varios elementos.

ESTRUCTURAS ESTATICAS	ESTRUCTURAS DINMICAS
arreglos	listas
registros	arboles
conjuntos	

Es de observar que las pilas y colas no fueron consideradas en esta clasificación, puesto que dependen de la estructura que se utilice para implementarlas.

ESTRUCTURAS LINEALES	ESTRUCTURAS NO-LINEALES
arreglos	arboles
registros	
conjuntos	
pilas	
colas	
listas	

4.1. Mapa Conceptual



OBJETIVO GENERAL

Guiar el estudiante en la aplicación de los procedimientos para el uso de estructuras jerárquicas en árbol con el fin de automatizar los procesos que así lo requieran.

OBJETIVOS ESPECÍFICOS

- Involucrar el estudiante la conceptualización y manejo de árboles como estructuras no lineales de almacenamiento de datos en memoria.
- Inducir el estudiante en el planteamiento de soluciones a situaciones problemicas reales mediante el uso de árboles binarios.

Prueba Inicial

Según lo abordado hasta el momento y su conocimiento previo, trate de responder esta prueba diagnóstica cuyo objetivo es que identifique y evidencie su manejo conceptual y práctico sobre las estructuras de almacenamiento en memoria no lineales.

1. Conoce y maneja usted las estructuras lineales de almacenamiento de datos en memoria? Indique algunas características funcionales de ellas.
2. De los elementos trabajados hasta ahora cuales no se constituyen estructuras de almacenamiento por sí mismas y porque?
3. A qué tipo de estructura de almacenamiento se le conoce como no lineal y porque?
4. De qué elementos está compuesta una estructura de tipo Árbol? Y cuál es su aplicabilidad? indique con ejemplos.
5. Que característica tienen los árboles binarios que los diferencie del resto?
6. Conoce usted alguna forma o parámetros particulares a tener en cuenta al momento de hacer referencia a un nodo particular de un árbol?

4.2. Árboles en General

Un **árbol** es una estructura jerárquica aplicada sobre una colección elementos u objetos llamados nodos; uno de los cuales es conocido como raíz. Además se crea una relación o parentesco entre los nodos dando lugar a términos como padre, hijo, hermano, antecesor, sucesor, ancestro, etc.

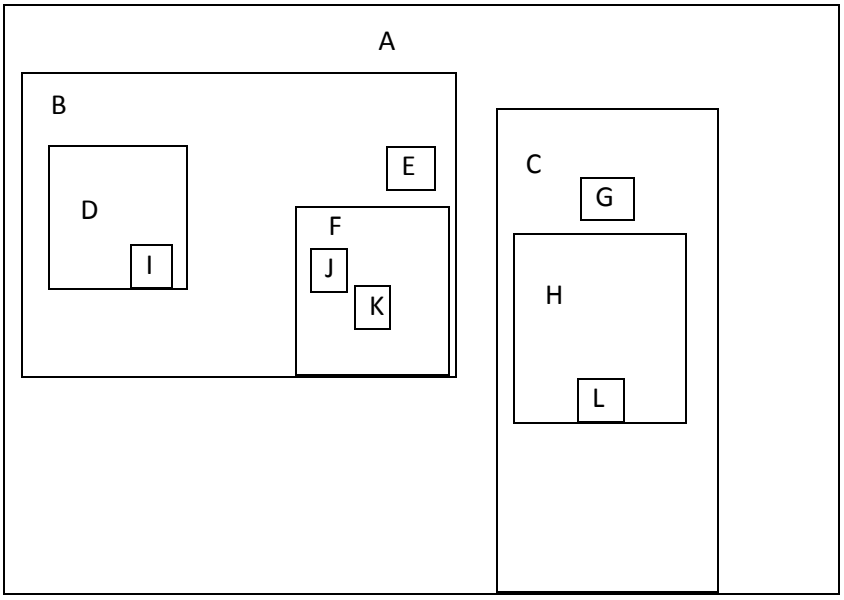
Formalmente se define un árbol de tipo T como una estructura homogénea que es

Esta es la forma de lectura jerarquizada del árbol según

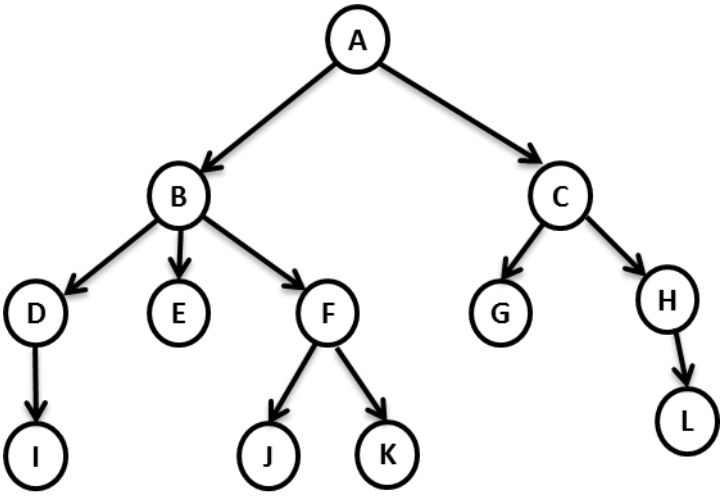
Animación de (A (B (D (I), E, F (J, K)), C (G, H(L)))

Notación Decimal, Deway
1. A, 1.1.B, 1.1.1.D, 1.1.1.1.I, 1.1.2.E, 1.1.3.1.J, 1.1.3.2.K, 1.2.C, 1.2.1.G, 1.2.2.H, 1.2.2.1.L

Diagrama de Venn



Representación de Grafos.



Una forma particular de árbol puede ser la estructura vacía.
Se utiliza la recursión para definir un árbol porque representa la forma más apropiada y porque además es una característica inherente a los mismos.

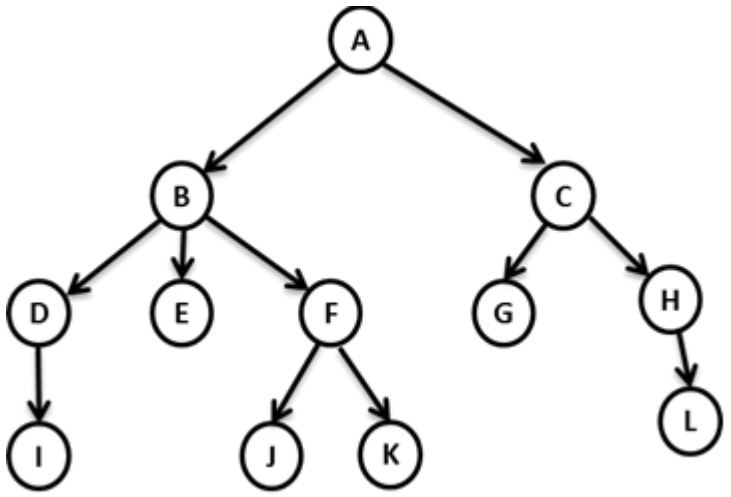
Los arboles tienen una gran variedad de aplicaciones. Por ejemplo, se pueden utilizar para representar fórmulas matemáticas, para organizar adecuadamente la información, para registrar la historia de un campeonato de tenis, para construir u árbol genealógico, para el análisis de circuitos electrónicos y para numerar los capítulos y secciones de un libro.

1. Características y propiedades de los arboles

Las siguientes son las características y propiedades más importantes de los árboles en general:

1. Todo árbol que no es vacío, tiene un único nodo raíz.
2. Un nodo X es descendiente directo de un único nodo un nodo Y en este caso es común utilizar la expresión **X es hijo de Y.**
3. Un nodo X es antecesor directo de un nodo Y, si el nodo X apunta al nodo Y. en este caso es común utilizar la expresión **X es padre de Y.**
4. Se dice que todos los nodos que son descendientes directos (hijos) de un mismo nodo (padre), son **hermanos.**
5. Todo nodo que no tiene ramificaciones (hijos), se conoce con el nombre de **terminal** u **hoja.**
6. Todo nodo que no es raíz, ni terminal u hoja se conoce con el nombre de **interior. Grado** es el número de descendientes directos de un determinado nodo. **Grado del árbol** es el máximo grado de todos los nodos del árbol.
7. **Nivel** es el número de arcos que deben ser recorridos para legar a un determinado nodo. Por definición la raíz tiene nivel 1
8. **Altura** del árbol es el máximo número de niveles de todos los nodos del árbol.

Ejemplo



A es la raíz del árbol.
B es hijo de A.
C es hijo de A.
D es hijo de B.
E es hijo de B.
L es hijo de H.
A es padre de B.
B es padre de D.
D es padre de I.
C es padre de G.
H es padre de L.
B y C son hermanos.
D, E y F son hermanos
G y H son hermanos.
J y K son hermanos.
I, E, J, K, G y L son nodos terminales u hojas.
B, D, F, C y H son nodos interiores.

El grado del nodo A es 2.
el grado del nodo B es 3.
el grado del nodo C es 2.
el grado del nodo D es 1.
el grado del nodo E es 0.
el grado del árbol es 3.
El nivel del nodo A es 1.
el nivel del nodo B es 2.
el nivel del nodo D es 3.
el nivel del nodo C es 2.
el nivel del nodo L es 4.
La altura dela árbol es 4.

2. Longitud de camino

Se define la longitud de camino **X** como el número de arcos que deben ser recorridos para llegar desde la raíz a dicho nodo. Por definición la raíz tiene longitud de camino 1, sus descendientes directos longitud 2 y así sucesivamente. Según la figura anterior el nodo B tiene longitud de camino 2, el nodo I longitud de camino 4 y el nodo H longitud de camino 3.

Longitud de camino interno

La longitud de camino interno es la suma de las longitudes de camino de todos los nodos del árbol. Puede calcularse por medio de la siguiente formula:

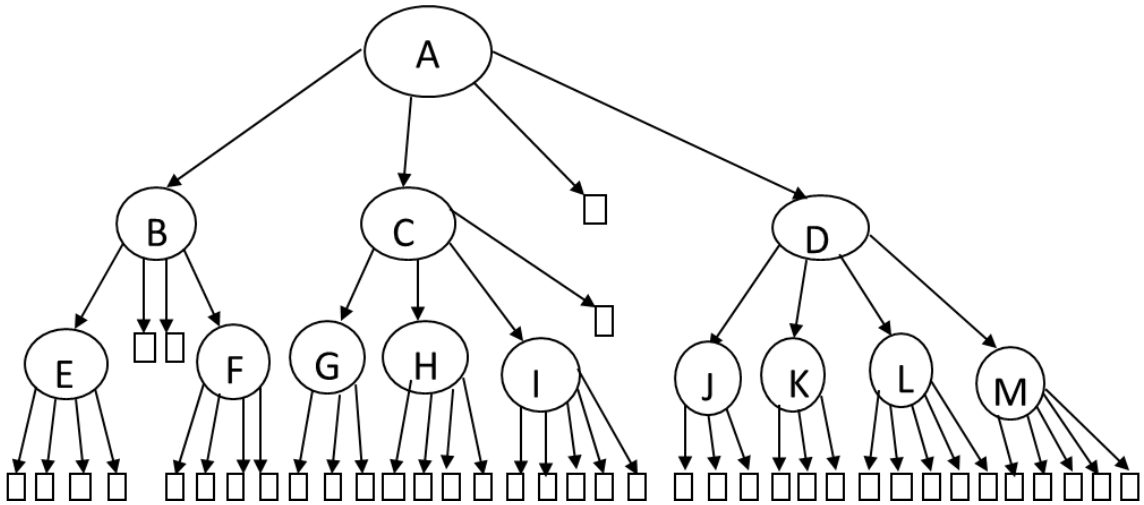
$$LCI = \sum_{i=1}^{h} n_i * i$$

Donde i representa el nivel del árbol, h su altura y n_i el número de nodos en el nivel i.

La LCI del árbol anterior se calcula así: LCI = 1 * 1 + 2 * 2 + 5 * 3 + 4 * 4 = 36

Ahora bien, la media de la longitud de camino interno (LCIM) se calcula dividiendo la LCI entre el número de nodos del árbol (n). Se expresa:

LCIM = LC/n



La longitud de camino externo: LCE = 1 * 2 + 3 * 3 + 36 * 4 = 155

Y la media de la longitud de camino externo: LCEM = 155/40 = 3.87

Ejercicio:
Identifique en la información de diario vivir, estructuras jerarquizadas que puedan ser presentadas como árboles, Plantee su estructura gráfica según las formas vistas, determine sus caminos de lectura, elementos, niveles y longitud.

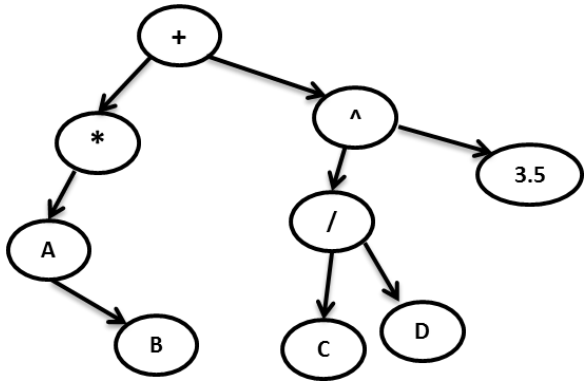
4.3. Árboles Binarios

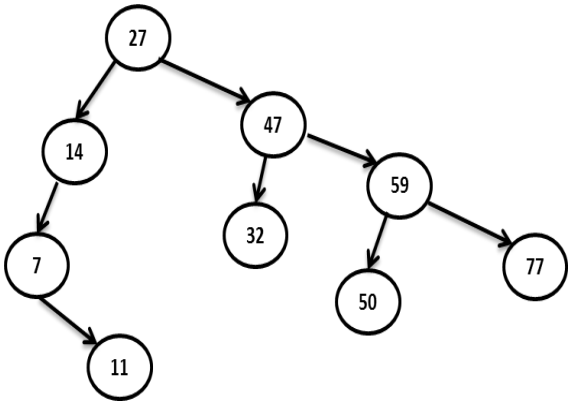
Un árbol ordenado es aquel en el que las ramas de los nodos del árbol están ordenadas. Los arboles ordenados de grupo 2 son de especial interés puesto que representan una de las estructuras de datos más importantes en computación, conocida como arboles binarios.

En un árbol binario cada nodo puede tener como máximo dos subárboles; y siempre es necesario distinguir entre el subárbol izquierdo y el subárbol derecho. Formalmente se define un árbol binario de tipo T como una estructura homogénea que es la concatenación de un elemento tipo T, llamada raíz, con dos árboles binarios disjuntos, llamados subárbol izquierdo y derecho de la raíz. Una forma particular de árbol binario puede ser la estructura vacía.

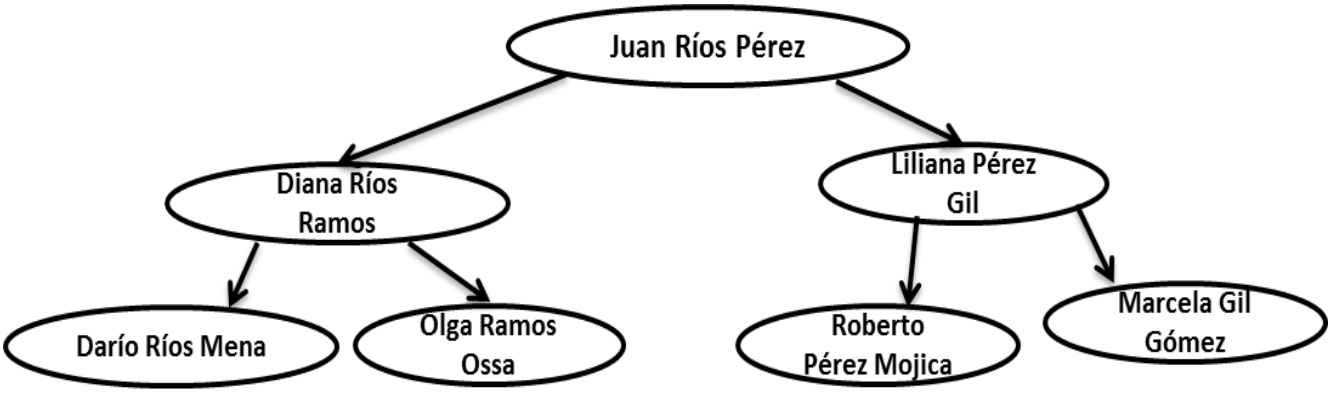
Los arboles binarios tienen múltiples aplicaciones. Se lo puede utilizar para representar una estructura en la cual es posible tomar decisiones con dos opciones en distintos puntos de un proceso, para representar un árbol genealógico (construido en forma ascendente y donde se muestran los ancestros de un individuo dado), para representar la historia de un campeonato de tenis (construido en forma ascendente y donde existe un ganador, 2 finalistas, 4 semifinalistas y así sucesivamente) y para representar Expresiones algebraicas construidas con operadores binarios, son algunos de sus múltiples usos. Veamos algunas estructuras de aplicación de árboles binarios.

a) Árboles binarios de búsqueda.



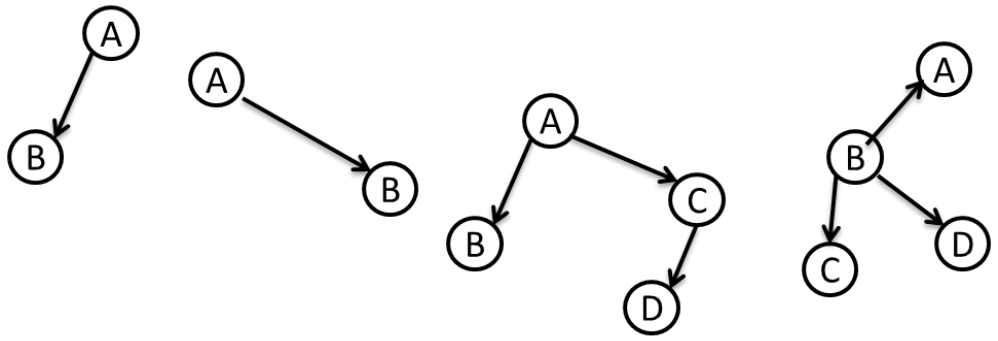


b) Representación de una expresión algebraica.

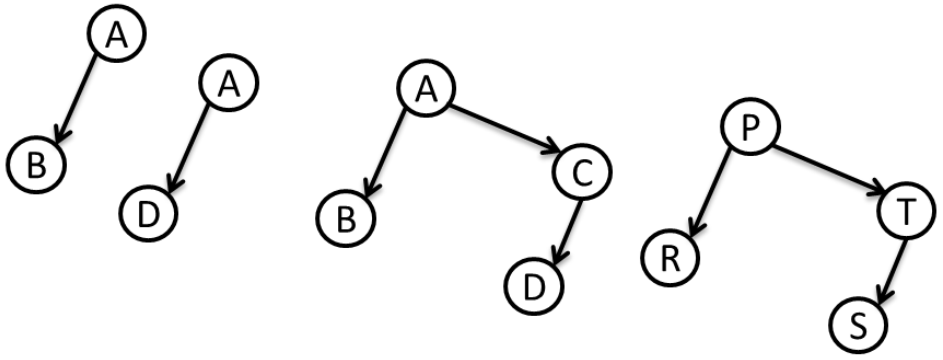


c) Árbol genealógico binario

3. Árboles Binarios Distintos, Similares y Equivalentes

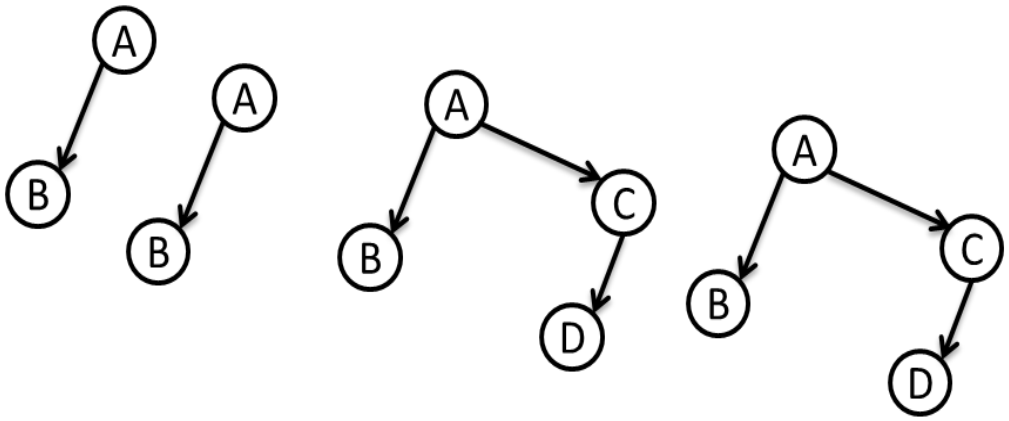


Dos árboles binarios son distintos cuando sus estructuras son diferentes



Dos árboles binarios son **similares** cuando sus estructuras son idénticas, pero la información que contienen sus nudos defiere entre sí.

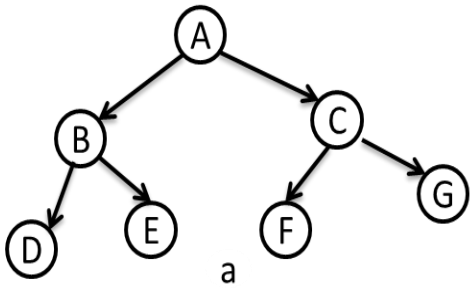
Por último, los árboles binarios **equivalentes** se definen como aquellos que son similares y además los nodos contienen la misma información.



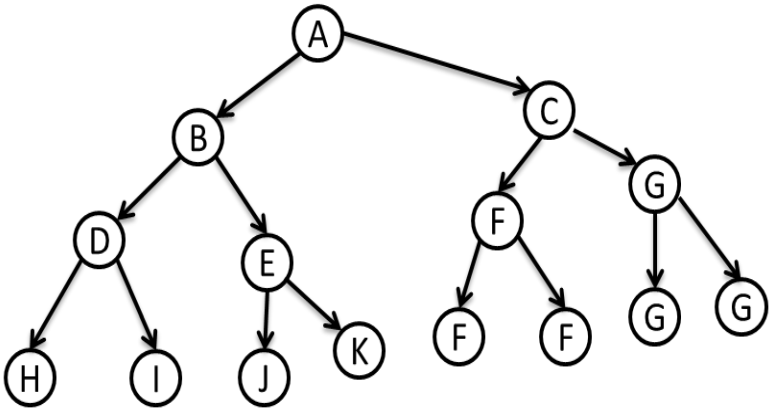
4. Árboles Binarios Completos

Se define un árbol binario **completo** con un árbol en el que todos sus nodos, excepto los del último

Nivel, tienen dos hijos; el subárbol izquierdo y el subárbol derecho. Ejemplo:



a



b

Árboles binarios completos. a) De altura 3 y b) de altura 4 se puede calcular el número de nodos de un árbol binario completo de altura h, aplicando la siguiente formula:

$$\text{NÚMERO DE NODOS}_{ABC} = 2^h - 1$$

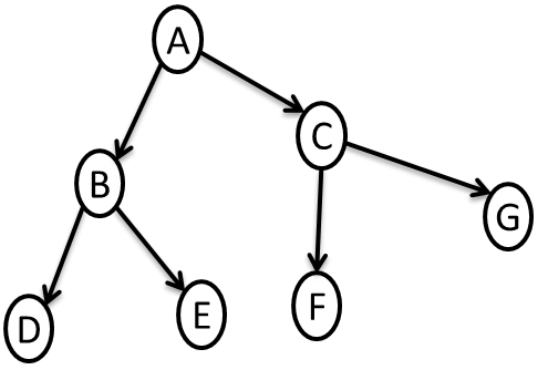
Donde ABC significa árbol binario completo, y h la altura del árbol. Así, por ejemplo, un árbol binario completo de altura 5 tendrá 31 nodos, de altura 9 tendrá 511 nodos y un árbol de altura 17 tendrá 131 071 nodos. Algunos autores definen un árbol binario completo de otra forma; y otros que utilizan el término lleno para referirse a completo.

5. Recorrido En Árboles Binarios

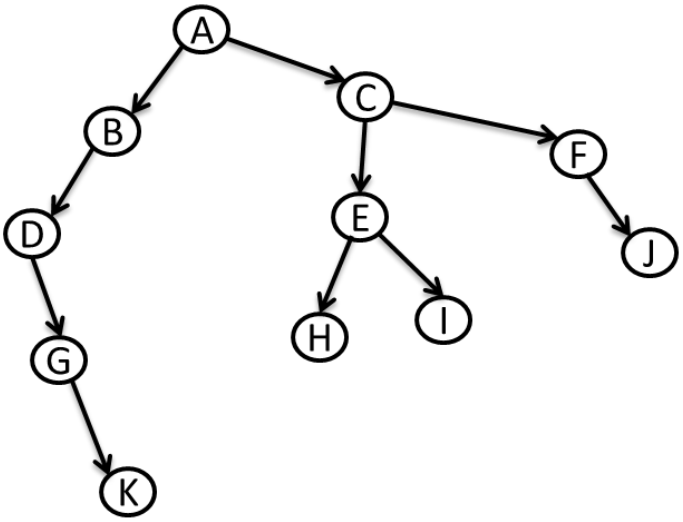
Una de las operaciones más importantes al a realizar en un árbol binario es el recorrido de los mismos. Recorrerlo significa visitar los nodos del árbol en forma sistemática, de tal forma que todos los nodos del mismo sean visitados una sola vez. Existen tres formas diferentes para realizar dicho recorrido.

a. Recorrido en Preorden

- Visitar la raíz
- Recorrer el subárbol izquierdo
- Recorrer el subárbol derecho



PreOrden: A B D E C F G
InOrden: D B E A F C B
PostOrden: D E B F G C A



PreOrden: A B D G K C E H I F J
InOrden: G K D B A H E I C F J
PostOrden: K G D B H I E J F C A

Recorrido en inorden

- Recorrer el subárbol izquierdo
- Visitar la raíz
- Recorrer el subárbol derecho

Recorrido en PostOrden

- Recorrer el subárbol izquierdo
- Recorrer el subárbol derecho
- Visitar la raíz

6. Algoritmos Correspondientes

PREORDEN (NODO)
"El algoritmo realiza el recorrido preorden en árbol binario. NODO es uno de los datos tipo puntero"
INFO, IZQ Y DER son campos del registro. NODO es una variable de tipo carácter. IZQ y DER son variables de tipo puntero.

1. Si NODO#NIL entonces
2. Escriba el NODO {escribir NODO^.INFO}
3. Regresa a PREORDEN con NODO^.IZQ
4. Llamada recursiva a PREORDEN con la rama izquierda del nodo en cuestión
5. Regresa a PREORDEN con NODO^.DER
6. Llamada recursiva a PREORDEN con la rama derecha del nodo en cuestión
7. Finsi

INFORDEN(NODO)
"El algoritmo realiza el recorrido inorden en árbol binario. NODO es un registro de tipo puntero"
INFO, IZQ Y DER son campos del registro. INFO es una variable de tipo carácter. IZQ y DER son variables de tipo puntero.

1. Si NODO#NIL entonces
2. Regresa a INFORDEN con NODO^.IZQ
3. Llamada recursiva a INORDEN con la rama izquierda del nodo en cuestión
4. Escribir el NODO NODO^.INFO
5. Regresa a INFORDEN con NODO^.DER
6. Llamada recursiva a INFORDEN con la rama derecha del nodo en cuestión
7. Finsi

INFORDEN(NODO)
"El algoritmo realiza el recorrido PostOrden en árbol binario. NODO es un dato de tipo puntero" INFO, IZQ Y DER son campos del registro nodo. INFO es una variable de tipo carácter. IZQ y DER son variables de tipo puntero.

1. Si NODO#NIL entonces
2. Regresa a POSTORDEN con NODO^.IZQ
3. Llamada recursiva a POSTORDEN con la rama izquierda del nodo en cuestión
4. Regresa a POSTORDEN con NODO^.DER
5. Llamada recursiva a POSTORDEN con la rama derecha del nodo en cuestión
6. Escribir el NODO^.INFO
7. Finsi

Carga de un Nodo
El algoritmo carga los nodos de un árbol binario en memoria. NODO es una variable de tipo puntero. La primera vez NODO es creado en el programa principal.

Además de las variables ya utilizadas utilizaremos INFO de tipo carácter y OTRO de tipo puntero.

1. leer información (INFO)
2. Hacer NODO^.INFO =INFO
3. Escribir "existe nodo por izquierda?"
4. Leer respuesta
5. Si respuesta es afirmativa Entonces
6. CREA(OTRO) crear un nuevo nodo.
7. Hacer NODO^.IZQ =OTRO
8. Regresar a CARGA con NODO^.IZQ
9. Llamada recursiva
10. SiNo
11. Hacer NODO^.IZQ =NIL
12. Finsi
13. Escribir "existe nodo por derecha?"
14. Leer respuesta
15. Si respuesta es afirmativa Entonces
16. CREA(OTRO) crear un nuevo nodo.
17. Hacer NODO^.DER =OTRO
18. Regresar a CARGA con NODO^.DER
19. Llamada recursiva
20. SiNo

21. Hacer NODO^.DER =NIL
22. FinSi

7. Árboles Binarios de Búsqueda

Es una estructura sobre la cual se pueden efectuar eficientemente operaciones de búsqueda, inserción y borrado.

BÚSQUEDA (NODO, INFO)
El algoritmo localizará un nodo en un árbol binario de búsqueda. NODO es una variable de tipo puntero que apunta a la raíz del árbol. INFOR es una variable de tipo entero que contiene la información a localizar en el árbol. Cabe aclarar que la primera vez la variable NODO no puede ser vacía.

1. Si INFO<NODO^.INFO entonces
2. Si NODO^.IZQ=NIL entonces
3. Escribir “el nodo no se encuentra en el árbol”
4. SiNo
5. Regrese a BÚSQUEDA con NODO^.IZQ e INFO
6. Llamada recursiva
7. FinSi
8. SiNo
9. Si INFO>NODO^.INFO entonces
10. Si NODO^.DER=NIL entonces
11. Escribir “el nodo no se encuentra en el árbol”
12. SiNo
13. Regrese a BÚSQUEDA con NODO^.DER e INFO
14. Llamada recursiva
15. FinSi
16. SiNo
17. Escribir “el nodo se encuentra en el árbol”
18. FinSi
19. FinSi

8. Inserción de un árbol binario de búsqueda

La estructura corre conforme se insertan elementos de árbol. Los pasos a seguir son:

Debe compararse la clave a insertar con la raíz del árbol de búsqueda., si es mayor debe avanzarse hacia el subárbol derecho, si es menor hacia el subárbol izquierdo.

Repetir sucesivamente el paso 1 hasta que se cumpla alguna de estas condiciones.

El subárbol derecho es igual a vacío, o el subárbol izquierdo es igual a vacío, en cuyo caso se inserta el elemento en el lugar respectivo.

La clave que quiere insertarse es igual a la raíz del árbol, en cuyo caso no se realizará la inserción.

INSERCIÓN (NODO, INFOR)
El algoritmo realiza la inserción de un nodo en un árbol binario de búsqueda. NODO es una variable de tipo puntero y la primera vez debe ser distinta de vacío. INFOR es una variable de tipo entero que contiene la información del elemento a insertar.se utiliza además la variable auxiliar OTRO de tipo puntero.

1. Si INFO<NODO^.INFO entonces
2. Si NODO^.IZQ=NIL entonces
3. CREA (OTRO) crear un nuevo nodo.
4. Hacer OTRO^.IZQ =NIL
5. OTRO^.DER=NIL, OTRO^.INFO=INFOR, NODO^.IZQ=OTRO
6. SiNo
7. Regresa a INSERCIÓN con NODO^.IZQ e INFOR
8. Llamada recursiva
9. FinSi
10. SiNo
11. Si INFO>NODO^.INFO entonces
12. Si NODO^.DER=NIL entonces
13. CREA (OTRO) crear un nuevo nodo.
14. Hacer OTRO^.IZQ =NIL
15. OTRO^.DER=NIL, OTRO^.INFO=INFOR, NODO^.DER=OTRO
16. SiNo
17. Regresa a INSERCIÓN con NODO^.DER e INFOR
18. Llamada recursiva
19. FinSi
20. Escribir “El nodo ya se encuentra en el árbol”
21. FinSi
22. FinSi

9. Borrado de un árbol binario de búsqueda
Consiste en eliminar un nodo del árbol sin violar los principios que definen justamente un árbol binario de búsqueda. Deben tenerse en cuenta los casos siguientes:

Si el elemento a borrar es terminal u hoja simplemente se suprime.
Si el elemento a borrar tiene un solo descendiente, entonces tiene que sustituirse por ese descendente.

Si el elemento a borrar tiene los dos descendientes, entonces se tiene que sustituir por el nodo de más a la izquierda en el subárbol derecho o por el nodo de más a la derecha en el subárbol izquierdo.

Antes de eliminar un nodo, debe localizarse en el árbol. Para esto se usa el algoritmo de búsqueda.

ELIMINAR (NODO, INFOR)
El algoritmo realiza la Eliminación de un nodo en un árbol binario de búsqueda. NODO es una variable de tipo puntero por referencia. INFOR es una variable de tipo entero que contiene la información del elemento a eliminar, se utilizan además las variables auxiliares AUX, AUX1 y OTRO de tipo puntero.

1. Si INFO#NIL entonces
2. Si INFOR<NODO^.INFO entonces
3. Regresa a ELIMINAR con NODO^.IZQ e INFOR
4. Llamada recursiva
5. SiNo
6. Si INFOR>NODO^.INFO entonces
7. Regresa a ELIMINAR con NODO^.DER e INFOR
8. Llamada recursiva
9. SiNo
10. Hacer OTRO=NODO
11. Si OTRO^.DER=NIL entonces
12. Hacer NODO=OTRO^.IZQ
13. SiNo
14. Si OTRO^.IZQ=NIL entonces
15. Hacer NODO=OTRO^.DER
16. SiNo
17. Hacer AUX=OTRO^.IZQ, AUX1=AUX
18. Repetir mientras AUX^.DER#NIL
19. Hacer AUX1=AUX, AUX1=AUX^.DER
20. FinRepetir
21. Hacer OTRO^.INFO=AUX^.INFO,
22. OTRO=AUX, AUX1^.DER=AUX^.IZQ
23. FinSi
24. FinSi

25. FinSi
26. FinSi
27. Quita(OTRO)
28. Libera memoria del nodo
29. SiNo
30. Imprima “el Nodo no se encuentra en el árbol”
31. FinSi

Ejercicio:
En un trabajo en grupo, simúlese una estructura de oficina de desarrollo de software, donde cada grupo lleve a cabo una tarea particular dentro de la jerarquización de la información en árboles binarios. Las obligaciones serían.

1. Analista de la información con miras al planteamiento en estructura árbol, necesidades y posibilidades de manejo jerarquizado de la información.
2. Diseño de la estructura jerarquizada con la representación más apropiada y optima dado el tipo de información
3. Planteamiento de los requerimientos funcionales con los cuales debe cumplir el sistema.(informes, búsquedas, inserciones, borrado, modificación)
4. Algoritmo que desarrolle las lógicas efectivas para llevar a cabo cada requerimiento.
5. Codificación en un lenguaje simulador o comercial de los algoritmos generados.

Al final deberá tenerse un producto terminado, competente, óptimo y funcional según las necesidades planteadas y deberá valorarse el tipo de participación de cada uno de los involucrados,

Información complementaria
Puedes encontrar información de bastante ayuda en su comprensión temática y lógica de procesos para el manejo de los árboles, en los siguientes link, accedidas en sept. 2011

http://www.youtube.com/watch?v=1s0vIXsx5Pg&feature=relmfu
Estructuras en árbol
http://www.youtube.com/watch?v=pFaQMXz7HCo&feature=related
Segmentos en Postorden
http://www.youtube.com/watch?v=Vqyh1q95rlU&feature=related
Recorrido por los nodos de un árbol
http://www.youtube.com/watch?v=VVhIZjCcArg&feature=related
Arboles binarios en el salón
http://www.youtube.com/watch?v=wfUhjpo0tMM

4.4. Pistas de Aprendizaje

Tenga en cuenta: para un mejor aprovechamiento y comprensión de las estructuras de datos, es preferible haber cursado ya la asignatura de fundamentos de programación donde se espera haya obtenido destreza en la construcción de algoritmos, manejo de estructuras lógicas y repetitivas.

No olvide: Algunos ejemplos de algoritmo presentados en éste módulo son referidos a subprogramas invocados desde un programa principal, por ello simulan la emisión y recepción de parámetros y omiten la inicialización de lagunas variables por extensión.

Asegúrese de: Ser muy ordenado en su estructura en la escritura código bien sea en algoritmos o lenguaje de programación, una vez que las sentencias repetitivas y de decisión lógica suelen de cuidado al momento de quedar mal apareadas, y pueden darnos dolores de cabeza en una prueba de escritorio o una ejecución.

Utilice: La documentación de las instrucciones nuevas ya sea en el algoritmo o código, una vez que esto evita que a la hora de buscar un error de lógica o codificación sea mucho más fácil de hacerle seguimiento

Acostumbre: Realizar pruebas de escritorio con las variables y componentes que sea necesario a fin de determinar si la forma lógica de planteamiento de algoritmo si es eficiente y apropiada.

Indagación y Práctica: El módulo únicamente plantea la lógica y forma de manejo de las estructuras de almacenamiento de datos en memoria, pero el estudiante para apropiarse de los concepto y su aplicación, deberá valerse de mucha práctica, ejercicios de aplicación y pruebas lógicas que le permitan desarrollas las destrezas que demanda el desarrollo de software a nivel comercial. En internet es posible encontrar algoritmos ya desarrollados y diversas formas de ilustración grafica de este tipo de componentes.

Graficar: Cuando se trabaja con formas abstractas como son las estructuras de almacenamiento en memoria, lo mejor es ilustrar gráficamente los elementos usados con el fin de comprender mejor la lógica de los procesos que se suceden en ellos.

Contextualizar: Trate de llevar cada estructura de almacenamiento trabajada a situaciones y contextos reales y cotidianos que permitan evidenciar su aplicación y funcionalidad directa, de esta forma será mucho más fácil su asimilación y apropiación.

4.5. Glosario

Arreglo Unidimensional o Vector: Términos utilizados para referirse a una secuencia de posiciones en memoria usadas para almacenar datos mediante un acceso directo a cada posición, elementos que constan de una sola fila y múltiples columnas.

Matrices o Arreglos Bidimensionales: Son estructuras de dos dimensiones, constituidas por filas y columnas, conformando posiciones determinadas por la coordenada entre una fila y una columna, en la cual se almacenan datos.

Arreglos Tridimensionales: Son estructuras conformadas por secuencia de matrices estilo lista de planillas, cada matriz definida por filas y columnas número identificados de matriz.

Búsqueda Binaria: Procedimiento por medio del cual se localiza un dato en un Arreglo o vector, descartándose por tramos de éste según comparación de su primer elemento con el dato buscado, implica una lista de elementos ordenados.

Ordenamiento por Burbuja: Proceso por medio del cual se ordenan los elementos de un arreglo o vector, mediante la comparación repetitiva de cada par de datos adyacentes.

Arreglo de Registros: Estructura en la cual un arreglo o Vector conforma cada una de sus posiciones con varios datos del mismo tipo y un nombre apuntador o de índice.

Estructura de Listas: **C**olección de elementos llamados generalmente nodos, que son representaciones en memoria de una variable o posición a la cual aparte del dato se le adiciona una dirección o apuntador a otro u otros nodos.

Estructura de Pilas: Al igual que las listas son un conjunto de nodos en un arreglo de lista que permite acceder a ellos en orden inverso a su ingreso.

LIFO: Iniciales de Last input First Output, es decir hace referencia el proceso mediante el cual los últimos elementos que ingresan son los primeros que salen.

FIFO: Iniciales de First input First Output, es decir hace referencia el proceso mediante el cual los primeros elementos que ingresan son los primeros que salen.
Estructura de Cola: Se refiere a una serie de elementos almacenados con características funcionales de colas, haciendo referencia el método FIFO.

4.6. Bibliografía

Villalobos A. Jorge; Casallas Rubby. Fundamentos de programación, aprendizaje activo basado en casos, Sevilla 1996, ISBN: 970-10-3319-4

Flores Rueda Roberto. Algoritmos, estructuras de datos y programación orientada a objetos. Pamplona, 1998, ISBN: 870-12-43197-3

Cairo /Guradath, M, c, EstructuraS de Datos, Mexico,1996 ISBN: 970-10-0258-X

Loomis, E.S. Mary, Estructura de Datos y Organizaciones, México, 1995,
ISBN: 968-880-190-9

Pozo Coranado, Salvador, Estructuras Dinámicas de Datos, Mexico, 2003. ISBN 968-4845-200-8

4.7. Fuentes digitales o electrónicas

Algoritmos:http://www.geocities.com/David_ees/Algoritmia/indice.htm, Accedida en Septiembre de 2011
El poder de lo simple: Http//riosur.net/, Accedida el 20 de Septiembre de 2011
Con Clase: http://www.conclase.net, accedida el 2 de octubre de 20011
Estructuras en árbol
http://www.youtube.com/watch?v=pFaQMXz7HCo&feature=related, accedida en sep. 12 2011