UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO

FACULTAD DE INGENIERÍA

INGENIERÍA EN COMPUTACIÓN

UNIDAD DE APRENDIZAJE:
ESTRUCTURAS DE DATOS

APUNTES

ELABORADO POR:
M. EN I. MIREYA SALGADO GALLEGOS
ING. BEATRIZ OROZCO GARDUÑO

ENERO 2018

ÍNDICE

Presentación........ 3

UNIDAD DE COMPETENCIA I: Reconocer y manejar las variables dinámicas........ 6

1.1 Estructuras de datos........ 6

1.2 Abstracción, estructuras de datos y representación........ 8

1.3 Variables dinámicas: apuntadores, operaciones básicas........ 9

Referencias........ 14

UNIDAD DE COMPETENCIA II :Aplicar las principales estructuras de datos lineales........ 15

2.1 Pilas: representación, operaciones (inserción, eliminación, pila llena, pila vacía), aplicaciones........ 15

2.2 Colas: representación, operaciones (inserción, eliminación, cola llena, cola vacía), aplicaciones........ 23

2.3 Cola circular........ 31

2.4 Listas: representación, operaciones (inserción, eliminación, recorrido, búsqueda), listas simplemente y doblemente ligadas, lista circular, lista doble, lista doble circular, aplicaciones........ 36

Referencias........ 78

UNIDAD DE COMPETENCIA III: Aplicar la estructura de datos árbol........ 79

3.1 Recursividad directa........ 79

3.2 Árboles: usos, características, representación y construcción........ 82

3.3 Árboles binarios (representación. recorrido en preorden, inorden, postorden)........ 84

3.4 Árboles binarios de búsqueda (operaciones – inserción, eliminación, búsqueda)........ 93

Referencias........ 100

UNIDAD DE COMPETENCIA IV: Aplicar la estructura de datos grafo........ 101

4.1 Grafos: características y clasificación........ 101

4.2 Grafos dirigidos........ 105

4.3 Grafos no dirigidos........ 111

4.4 TAD grafo (estático y dinámico)........ 116

4.5 Recorridos de un grafo........ 126

4.6 Algoritmos........ 131

Referencias........ 150

Referencias generales........ 151

Actividades........ 152

PRESENTACIÓN

El programa de Estructuras de Datos tiene por objetivo *que el alumno identifique las herramientas teóricas fundamentales para la representación y manipulación de información en la computadora, haciendo énfasis en el tipo de datos dinámicos;* con base en éste, el programa está conformado en la actualidad de 4 unidades de competencia en las cuales están basados estos apuntes y son:

1. Reconocer y manejar las variables dinámicas
2. Aplicar las principales estructuras de datos lineales
3. Aplicar la estructura de datos árbol
4. Aplicar la estructura de datos grafo

Este material está enfocado en apoyar a los alumnos en el reforzamiento de los conocimientos adquiridos en el aula y transmitidos por el profesor.

Está constituido por 4 apartados que corresponden a cada una de las unidades de competencia antes mencionadas, con la finalidad de que al alumno le sirva de guía siguiendo una continuidad del programa.

En la unidad 1 *"Reconocer y manejar las variables dinámicas"* se abordan los temas conceptos básicos de programación, apuntadores y los tipos de datos abstractos con la finalidad de presentar un contexto introductorio a los temas de estructuras dinámicas. Aunque en esta unidad no se contemplan temas básicos de programación, se incluyen actividades elementales para reforzar estos conocimientos previos.

La unidad 2 "*Aplicar las principales estructuras de datos lineales*" aborda los temas relacionados con las principales estructuras de datos dinámicas como son pila y cola en su implementación estática y dinámica, las listas simples y doblemente enlazadas, así como las listas circulares simple y doblemente enlazadas. Una de las aplicaciones de la pila dinámica es la evaluación de expresiones, tema que también está incluido en esta unidad.

"Aplicar la estructura de datos árbol" que es la unidad 3 de este material y del programa, contempla los temas relacionados a la estructura de datos dinámica árbol, considerando los tipos principales de árboles como son los binarios, los de búsqueda, los balanceados y los de expresión, de la misma manera que aborda los recorridos y las operaciones para la implementación de un árbol binario.

Finalmente la unidad 4 *"Aplicar la estructura de datos grafo"* integra los temas relacionados a la estructura de datos Grafo, considerando sus 2 tipos principales: dirigidos y no dirigidos, tanto en sus implementaciones estática (matriz de adyacencia) y dinámica (lista de adyacencia), ambos con factor de peso. Así como también los algoritmos del camino más corto, costo mínimo y árbol abarcador de costo mínimo.

Dentro de cada unidad se presentan las bases teóricas de cada tema así como ejemplos para reforzar los conocimientos adquiridos y la bibliografía en la que se pueden basar para ampliar sus conocimientos.

Este material consta de una sección de actividades las cuales se deben de ir resolviendo conforme se van abordando los contenidos propios de la unidad de competencia. Estas actividades incluyen tanto ejercicios como programas a elaborar en cualquier lenguaje de programación

Cabe mencionar que en algunas actividades se especifica que se realicen en lenguaje C simplemente por recomendar aunque no es forzoso ya que el dominio de un lenguaje de programación no está incluido en el objetivo de la unidad de aprendizaje y pueden resolverlo en cualquier lenguaje que el alumno domine.

Los conocimientos previos requeridos para este material son los temas de la unidad de aprendizaje de Programación Estructurada que incluyen: programación estructurada, programación modular, manejo de vectores, matrices y registros.

Este material está orientado a alumnos que cursen la materia de estructuras de datos principalmente, aunque también a alumnos de programación avanzada y para todos aquellos interesados en programar empleando estructuras de datos dinámicas.

ESTRUCTURAS DE DATOS

UNIDAD DE COMPETENCIA I
RECONOCER Y MANEJAR LAS VARIABLES DINÁMICAS

10 HRS.
2 SEMANAS

1.1 ESTRUCTURAS DE DATOS

Un **programa** es una secuencia de instrucciones mediante las cuales se ejecutan diferentes acciones de acuerdo con los datos que se estén procesando. Es un conjunto de instrucciones que sigue la computadora para alcanzar un resultado específico concepto desarrollado por Von Neumann [1]. El programa debe incluir instrucciones para las acciones que deban ejecutarse sobre cada uno de los tipos de datos admitidos. Un programa se compone de estructuras de datos, operaciones primitivas elementales y estructuras de control, como se muestra a continuación:

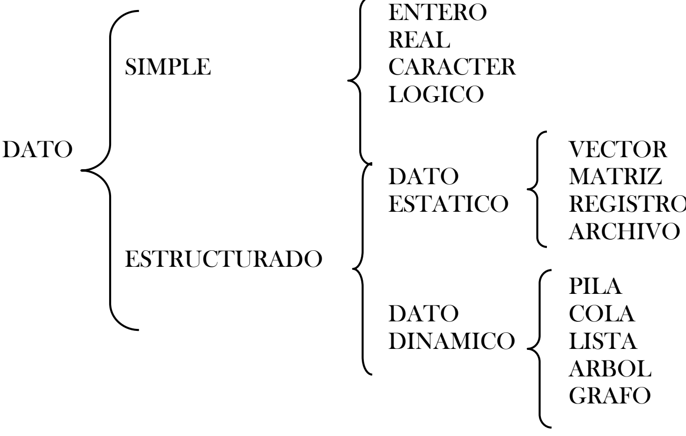
Programa [1] = estructuras de datos
+ operaciones primitivas elementales
+ estructuras de control

Estructuras de datos: Los hechos reales, representados en forma de datos, pueden estar organizados de diferentes maneras llamadas estructuras de datos. Por ejemplo el nombre, las horas trabajadas y el sueldo por hora son los datos mediante los cuales se representa un empleado en una situación de nómina (pago de sueldos).

Dato: Es una expresión general que describe los objetos con los cuales opera una computadora. Se refiere a la representación de algún hecho, concepto o entidad real (pueden tomar diferentes formas, por ejemplo palabras, números o dibujos).

Las **estructuras de datos** son las diversas maneras de representar un objeto en la computadora; es decir, la forma en que se organizan los datos para ser manipulados en la computadora. Estos datos pueden ser constantes o variables:

Los tipos de datos más comunes son:



[1] Basado en la idea original de Niklaus Wirth, en su libro Algoritmos + Estructuras de datos = Programas, 1986

DATOS SIMPLES: Primitivos.

- **ENTEROS**: Son un subconjunto finito se los números enteros. No tienen componentes fraccionarios. Pueden ser positivos o negativos.
- **REALES**: Son un subconjunto finito de los números reales. Siempre tienen un punto decimal. Pueden ser positivos o negativos. Entero y parte decimal.
- **LÓGICOS**: Son aquellos que sólo pueden tomar uno de dos valores: verdadero o falso.
- **CARÁCTER Y CADENA**: Son un conjunto finito y ordenado de caracteres que la computadora reconoce. Un dato tipo carácter sólo tiene un carácter (Alfanumérico y Símbolos) y el tipo de dato cadena es un conjunto de caracteres.

DATOS COMPUESTOS O ESTRUCTURADOS: Están basados en tipos de datos primitivos. Se dividen en datos estáticos y datos dinámicos.

DATOS ESTÁTICOS: Son aquellos en los que el tamaño ocupado en memoria se define antes que el programa se ejecute y no puede modificarse durante la ejecución del programa.

DATOS DINÁMICOS: Son aquellos en los que el tamaño ocupado en memoria NO se define antes que el programa se ejecute, es decir, éste varía durante la ejecución del programa y aumenta o disminuye su tamaño de acuerdo a las necesidades de ejecución del programa.

1.2 ABSTRACCIÓN, ESTRUCTURAS DE DATOS Y REPRESENTACIÓN.

La abstracción de datos es la técnica de inventar nuevos tipos de datos que sean más adecuados a una aplicación permitiendo así diseñar programas más cortos, legibles y flexibles [2, 3].

Los lenguajes de programación soportan diferentes tipos de datos, de los cuales más conocidos entero, real, lógico, carácter, etc. en sus diferentes codificaciones; sin embargo, algunos lenguajes de programación permiten al programador que éste defina sus propios tipos de datos con base en sus necesidades de almacenamiento y manipulación de información. Estos tipos de datos se conocen como *tipo de dato abstracto* ***TAD*** *(Abstract Data Type,* ***ADT****).*

El **TAD** es el conjunto de valores que pueden tomar los datos de ese tipo y las operaciones que los manipulan, de forma que sólo debe acceder a los valores de los datos empleando las operaciones abstractas definidas sobre ellos [2]. El **TAD** es un tipo definido por el programador que:

- Tiene un conjunto de valores y un conjunto de operaciones que se aplican a esos valores.
- Se puede manejar sin conocer la representación interna.

Un *tipo de dato abstracto* consta de:

- La representación: elección de las estructuras de datos.
- Las operaciones: elección de los algoritmos.

Entonces se puede decir que:

TAD = Representación (datos) + Operaciones (funciones y procedimientos)

De la misma manera se dice que un TAD es un modelo (estructura) con un número de operaciones que afectan a ese modelo.

Tipos básicos de operaciones en un TDA

- **Constructores**: Crean una nueva instancia del tipo.
- **Transformación**: Cambian el valor de uno o más elementos de una instancia del tipo.
- **Observación**: Permiten observar el valor de uno o varios elementos de una instancia sin modificarlos.
- **Iteradores**: Permiten procesar todos los componentes en un TDA de forma secuencial.

1.3 VARIABLES DINÁMICAS: APUNTADORES. OPERACIONES BÁSICAS.

Las estructuras de datos dinámicas son una colección de elementos (denominados nodos) que son nodos que se amplían (expanden) o reducen (contraen) a medida que se requiera durante la ejecución de un programa cambiando sus posiciones de memoria [4]

Una estructura de datos dinámica, como ya se mencionó, es una colección de elementos llamados nodos que se enlazan o encadenan juntos. Este enlace se hace con una variable llamada puntero (apuntador) que apunta a la dirección del nodo siguiente [4].

Un apuntador es una variable que contiene la dirección en memoria de otra variable. Se pueden tener apuntadores a cualquier tipo de variable. Por lo tanto, los apuntadores son variables que almacenan direcciones de memoria.

- En general una variable contiene un valor específico dependiendo de cómo fue declarada.

- Un apuntador contiene la dirección de una variable que contiene un valor específico.

Dirección	Contenido
00D0	
00D1	5
00D2	
00D3	
00D4	6
00D5	

1.3.1 Declaración de apuntadores

Los apuntadores como cualquier otra variable deben de ser declarados antes de que puedan ser utilizados. La sintaxis general de la declaración de un apuntador en lenguaje C es:

*Tipo *nom_var;*

Por ejemplo:

```
int *num;
float *cal;
```

y en lenguaje Pascal es:

```
Var
        Nom_var: ^tipo
        P: ^integer
```

Cabe señalar que siempre que una variable (de tipo no apuntador) es definida a ésta ya es asignada una dirección de memoria, sin embargo lo que realmente se trabaja con ella es el contenido y no la dirección. En contraparte con una variable tipo apuntador, ésta desde su declaración se trabaja principalmente con su dirección.

Lo anterior no implica que no se pueda trabajar con contenido y valores indistintamente de si es una variable tipo apuntador o no. Para utilizar esto se hace uso de dos operadores que permiten el acceso a la dirección o al contenido de una variable tipo apuntador o no.

Con lo anterior se puede decir que cuando una variable se declara, se asocian tres atributos fundamentales: su ***nombre,*** su ***tipo*** y su ***dirección*** en memoria [2].

1.3.2 Contenido y Dirección

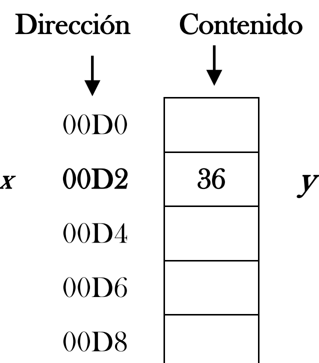
- El operador unario o monádico **&** devuelve la dirección de memoria de una variable.
- El operador de dirección o de referencia ***** devuelve el “contenido de un objeto apuntado por un apuntador".

Considerando el ejemplo siguiente

```
void main(){
    int *x, y;

    y = 36;
    x = &y;
}
```

Y suponiendo que el compilador le asigne a ***y*** en el momento de su declaración la dirección de **00D2**, esquemáticamente se puede ver lo siguiente



Es preciso diferenciar entre las dos entidades implicadas en el apuntamiento: la *variable puntero (*quien hace el apuntamiento) y la *variable apuntada* (a quien se apunta).

Considerando el ejemplo anterior, ***x*** es la variable puntero y ***y*** es la variable a quien se le apunta.

1.3.3 Aritmética con apuntadores

Parte del poder de los apuntadores viene de la habilidad de realizar operaciones matemáticas sobre los mismos apuntadores. Los apuntadores pueden ser incrementados, decrementados y manipulados usando expresiones matemáticas.

La aritmética de punteros se limita a suma, resta, comparación y asignación. Las operaciones aritméticas en los punteros de tipoX (punteros-a-tipoX) tienen automáticamente en cuenta el tamaño real de tipoX. Es decir, el número de bytes necesario para almacenar un objeto tipoX.

Considerando que *ptr1, ptr2* sean punteros a objetos del mismo tipo y *n* un tipo entero, las operaciones permitidas y los resultados obtenidos con ellas en lenguaje C son:

Operación	Resultado	Comentario
pt1++	puntero	Desplazamiento ascendente de 1 elemento
pt1--	puntero	Desplazamiento descendente de 1 elemento
pt1 + n	puntero	Desplazamiento ascendente n elementos
pt1 - n	puntero	Desplazamiento descendente n elementos
pt1 - pt2	entero	Distancia entre elementos

1.3.4 Operaciones Básicas

Resumiendo, en la siguiente tabla se muestran las operaciones básicas con las que se trabajan las variables tipo apuntador.

Op	Función	Ejemplo	Explicación
(void *)	Convierte entero a dirección	(void *)0	dirección nula
(tipo *)	Convierte apuntador a tipo	void *p; *(int *) p = 1;	Se declara puntero de ningún tipo de dato y puede ser transformado a un cierto tipo, como este ejemplo a entero.
*	Para declarar apuntadores	int *p;	p es un apuntador
&	Obtener dirección.	q=&i;	q apunta a i
=	Asignar dirección	p=q=&i;	p y q apuntan a i
*	Operador indirección Referencia al contenido de un apuntador	*p=7;	a donde apunta p almacena un 7
++	Incremento-dato	++*p;	incrementa el valor que apunta p
++	Incremento-apuntador	*p++; p++;	incrementa apuntador p
%p	Especificador tipo apuntador	printf("%p",p);	imprime dirección almacenada en p
==	igualdad entre apuntadores	p==q	regresa falso si no son iguales
!=	desigualdad entre apuntadores	p!=q	regresa falso si son iguales
<	menor, <= menor o igual	p<=q	regresa falso si p es mayor que q
>	mayor, >= mayor o igual	p>=q	regresa falso si p es menor que q
sizeof	Espacio que ocupa un dato en bytes	sizeof(void *)	tamaño de cualquier apuntador

Actividades 10-11

1.3.5 Apuntadores con registros

Así como se pueden definir apuntadores de tipos de datos simples (entero, real, etc.) se pueden también definir punteros a registros, el manejo de éstos es el mismo, sólo se diferencía en la referencia a los elementos del registro, esto se puede apreciar en el siguiente ejemplo:

Se define un registro denominado *Fecha* con los campos *mes, día y anio* de tipo entero:

Fecha:REGISTRO
mes: entero
día: entero
anio: entero
Fin REGISTRO

Las variables a utilizar de tipo *Fecha* son:

Fec : Fecha */*es una variable simple de tipo Fecha*/*
***AFec : Fecha** */*es una variable apuntador de tipo Fecha*/*

Con base en lo anterior, la referencia a elementos para cada una de las variables definidas es:

Fec	*AFec
Fec.mes Fec.día Fec.anio	Fec->mes Fec->día Fec->anio

De esta manera se puede decir que cuando se hace referencia a los campos de un registro mediante **->** se está hablando de un apuntador a los campos de ese registro.

1.3.6 Paso de parámetros por valor y por referencia

El flujo de información entre programas y subprogramas o módulos se realiza con el paso de parámetros; a través de éstos, el programa principal puede pasar valores para que sean usados en otros procesos y subprogramas [5]; este paso de parámetros se puede realizar mediante dos casos: por *valor* y por *referencia.*

- **Paso por valor:** se utiliza para suministrar datos de entrada a un módulo desde otro programa o subprograma que lo llama, de modo que el valor del parámetro real o actual se copie en el parámetro formal correspondiente. El subprograma que lo recibe, trabaja con una copia del parámetro en que se recibe, lo cual implica que el parámetro real no se modifica, sólo varía la copia del parámetro.

MÓDULO QUE ENVÍA

Programa Principal()
Inicio
x,y: E
x← 10
y← 20
*Suma(**x,y**)*
Termina

MÓDULO QUE RECIBE

*Módulo Suma(**a:E, b:E**)*
Inicio
S: E
S← a + b
Escribir("La suma es: ",S)
Termina

- **Paso por referencia:** se utiliza indistintamente para suministrar datos de entrada y recibir datos de salida. El programa que llama al módulo manda la referencia de memoria del parámetro actual al parámetro formal; el módulo que lo recibe lo puede modificar a su conveniencia. Para declarar un parámetro por referencia se utiliza la palabra **ref**. Es el lenguaje de programación quien determinará la sintaxis correspondiente, por ejemplo en lenguaje C es con un **&.**

A continuación se presenta un ejemplo.

MÓDULO QUE ENVÍA

```
Programa Principal()
Inicio
        x,y: E
        x← 10
        y← 20
        Suma(&x,&y)
        Escribir("La suma es: ",x)
Termina
```

MÓDULO QUE RECIBE

```
Módulo Suma(*a:E, *b:E)
Inicio
        a ← a + b
Termina
```

En la instrucción *Suma(&**x,&y**)* se envía la dirección de cada una de las variables *x y y* que son recibidas por el módulo *Suma,* estos valores son afectados directamente en la localidad de memoria de *a* por lo que al regresar al módulo *Principal* el valor de *x* ya fue afectado.

Para más claridad al respecto realizar la **actividad 12** y encontrar las diferencias.

Actividad 12

Resumen

- Existen 2 tipos de datos: simples y estructurados; a su vez los datos estructurados se dividen en dinámicos y estáticos.
- Los datos dinámicos se caracterizan por la asignación y uso de localidades de memoria de la computadora.
- TAD es un tipo de dato abstracto el cual consta de nombre, estructuras de datos y operaciones específicas.
- Un apuntador es una variable que contiene la dirección de memoria de otra variable.
- Cuando una variable se declara, se asocian tres atributos fundamentales: su *nombre,* su *tipo* y su *dirección* en memoria.
- Existen dos formas de enviar datos entre dos módulos de un programa: *Paso por valor y paso por referencia.*
- Paso por valor: Un módulo envía el valor del parámetro real a un segundo módulo, copiando sólo el valor en el parámetro formal correspondiente, lo que implica que el subprograma que lo recibe, trabaja con una copia del parámetro en que se recibe, lo cual implica que el parámetro real no se modifica, sólo varía la copia del parámetro.
- Paso por referencia: El programa que llama al módulo manda la referencia de memoria del parámetro actual al parámetro formal; el módulo que lo recibe lo puede modificar y su valor es alterado del original.

REFERENCIAS

1. Cairó, O., *Metodología de Programación*. 1995, México: Computec.

2. Joyanes, L. and I. Zahonero, *Programación en C. Metodología, algoritmos y estructuras de datos*. 2a. ed. 2005, España: Mc Graw Hill.

3. Tenembaum, et al., *Estructuras de datos en C*. 1997, México: Prentice-Hall.

4. Joyanes A., L. and I. Zahonero M., *Estructura de Datos. Algoritmos, Abstracción y Objetos*. 1998, Madrid: McGraw-Hill.

5. Criado, M.A., *Programación en Lenguajes Estructurados*. 2006, México: Alfaomega.

UNIDAD DE COMPETENCIA II
Aplicar las principales estructuras de datos lineales

25 HRS.
5 SEMANAS

ESTRUCTURAS DE DATOS DINÁMICAS

Las estructuras de datos dinámicas son una colección de elementos denominados nodos que se amplían (expanden) o reducen (contraen) a medida que se requiera durante la ejecución de un programa cambiando sus posiciones de memoria y éstos se enlazan o encadenan juntos. Este enlace se hace con una variable llamada puntero (apuntador) que apunta a la dirección del nodo siguiente [1].

Las estructuras de datos dinámicas son útiles especialmente para almacenar y procesar conjuntos de datos cuyos tamaños cambian durante la ejecución del programa [1].

Este tipo de estructuras de datos pueden ser divididas en dos tipos: *lineales y no lineales.* Entre las lineales se tienen la *lista*, la *pila* y la *cola*; y entre las no lineales se tienen los *árboles* y los *grafos* en sus distintas modalidades.

Esta unidad de aprendizaje abarca exclusivamente las estructuras de datos lineales en su implementación estática.

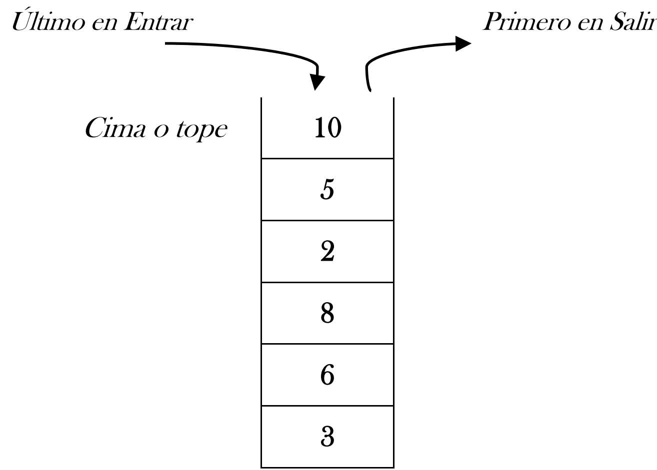
2.1 PILAS: REPRESENTACIÓN. OPERACIONES (INSERCIÓN, ELIMINACIÓN, PILA LLENA, PILA VACÍA). APLICACIONES.

La ***pila (stack)*** es un **TAD**, la cual es definida como una estructura de datos en la que todas las inserciones y eliminaciones de elementos se realizan por un mismo extremo denominado *cima o tope [1-3]*.

En una pila el último elemento añadido es el primero en salir de la pila, es decir *último en entrar, primero en salir*, debido a esta propiedad específica se conoce a las pilas como una estructura de datos **LIFO** (Last In, First Out) [1, 2].

2.1.1 Representación

Esquemáticamente la pila se puede representar de la siguiente manera.



2.1.2 Operaciones del TAD *pila*

Las operaciones más usuales en la pila son ***Insertar (push)*** y ***Quitar (pop),*** considerando otros nombres por diferentes autores como *Meter, Añadir* y *Sacar, Borrar)* [1-4].

La operación **Insertar (push)**, añade un elemento en la cima de la pila, en contraparte de la operación **Quitar (pop)** elimina o saca un elemento de la pila recordando la propiedad **LIFO**.

Por ejemplo:

Insertar (7)	Insertar (9)	Quitar	Insertar (4)	Insertar (8)
				8
	9		4	4
7	7	7	7	7

Actividad 13

Las operaciones básicas que definen el **TAD pila** son las siguientes:

Insertar: Meter un dato a la pila
Quitar: Eliminar (sacar) un dato de la pila
Pila Vacía: Comprobar si la pila no tiene elementos
Pila Llena: Comprobar si la pila está llena de elementos

Recordando que la pila es un TAD, también pueden definirse operaciones dependiendo de la implementación que sea requerida, por esta razón los autores integran diferentes operaciones como:

Cima: Devuelve el elemento que está en la cima de la pila.
Inicializar: Quita todos los elementos y deja la pila vacía. Colocar el puntero de la pila en la primera posición -1.
Tamaño de la pila: número de elementos máximos que puede tener la pila, esto en caso de que sea implementado con un arreglo (vector).

Con base en lo anterior, los algoritmos de las operaciones *Insertar, Quitar Pila Vacía, Pila Llena, Cima e Inicializar* de una pila son los presentados en la Tabla 1:

Tabla 1. Operaciones de una pila

Operación	Algoritmo	Especificaciones
Insertar (push)	1. Verificar si la pila no está llena 2. Incrementar en 1 el puntero (tope) de la pila 3. Almacenar el elemento nuevo en la posición del puntero de la pila	Verificar que la pila no esté llena antes de intentar insertar un elemento. Si está llena el programa debe enviar un mensaje de error y el programa debe terminar su ejecución
Quitar (pop)	1. Verificar si la pila no está vacía	Verificar que la pila no esté vacía antes de

Operación	Algoritmo	Especificaciones
	2. Leer el elemento de la posición del puntero (tope) de la pila 3. Decrementar en 1 el tope de la pila	intentar quitar un elemento. Si está vacía el programa debe enviar un mensaje de error y el programa debe terminar su ejecución
Pila Vacía	1. Verificar si el puntero de la pila vale 0	Devuelve 1 (verdadero) si la pila está vacía y 0 (falso) en caso contrario.
Pila Llena	1. Verificar si el puntero de la pila vale el número máximo de elementos permitidos en el arreglo especificado en la declaración de la pila.	Devuelve 1 (verdadero) si la pila está llena y 0 (falso) en caso contrario.
Inicializar	1. Asignar al puntero de la pila (tope) el valor de 0. *NOTA: Esta posición (valor) varía dependiendo del lenguaje de programación en que se codifica el programa.*	Se limpia o vacía la pila, dejándola sin elementos.
Cima	1. Verificar si la pila no está vacía. 2. Leer el elemento situado en la posición especificado por el puntero de la pila (tope) en el arreglo.	Si la pila no está vacía, devuelve el valor situado en la cima de la pila, pero se no decrementa el puntero de la pila ya que la pila queda intacta. Si está vacía regresa el valor de 0.

2.1.3 Implementación estática y dinámica

La implementación de la pila se puede realizar de manera estática (utilizando arreglos) o de manera dinámica (utilizando punteros como aplicación de una lista ligada. Este apartado será abordado más adelante en el punto **2.4**) [3].

Implementación estática

La implementación estática de una pila se realiza mediante el manejo de arreglos unidimensionales es decir con el uso de vectores. Para esta implementación se declara un registro (por ejemplo denominado *pila*) compuesto de dos campos, el arreglo de datos y el tope o cima de la pila como se muestra a continuación.

pila : Registro
datos[MAX]: tipo de datos
tope: E
FinRegistro

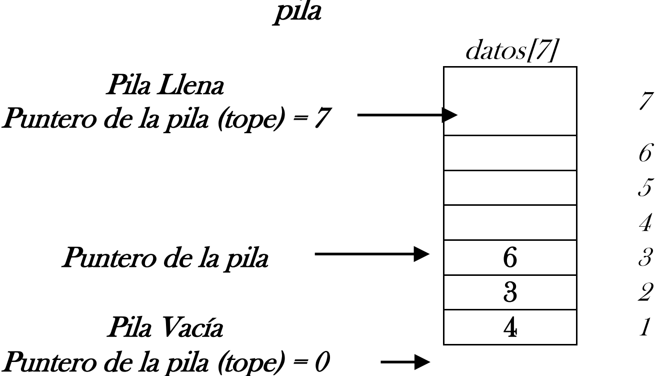
Haciendo referencia a la declaración anterior:

- *datos[ ]:* es el arreglo que en este caso es un vector, en el cual se almacenan los elementos de la pila.
- *MAX:* es el número de elementos máximo que se pueden almacenar en la pila.
- *tipo de datos*: es el tipo de dato del cual se van a almacenar los elementos de la pila, es decir, entero, real, carácter, etc.
- *tope:* es una variable del tipo de dato *pila*, en la cual se almacena la posición del último elemento de la pila o bien de tipo entero.

El método usual de introducir elementos en una pila bajo este esquema, es definir el *fondo* de la pila en la posición 0 del arreglo (vector) y sin ningún elemento en su interior, es decir, definir una *pila vacía [2],* esto implica que el puntero *Tope* tome el valor de 0. De esta

manera para insertar un elemento, primero se incrementa la posición de tope y después se inserta el elemento y así sucesivamente, de tal forma que el *tope* (*puntero de la pila*) se va incrementando cada vez que se añade un nuevo elemento y se va decrementando cada vez que se elimina o quita un elemento de la pila.

Esquemáticamente se puede representar de la siguiente manera:



Ejercicio N° 1:

Considerando las operaciones del TAD Pila de la Tabla 1, representar esquemáticamente el siguiente conjunto de operaciones, considerando una pila que pueda almacenar como máximo 5 elementos:

Instrucción	Valor de Tope	Representación gráfica	Descripción
Inicializar	0	1 2 3 4 5 [ ] [ ] [ ] [ ] [ ]	*Asigna el valor de 0 a puntero de Tope*
Pila Vacía	0	Devuelve 1	*Como la pila está vacía devuelve un **1***
Insertar 18	1	1 2 3 4 5 [18] [ ] [ ] [ ] [ ]	*Tope está en posición **0**, se incrementa el valor de Tope **(1)** y entonces inserta elemento **18.***
Cima	1	Devuelve valor 18	*Devuelve el elemento que está en la posición de Tope (1)*
Insertar 4	2	1 2 3 4 5 [18] [4] [ ] [ ] [ ]	*Tope está en posición **1**, se incrementa el valor de Tope **(2)** y entonces inserta elemento **4.***
Pila Llena	2	Devuelve 0	*Como la pila está llena devuelve un **1** y un **0** si no está llena*
Quitar	1	1 2 3 4 5 [18] [ ] [ ] [ ] [ ]	*Tope está en posición **2**, se decrementa el valor de Tope **(1)** y entonces elimina elemento **4.***
Insertar 100	2	1 2 3 4 5 [18] [100] [ ] [ ] [ ]	*Tope está en posición **1**, se incrementa el valor de Tope **(2)** y entonces inserta elemento **100.***

Instrucción	Valor de Tope	Representación gráfica	Descripción
Cima	2	Devuelve valor 100	*Devuelve el elemento que está en la posición de Tope (2)*
Insertar 50	3	1 2 3 4 5 18 100 50	*Tope está en posición **2**, se incrementa el valor de Tope **(3)** y entonces inserta elemento **50.***
Pila Vacía	3	Devuelve 0	*Como la pila no está vacía devuelve un **0***

Actividad 14

Con base en las especificaciones de las operaciones de una pila, descritas en la Tabla 1 las implementaciones finales de cada una de éstas se presentan en la Tabla 2:

Definición de la pila:

```
Pila : Registro
        datos[MAX]: E                 /*MAX: Máximo número de elementos*/
        tope: E
FinRegistro

*p: Pila
```

Tabla 2. Implementación de las operaciones de una pila

Operación	Algoritmo	Especificaciones	Implementación

Insertar (Push)	1. Verificar si la pila no está llena
	2. Incrementar en 1 el puntero (tope) de la pila
	3. Almacenar el elemento nuevo en la posición del puntero de la pila
		Verificar que la pila no esté llena antes de intentar insertar un elemento.
		
		Si está llena el programa debe enviar un mensaje de error y el programa debe terminar su ejecución
			Modulo Insertar (dato: E)
			Inicio
			    Si (p->tope = MAX) entonces
			        Escribir ("Pila llena")
			    Otro
			        p->tope ← p->tope + 1
			        p->datos[p->tope] ← dato
			    FinSi
			Termina
			
			O bien
			
			Modulo Insertar(*p: pila, dato: E)
			Inicio
			    Si (PilaLlena(p) = 1) entonces
			        Escribir ("Pila llena")
			    Otro
			        p->tope ← p->tope + 1
			        p->datos[p->tope] ← dato
			    FinSi
			Termina

Quitar (Pop)	1. Verificar si la pila no está vacía
	2. Leer el elemento de la posición del puntero (tope) de la pila
	3. Decrementar en 1 el tope de la pila
		Verificar que la pila no esté vacía antes de intentar quitar un elemento.
		
		Si está vacía el programa debe enviar un mensaje de error y el programa debe terminar su ejecución
			Modulo Quitar( )
			Inicio
			    Si (PilaVacia(p) = 1) entonces
			        Escribir ("Pila Vacía")
			    Otro
			        p->tope ← p->tope - 1
			    FinSi
			Termina

Operación	Algoritmo	Especificaciones	Implementación
Pila Vacía	1. Verificar si el puntero de la pila vale 0	Devuelve 1 (verdadero) si la pila está vacía y 0 (falso) en caso contrario.	***Modulo PilaVacia(*p: pila): E*** *Inicio* *Si (p->tope = 0) entonces* *regresa 1* *Otro* *regresa 0* *FinSi* *Termina*
Pila Llena	1. Verificar si el puntero de la pila vale el número máximo de elementos permitidos en el arreglo especificado en la declaración de la pila.	Devuelve 1 (verdadero) si la pila está llena y 0 (falso) en caso contrario.	***Modulo PilaLlena(*p: pila): E*** *Inicio* *Si (p->tope = MAX) entonces* *regresa 1* *Otro* *regresa 0* *FinSi* *Termina*
Inicializar	1. Asignar al puntero de la pila (tope) el valor de 0.	Se limpia o vacía la pila, dejándola sin elementos.	***Modulo Inicializar(*p : pila)*** *Inicio* *p->tope ← 0* *Termina*
Cima	1. Verificar si la pila no está vacía. 2. Leer el elemento situado en la posición especificado por el puntero de la pila (tope) en el arreglo.	Si la pila no está vacía, devuelve el valor situado en la cima de la pila, pero no decrementa el puntero de la pila ya que la pila queda intacta. Si está vacía regresa el valor de 0.	***Modulo Cima(*p: pila)*** *Inicio* *Si (PilaVacia(p) = 1) entonces* *regresa 0* *Otro* *regresa p->datos[p->tope]* *FinSi* *Termina*

NOTA: *Las posiciones del arreglo de la pila dependen del lenguaje de programación en que se codifica el programa. Estas implementaciones son consideradas con un arreglo que inicia en el posición 1.*

✍ Actividades 15-17

Resumen

- La pila, la cola, la lista, el árbol y el grafo son ejemplos de tipos de datos dinámicos.
- Los datos dinámicos se caracterizan por la asignación y uso de localidades de memoria de la computadora.
- El TAD pila se caracteriza por su estructura FILO (Primero en entrar último en salir, *First In Last Out*)
- Existen dos implementaciones del TAD Pila: la estática (vectores) y la dinámica (listas)
- El tratamiento de operaciones de una pila está basado en el apuntador denominado *Tope*
- En la inserción de un elemento, se inserta el elemento y después se incrementa el valor de *Tope*.
- En la eliminación de un elemento, sólo se decrementa el valor de *Tope.*
- En una implementación estática, una pila está vacía cuando no se tienen elementos o bien cuando el valor de Tope es 0.
- En una implementación dinámica, una pila está llena cuando Tope toma el valor del número máximo de elementos permitidos para ser almacenados.

2.2 COLAS: REPRESENTACIÓN. OPERACIONES (INSERCIÓN, ELIMINACIÓN, COLA LLENA, COLA VACÍA). COLA CIRCULAR. APLICACIONES.

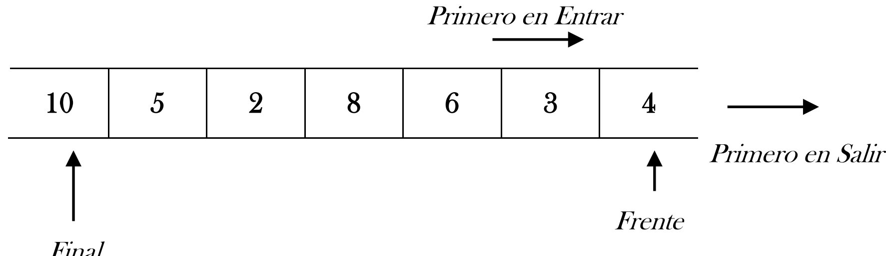
La ***cola (queu)*** es un **TAD**, la cual es definida como una estructura de datos en la que todas las inserciones se realizan por un solo extremo denominado *fondo o final* y todas las eliminaciones por el otro extremo denominado *frente o cabeza [1-3]*, por lo que una cola permite acceder a sus elementos por uno de los dos extremos [2].

En una cola el primer elemento añadido es el primero en salir de la cola, es decir *primero en entrar, primero en salir*, debido a esta propiedad específica se conoce a las colas como una estructura de datos **FIFO** (First In, First Out) [1, 2], es decir almacenan elementos en su orden de aparición [2].

Este tipo de estructura asemeja a una fila de algún servicio de atención, el primero que llega es el primero en ser atendido y en salir.

2.2.1 Representación

Esquemáticamente la cola se puede representar de la siguiente manera.



2.2.2 Operaciones del TAD *cola*

Las operaciones más usuales en la cola, al igual que en la pila, son ***Insertar*** y ***Quitar,*** considerando otros nombres por diferentes autores como *Meter, Añadir* y *Sacar, Borrar [1-4].*

La operación **Insertar**, añade un elemento al final de la cola, mientras que la operación **Quitar** elimina un elemento en el frente de la cola recordando la propiedad **FIFO**.

Por ejemplo:

			Final, Frente
Insertar A			**A**
		Final	*Frente*
Insertar M		**M**	**A**
	Final		*Frente*
Insertar Ñ	**Ñ**	**M**	**A**

	Final			*Frente*
Insertar R	**R**	**Ñ**	**M**	**A**
		Final		*Frente*
Quitar		**R**	**Ñ**	**M**
	Final			*Frente*
Insertar G	**G**	**R**	**Ñ**	**M**
		Final		*Frente*
Quitar		**G**	**R**	**Ñ**
			Final	*Frente*
Quitar			**G**	**R**

Actividad 18

Las operaciones básicas que definen el **TAD cola** son las siguientes:

Insertar: Meter un dato a la cola
Quitar: Eliminar (sacar) un dato de la cola
Cola Vacía: Comprobar si la cola no tiene elementos
Cola Llena: Comprobar si la cola está llena de elementos
Inicializar: Quita todos los elementos y deja la cola vacía.
Frente: Devuelve el elemento frente de la cola.

Con base en lo anterior, los algoritmos de las operaciones antes mencionadas son las descritas en la Tabla 3:

Tabla 3. Operaciones de una cola

Operación	Algoritmo	Especificaciones
Insertar	1. Verificar si la cola no está llena 2. Incrementar en 1 el puntero del final de la cola 3. Almacenar el elemento nuevo en la posición del puntero final de la cola	Verificar que la cola no esté llena antes de intentar insertar un elemento. Si está llena el programa debe enviar un mensaje de error y el programa debe terminar su ejecución
Quitar	1. Verificar si la cola no está vacía 2. Leer el elemento de la posición del puntero (frente) de la cola 3. Incrementa en 1 el frente de la cola	Verificar que la cola no esté vacía antes de intentar quitar un elemento. Si está vacía el programa debe enviar un mensaje de error y el programa debe terminar su ejecución
Cola Vacía	1. Verificar si el frente de la cola es igual al final de la cola	Devuelve 1 (verdadero) si la cola está vacía y 0 (falso) en caso contrario.
Cola Llena	1. Verificar si el final de la cola es mayor que el número máximo de elementos permitidos en el arreglo especificado en la declaración de la cola.	Devuelve 1 (verdadero) si la cola está llena y 0 (falso) en caso contrario.
Inicializar	1. Asignar tanto al puntero frente como final de la cola el valor de 1.	Se limpia o vacía la cola, dejándola sin elementos.

Operación	Algoritmo	Especificaciones
	NOTA: Esta posición (valor) varía dependiendo del lenguaje de programación en que se codifica el programa.	
Frente	1. Verificar si la cola no está vacía. 2. Leer el elemento situado en la posición especificado por el puntero frente de la cola en el arreglo.	Si la cola no está vacía, devuelve el valor situado en el frente de la cola, pero se no incrementa el puntero frente de la cola ya que la cola queda intacta. Si está vacía regresa el valor de 0.

2.2.3 Implementación

La implementación de la cola se puede realizar de manera estática (utilizando arreglos) o de manera dinámica (utilizando punteros como aplicación de una lista ligada). Este apartado será abordado más adelante en el punto **2.4** [3].

Implementación estática

La implementación estática de una cola se realiza mediante el manejo de arreglos unidimensionales es decir con el uso de vectores.

Para esta implementación se declara un registro (por ejemplo denominado *cola*) compuesto de tres campos, el arreglo de datos, el frente y el final de la cola como se muestra a continuación.

cola : Registro
datos[MAX]: tipo de datos
frente: E
final: E
FinRegistro

Haciendo referencia a la declaración anterior:

- *datos[ ]:* es el arreglo que en este caso es un vector, en el cual se almacenan los elementos de la cola.
- *MAX:* es el número de elementos máximo que se pueden almacenar en la cola.
- *tipo de datos*: es el tipo de dato del cual se van a almacenar los elementos de la cola, es decir, entero, real, carácter, etc.
- *frente:* es una variable en la cual se almacena la posición del primer elemento de la cola.
- *final:* es una variable en la cual se almacena la posición del último elemento de la cola.

La implementación de una cola con *vectores* queda esquematizada en la siguiente figura:

1	2	3	4	5	6	7
10	5	2	8	6	3	4

Frente *Final*

La inserción de un elemento en una cola se realiza con el puntero de *Final*, el cual se va incrementando cada vez que se ha insertado un elemento nuevo. La eliminación de un elemento, se realiza con el puntero de *Frente* y éste se incrementa para simular una eliminación.

Ejercicio N° 2:

Con base en la descripción de las operaciones de una cola de la Tabla 3, representar esquemáticamente el siguiente conjunto de operaciones, considerando una cola que pueda almacenar como máximo 5 elementos:

Operación	Valor		Representación gráfica	Descripción
	Frente	Final		
Inicializar	1	1	1 2 3 4 5 [ ] [ ] [ ] [ ] [ ]	*Asigna el valor de 1 a los punteros de Frente y Final*
Insertar **A**	1	2	1 2 3 4 5 [**A**] [ ] [ ] [ ] [ ]	*Frente está en posición **1**, se inserta elemento **A**, se incrementa el valor de Final **(2)***
Cola Vacía	1	2	Devuelve 0	*Como la cola no está vacía devuelve un **0***
Insertar **G**	1	3	1 2 3 4 5 [**A**] [**G**] [ ] [ ] [ ]	*Final está en posición **2**, se inserta elemento **G**, se incrementa el valor de Final **(3)***
Frente	1	3	Devuelve valor **A**	*Devuelve el elemento que está en la posición de Frente (1)*
Cola Llena	1	3	Devuelve 0	*Como la cola no está llena devuelve un **0***
Quitar	2	3	1 2 3 4 5 [ ] [**G**] [ ] [ ] [ ]	*Frente está en posición **1**, se incrementa su valor **(2)** y entonces elimina elemento **A**.*
Insertar **B**	2	4	1 2 3 4 5 [ ] [**G**] [**B**] [ ] [ ]	*Final está en posición **3**, se inserta elemento **B**, se incrementa el valor de Final **(4)***
Frente	2	4	Devuelve valor **G**	*Devuelve el elemento que está en la posición de Frente (2)*
Insertar **M**	2	5	1 2 3 4 5 [ ] [**G**] [**B**] [**M**] [ ]	*Final está en posición **4**, se inserta elemento **M**, se incrementa el valor de Final **(5)***
Insertar **P**	2	6	1 2 3 4 5 [ ] [**G**] [**B**] [**M**] [**P**]	*Final está en posición **5**, se inserta elemento **P**, se incrementa el valor de Final **(5)***

Actividad 19

Con base en las especificaciones de las operaciones de una cola, descritas en la Tabla 3 las implementaciones finales de cada una de éstas se presentan en la Tabla 4:

Definición de la cola:

Cola : Registro
datos[MAX]: E *\/*MAX: Número máximo de elementos de la cola*/*
frente: E
final: E
FinRegistro

**c: Cola*

Tabla 4. Implementación de las operaciones de una cola

Operación	Algoritmo	Especificaciones	Implementación
Insertar	1. Verificar si la cola no está llena 2. Almacenar el elemento nuevo en la posición del puntero final de la cola 3. Incrementar en 1 el puntero del final de la cola	Verificar que la cola no esté llena antes de intentar insertar un elemento. Si está llena el programa debe enviar un mensaje de error y el programa debe terminar su ejecución	***Modulo Insertar (dato:E)*** *Inicio* *Si(ColaLlena(c)=1) entonces* *Escribir("Cola llena")* *Otro* *c->datos[c->final] ← dato* *c->final ← c->final + 1* *FinSi* *Termina*
Quitar	1. Verificar si la cola no está vacía 2. Leer el elemento de la posición del puntero (frente) de la cola 3. Incrementa en 1 el frente de la cola	Verificar que la cola no esté vacía antes de intentar quitar un elemento. Si está vacía el programa debe enviar un mensaje de error y el programa debe terminar su ejecución	***Modulo Quitar( )*** *Inicio* *Si (ColaVacia(c) = 1) entonces* *Escribir ("Cola Vacía")* *Otro* *c->frente ← c->frente + 1* *FinSi* *Termina*
Cola Vacía	1. Verificar si el frente de la cola es igual al final de la cola	Devuelve 1 (verdadero) si la cola está vacía y 0 (falso) en caso contrario.	***Modulo ColaVacia(*c: cola) : E*** *Inicio* *Si ((c->final=1 y c->frente=1) o (c->final < c->frente)) entonces* *Regresa 1* *Otro* *Regresa 0* *FinSi* *Termina*

Operación	Algoritmo	Especificaciones	Implementación
Cola Llena	1. Verificar si el final de la cola es mayor que el número máximo de elementos permitidos en el arreglo especificado en la declaración de la cola.	Devuelve 1 (verdadero) si la cola está llena y 0 (falso) en caso contrario.	***Modulo ColaLlena(*c: cola): E*** *Inicio* *Si (c->final > MAX) entonces* *regresa 1* *Otro* *regresa 0* *FinSi* *Termina*
Inicializar	1. Asignar tanto al puntero frente como final de la cola el valor de 1.	Se limpia o vacía la cola, dejándola sin elementos.	***Modulo Inicializar(*c :cola)*** *Inicio* *c->frente ← 1* *c->final ← 1* *Termina*
Frente	1. Verificar si la cola no está vacía. 2. Leer el elemento situado en la posición especificado por el puntero frente de la cola en el arreglo.	Si la cola no está vacía, devuelve el valor situado en el frente de la cola, pero se no incrementa el puntero frente de la cola ya que la cola queda intacta. Si está vacía regresa el valor de 0.	***Modulo Frente(*c: cola)*** *Inicio* *Si (ColaVacia(c) = 1) entonces* *regresa 0* *Otro* *regresa c->datos[c->frente]* *FinSi* *Termina*

NOTA: *Las posiciones del arreglo de la cola dependen del lenguaje de programación en que se codifica el programa. Estas implementaciones son consideradas con un arreglo que inicia en el posición 1.*

Actividad 20

Resumen

- El TAD cola se caracteriza por su estructura FIFO (Primero en entrar primero en salir, *First In First Out*)
- Existen dos implementaciones del TAD Cola: la estática (vectores) y la dinámica (listas)
- El tratamiento de operaciones de una pila está basado en el manejo de dos apuntadores *Frente y Final*
- En la inserción de un elemento, se inserta el elemento en la posición de *Final* y después se incrementa el valor de éste.
- En la eliminación de un elemento, sólo se incrementa el valor de *Frente.*
- En una implementación estática, una cola está vacía cuando no se tienen elementos o bien cuando el valor de Frente y Final es 1 o Final es menor que Frente.
- En una implementación dinámica, una cola está llena cuando *Final* rebasa el valor del número máximo de elementos permitidos para ser almacenados.

2.3 COLA CIRCULAR

Cuando se realiza la implementación estática (con vectores) de una cola los espacios que van quedando al eliminar los elementos de ésta se "desperdician". Esto se aprecia en el ejercicio **N° 2** (pág. 28):

…

#	Operación	Frente	Final	Representación Gráfica				
				1	2	3	4	5
1	Quitar	2	3		**G**			
				1	2	3	4	5
2	Insertar **B**	2	4		**G**	**B**		
3	Frente	2	4	Devuelve valor **G**				
				1	2	3	4	5
4	Insertar **M**	2	5		**G**	**B**	**M**	
				1	2	3	4	5
5	Insertar **P**	2	6		**G**	**B**	**M**	**P**

Donde se puede ver que en la instrucción **Insertar P (5)** podría decirse que la cola está llena porque en este caso ya el valor del ***Final*** rebasó el valor máximo de elementos permitidos en este ejemplo, sin embargo se aprecia que en la posición **1** no hay elementos y que el valor de ***Frente*** es de 2, lo cual permitiría almacenar más elementos.

Para evitar lo anterior, se utiliza la implementación de una **cola circular** la cual se especifica a continuación.

En una **cola circular** (*arreglo circular*) se considera que la primera posición sigue a la última [3], es decir el elemento anterior al primero es el último [5] o bien al último elemento le sigue el primero [3]. Esto implica que aún ocupado el último elemento del arreglo, pueda añadirse uno nuevo detrás de éste, ocupando la primera posición del arreglo [3]. La implementación de las operaciones de *Insertar y Eliminar* de este tipo de cola se presenta la Tabla 5.

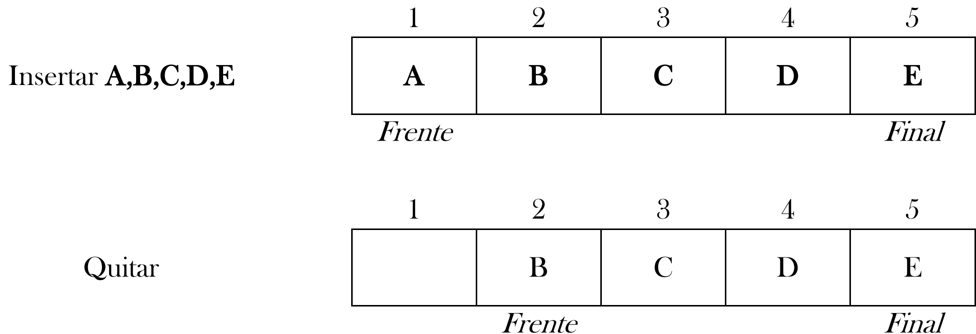
Tabla 5: Inserción y eliminación de elementos en una cola circular.

Operación	Algoritmo	Especificaciones	Implementación
Inicializar	1. Asignar tanto al puntero frente como final de la cola el valor de 0.	Se limpia o vacía la cola, dejándola sin elementos.	***Modulo Inicializar(*c :cola)*** *Inicio* *c->frente ← 0* *c->final ← 0* *Termina*
Insertar	1. Verificar si la cola no está llena y el frente esté en la posición 0 o que la posición final +1 sea igual a frente 2. Si no está llena la cola y el final es el máximo número de elementos, asignar 1 al puntero del final de la cola, si no incrementarlo en 1 3. Almacenar el elemento nuevo en la posición del puntero final de la cola 4. Si el puntero frente es 0 asignarle 1	Verificar que la cola no esté llena antes de intentar insertar un elemento. Si está llena el programa debe enviar un mensaje de error y el programa debe terminar su ejecución	***Modulo Insertar (*c: cola, dato:E)*** *Inicio* *Si((ColaLlena(c)=1 y c->frente = 1) o ((c->final+1) = c->frente)) entonces* *Escribir("Cola llena")* *Otro* *Si (c->final = MAX) entonces* *c->final ← 1* *Otro* *c->final ← c->final + 1* *FinSi* *c->datos[c->final] ← dato* *Si (c->frente = 0) entonces* *c->frente = 1* *FinSi* *FinSi* *Termina*
Cola Llena	1. Verificar si el final de la cola es igual que el número máximo de elementos permitidos en el arreglo especificado en la declaración de la cola.	Devuelve 1 (verdadero) si la cola está llena y 0 (falso) en caso contrario.	***Modulo ColaLlena(*c: cola): E*** *Inicio* *Si (c->final = MAX) entonces* *regresa 1* *Otro* *regresa 0* *FinSi* *Termina*

Operación	Algoritmo	Especificaciones	Implementación
Cola Vacía	2. Verificar si el frente de la cola es igual al final de la cola	Devuelve 1 (verdadero) si la cola está vacía y 0 (falso) en caso contrario.	**Modulo ColaVacia(*c: cola) : E** Inicio Si (c->final=0 y c->frente=0) entonces Regresa 1 Otro Regresa 0 FinSi Termina
Quitar	1. Verificar si la cola no está vacía 2. Verificar que frente y final no sean iguales, si son iguales se les asigna 0 3. Si el frente es igual al máximo de elementos, a frente se le asigna 1, si no es igual se incrementa en 1.	Verificar que la cola no esté vacía antes de intentar quitar un elemento. Si está vacía el programa debe enviar un mensaje de error y el programa debe terminar su ejecución	**Modulo Eliminar(*c: cola)** Inicia Si(ColaVacia(c) = 1) entonces Escribir ("Cola Vacía") Otro Si (c->frente = c->final) entonces c->frente ← 0 c->final ← 0 Otro Si (c->frente = MAX) entonces c->frente ← 1 Otro c->frente ← c->frente + 1 FinSi FinSi FinSi Termina

Ejercicio N° 3:

Considerando las operaciones de *Inserción (Insertar) y Eliminación (Quitar)* de la Tabla 5, realizar la siguiente secuencia de operaciones y su representación gráfica para una cola de máximo 5 elementos (MAX=5).



Si se inserta un elemento más **F** verificando si hay espacio disponible en la cola, entonces *final* se le asigna la 1 y *frente* sigue teniendo su mismo valor.

	1	2	3	4	5
Insertar **F**	**F**	B	C	D	E
	Final	*Frente*			

Si se quitan 3 elementos de la cola entonces el *frente* se incrementa en 1 por cada elemento eliminado.

	1	2	3	4	5
Quitar Quitar Quitar	F				E
	Final				*Frente*

Si se elimina el valor de **E** y como *Frente* = MAX entonces a *frente* se le asigna el valor de 1.

	1	2	3	4	5
Quitar	**F**				
	Frente *= Final*				

Una eliminación más **F**, entonces sólo se le asigna a *frente y final* el valor de 0.

	1	2	3	4	5
Quitar					

Frente = Final = 0

Actividades 21 - 23

Resumen

- **Para evitar el "desperdicio" espacios de almacenamiento en una cola estática, se hace uso de una implementación de una cola circular.**
- **La característica principal de una cola circular es que el valor de *Frente* siempre sigue al de *Final*.**

2.4 LISTAS: REPRESENTACIÓN. OPERACIONES (INSERCIÓN, ELIMINACIÓN, RECORRIDO, BÚSQUEDA). LISTAS SIMPLEMENTE Y DOBLEMENTE LIGADAS. LISTA CIRCULAR, LISTA DOBLE, LISTA DOBLE CIRCULAR. APLICACIONES.

☝La ***lista enlazada*** es una estructura de datos dinámica representada en un conjunto de elementos que se encuentran enlazados y ordenados [3, 6]. Es una colección o secuencia de elementos, llamados **nodos**, dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente por un *enlace* (puntero) [2].

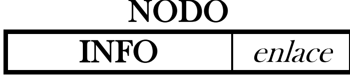
La representación básica [2] de una lista enlazada es la que se presenta a continuación:

Los enlaces se representan por flechas caracterizando la conexión entre dos nodos.

Cada **nodo** se compone de dos partes principalmente [1-4, 6]:

1. La primer parte contiene la *información*.
2. Y la segunda es un campo de tipo puntero (denominado *enlace)* que apunta al siguiente elemento de la lista.

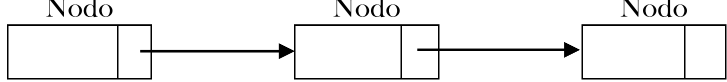
Comúnmente la representación gráfica más extendida del nodo es la que utiliza una caja (un rectángulo) con dos secciones en su interior [2].



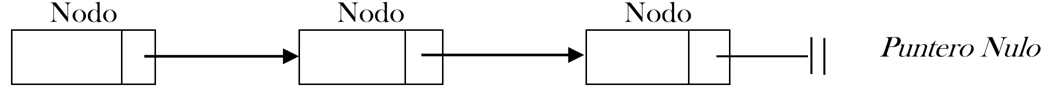
Donde:

INFO: es la información representada en un conjunto de campos, que se desea almacenar y/o manipular.
Enlace: es la dirección (apuntador) del siguiente elemento al que está enlazado o ligado el nodo en cuestión.

De tal manera que con base en la representación gráfica, una lista de nodos queda esquematizada de la siguiente forma:



El nodo de una lista se accede a través de la ***dirección*** que le fue asignada al enlazarlo a la lista. El campo dirección o enlace del último de elemento de la lista no debe apuntar a ningún elemento, no debe tener ninguna dirección, por lo que contiene un valor ***nulo*** *(null) [4],* es decir, el último nodo no debe enlazarse con ningún otro nodo [2].



El *puntero nulo* se utiliza, normalmente, en dos situaciones: [2]

1. Usar el puntero en el campo de enlace o siguiente del nodo final de una lista enlazada (como el esquema anterior)
2. Cuando una lista enlazada no tiene ningún nodo, es decir, está vacía.

Declaración de un nodo

Los nodos de una lista suelen ser normalmente registros. Considerando la representación gráfica del nodo, la declaración del nodo en pseudocódigo sería:

Nodo: Registro
Info /*Conjunto de campos a definir con su tipo de dato*/
**enlace: Nodo*
Fin Registro

Un ejemplo de esta declaración puede ser:

Nodo: Registro
x: E
y: R
cad[15]: S
**enlace: Nodo*
Fin Registro

Para poder hacer uso y referenciar a este nuevo tipo de dato denominado *Nodo* también se tiene que declarar una variable de este mismo tipo (*Nodo*), ejemplificando:

Nod : Nodo /*Variable Nod de tipo Nodo*/
**ANod : Nodo* /*Variable apuntador ANod de tipo Nodo*/

Bajo este ejemplo, la referencia a elementos de las variables *Nod* y *Anod* es:

Nod	*ANod
Nod.x	ANod->x
Nod.y	ANod->y
Nod.cad	ANod->cad
Nod.enlace	ANod->enlace

Por otro lado, el número de elementos o nodos de una lista puede variar rápidamente en un proceso, aumentando los nodos por inserciones o disminuyendo por supresión (eliminación) de nodos [4]. De aquí que las operaciones de una lista son las presentadas a continuación.

2.4.1 Operaciones del TAD *lista*

Para formar el tipo de datos abstracto *lista* las operaciones que se definen son: [2, 4, 7]

ListaVacia: Inicializa la lista o vacía la lista
EsVacia: Función que determina si la lista es vacía

Insertar: Inserta nodos en una lista
Eliminar: Elimina nodos de una lista
Buscar: Busca nodos en una lista
Recorrer: Recorre una lista enlazada (visitar cada nodo de la lista) hasta llegar al último elemento

De las operaciones anteriores se pueden desagregar más operaciones como:
InsertarPrim: Inserta un nodo como primer nodo de la lista
InsertarFin: Inserta un nodo como último nodo de la lista
EliminarPrim: Elimina el primer nodo de la lista
EliminarFin: Elimina el último nodo de la lista
Anterior: Encuentra el nodo anterior a otro nodo especificado
Siguiente: Encuentra el nodo siguiente a otro nodo especificado
Primero: Encuentra el primer nodo de la lista
Ultimo: Encuentra el último nodo de la lista
Visualiza: Despliega la información de cada nodo de la lista

Dependiendo del tipo de enlace que se tenga entre los nodos de una lista ésta se puede clasificar como lista simplemente enlazada, lista doblemente enlazada, lista circular simplemente enlazada y lista circular doblemente enlazada, éstas se describen a continuación.

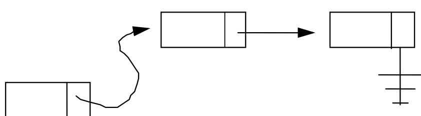
2.4.2 Clasificación de las listas enlazadas [2]

Las listas se pueden clasificar en 4 tipos:

- Lista simplemente enlazada
- Lista doblemente enlazada
- Lista circular simplemente enlazada
- Lista circular doblemente enlazada

2.4.3 Lista Simplemente Enlazada

Es un conjunto de elementos en el que cada uno de éstos contiene la posición o dirección de un elemento único (siguiente) de la lista. Es eficiente en recorridos directos (“adelante”) [2, 6]. Esquemáticamente se representa en la siguiente figura.



2.4.3.1 Representación y declaración

Recordando el esquema de definición de un nodo, la declaración de un nodo en la lista simplemente enlazada se especifica como:

Nodo: Registro
Info /*Conjunto de campos a definir con su tipo de dato*/
**sig:Nodo* /*Apuntador al elemento siguiente de la lista*/
Fin Registro

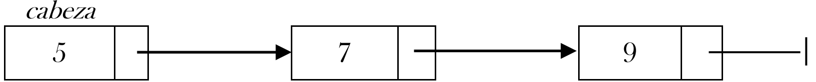
Además de definir siempre un apuntador *cabeza* que permita identificar el inicio de la lista.

cabeza: Nodo ***/*puntero cabeza (inicio o principio) de la lista*/***

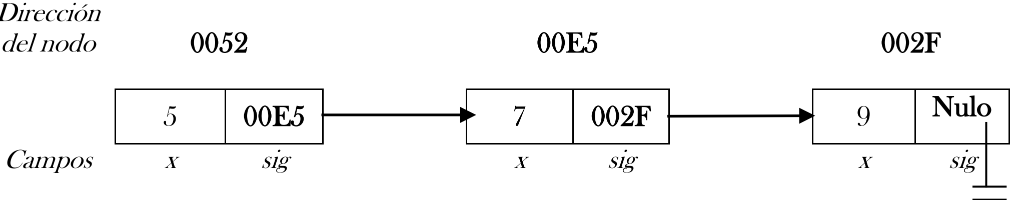
Un ejemplo de esta declaración es:

Nodo: Registro
x: E
**sig: Nodo*
Fin Registro
**cabeza: Nodo*

Esta declaración podría esquematizarse de la siguiente manera:



Recordando que a cada nodo se le asigna una dirección de memoria para poder hacer referencia a éste, entonces se tendría:



lo que significa que el nodo **0052** apunta en su campo de *sig* al nodo **00E5**, éste a su vez al **002F** y éste finalmente a **Nulo.**

Como ya se había mencionado anteriormente, cuando la lista está vacía el puntero (cabeza) debe ser nulo, de la misma manera cuando la lista tenga elementos el puntero de enlace al siguiente elemento del último nodo también debe ser nulo.

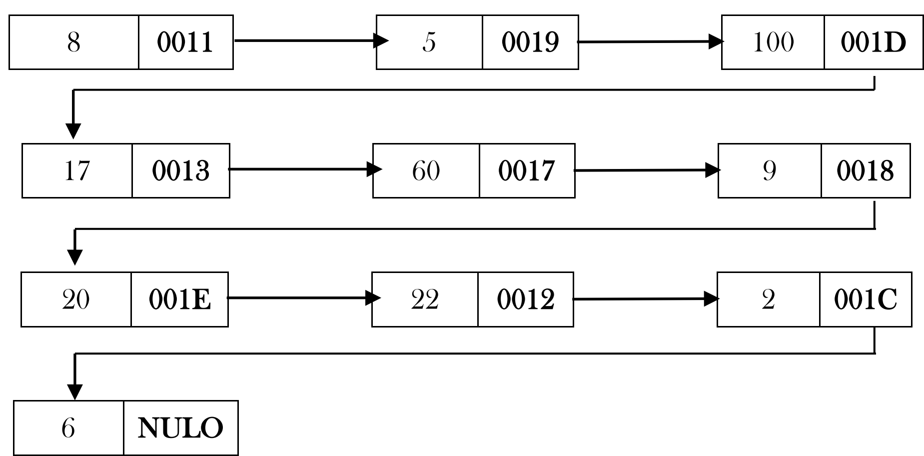
Ejercicio Nº 4

Considerar que la siguiente cuadrícula es la memoria de la computadora en la cual se almacenan los nodos de una lista simplemente ligada, bajo esta representación, realizar el esquema de la lista que se genera.

	info	*sig*	
0011	5	0019	
0012	2	001C	
0013	60	0017	
0014			
0015	8	0011	*cabeza*
0016			
0017	9	0018	

	info	*sig*
0018	20	001E
0019	100	001D
001A		
001B		
001C	6	NULO
001D	17	0013
001E	22	0012

ESQUEMA



Actividad 24

2.4.3.2 Operaciones básicas

Cualquier operación que se quiera implementar de una lista simplemente ligada debe poder manejar un puntero de cabeza para identificar el inicio de la lista, el cual se define:

Nodo: Registro
Info /**Conjunto de campos a definir con su tipo de dato**/
sig:Nodo* /Apuntador al elemento siguiente de la lista**/
Fin Registro
cabeza: Nodo ***/*puntero cabeza (inicio o principio) de la lista*/***

Considerando lo anterior, en la Tabla 6 se describen las operaciones principales de una lista simplemente enlazada.

Tabla 6. Operaciones de una lista simplemente enlazada

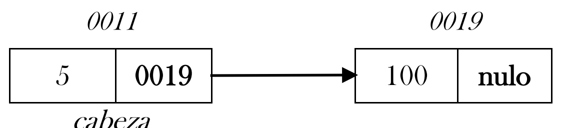
Operación	Algoritmo	Especificaciones
Insertar	1. Asignarle al nuevo nodo una dirección de memoria 2. Asignarle al nuevo nodo los valores de *información* 3. Verificar correcta asignación de memoria 4. Si la lista está vacía, asignarle a *cabeza* la dirección del nuevo nodo 5. Si la lista no está vacía, verificar si la inserción es al inicio, al final o en medio de dos elementos. 6. Realizar los enlaces correspondientes al tipo de inserción.	Verificar que la lista está vacía antes de intentar insertar un elemento. Verificar si la asignación de memoria es correcta, en caso contrario termina ejecución de esta operación. Considerar las ligas a cada uno de los elementos dependiendo del tipo de inserción.
Eliminar	1. Verificar si la lista no está vacía 2. Realizar las ligas correspondientes según sea la eliminación al inicio, al final o entre dos elementos. 3. Liberar la dirección de memoria del elemento a eliminar	Verificar que la lista no esté vacía antes de intentar quitar un elemento. Si está vacía el programa debe enviar un mensaje de error y el programa debe terminar su ejecución
EsVacia	1. Verificar si el apuntador a cabeza es nulo.	Devuelve 1 (verdadero) si la lista está vacía y 0 (falso) en caso contrario.
ListaVacia	1. Asignarle al puntero *cabeza* un nulo.	Se limpia o vacía la lista, creándola o dejándola sin elementos.

Operación	Algoritmo	Especificaciones
Buscar	1. Declarar un apuntador a la lista 2. Asignarle la dirección de la cabeza de la lista 3. Recorrer la lista hasta encontrar el elemento buscado o bien hasta el fin de la lista	Devuelve nulo si no encontró el elemento, en caso contrario la dirección del elemento.
InsertarPrim	1. Asignar al apuntador de enlace al siguiente elemento del nodo nuevo el valor del apuntador de cabeza de la lista 2. Asignar al apuntador *cabeza* la dirección del nuevo nodo	Previamente se identificó en la operación *Insertar* si la lista no estaba vacía y que la asignación de memoria del nuevo nodo fuera correcta.
InsertarFin	1. Recorrer la lista hasta el último elemento 2. Al enlace *siguiente* del nuevo nodo, asignarle el valor de nulo 3. Al enlace *siguiente* del último elemento, asignar la dirección de memoria del nuevo nodo	Previamente se identificó en la operación *Insertar* si la lista no estaba vacía y que la asignación de memoria del nuevo nodo fuera correcta.
EliminarPrim	1. Asignar al apuntador *cabeza* la dirección del apuntador *siguiente* de la *cabeza* de la lista. 2. Liberar el espacio de memoria ocupado por el elemento eliminado	Previamente se identificó en la operación *Eliminar* si la lista no estaba vacía. Liberar el espacio del elemento eliminado utilizando una variable auxiliar que antes de cambiar *cabeza* asigne la dirección de ésta.
EliminarFin	1. Al enlace *siguiente* del penúltimo elemento, asignarle el valor de nulo 2. Liberar la memoria del elemento eliminado	Previamente se identificó en la operación *Eliminar* si la lista no estaba vacía.
Anterior	1. Definir dos apuntadores a la lista y asignarles la dirección de *cabeza,* uno servirá para recorrer la lista y otro para guardar la dirección del elemento anterior al actual. 2. Recorrer la lista hasta encontrar el elemento buscado, ir asignando la dirección actual al anterior antes de avanzar al siguiente nodo.	Devuelve la dirección del elemento anterior al especificado en el argumento
Siguiente	1. Regresa el apuntador al siguiente elemento del nodo especificado	Devuelve la dirección del apuntador al siguiente elemento del nodo buscado
Primero	1. Regresa el valor de *cabeza*.	Devuelve la dirección del nodo *cabeza*. Tomar en cuenta que esta dirección puede ser Nula cuando la lista está vacía.
Ultimo (Recorrer)	1. Declarar un apuntador a la lista 2. Asignarle la dirección de la cabeza de la lista 3. Mientras no sea el último nodo, al apuntador de la lista definido en el punto 1 se le asignará el valor del apuntador al siguiente elemento.	Devuelve el último nodo encontrado
Visualiza	1. Verificar que la lista no esté vacía 2. Para cada elemento de la lista desplegar cada uno de los campos que constituyen a la información del nodo de la lista	Si la lista está vacía desplegar un mensaje de lista vacía y termia la ejecución de esta operación.

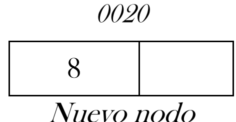
Para un mayor entendimiento las operaciones de inserción y eliminación de una lista se representan esquemáticamente a continuación.

INSERCIÓN AL INICIO (*InsertarPrim)*

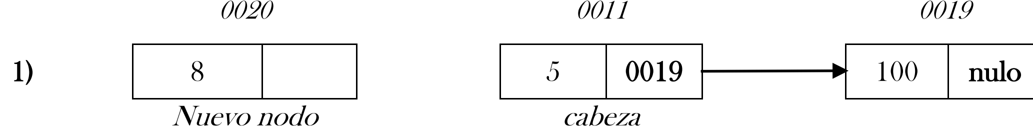
Considerando que la lista no está vacía



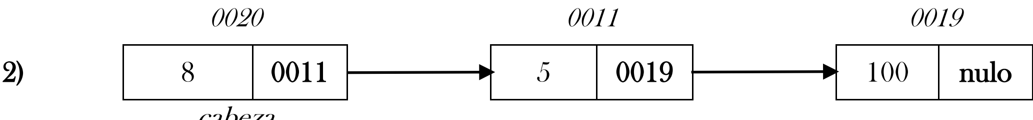
se quiere insertar el nodo:



Entonces,

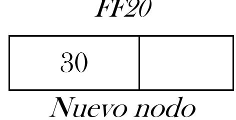


Se asigna al apuntador de *siguiente* del nodo nuevo la dirección de *cabeza, y a cabeza* se le asigna la dirección del nodo nuevo realizando así el enlace entre nodos, y considerando que ahora *cabeza* es el nuevo nodo que se insertó.

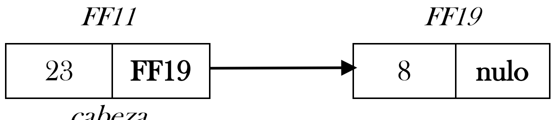


INSERCIÓN AL FINAL (*InsertarFin)*

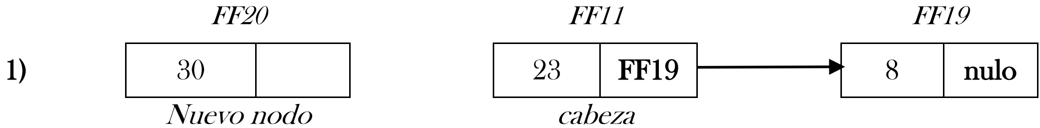
Con otro ejemplo, se quiere insertar el nodo



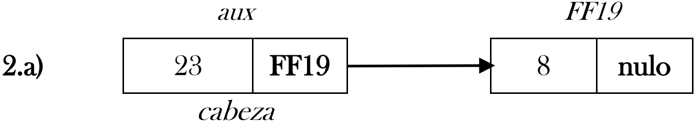
Al final de la lista

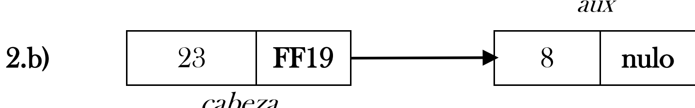


Entonces,

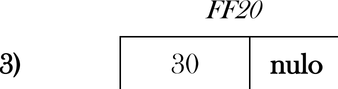


Se define una variable denominada *aux* como apuntador del tipo Nodo (**aux:Nodo*) para recorrer toda la lista hasta que encuentre el último elemento.

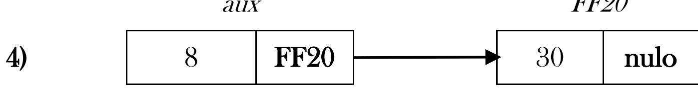




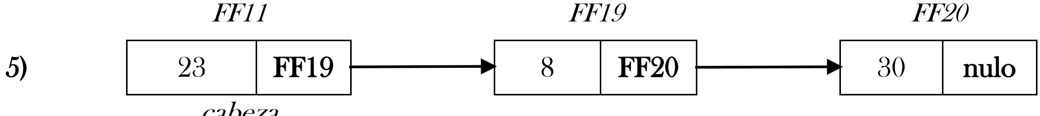
Al apuntador de *siguiente* del nuevo nodo se le asigna nulo.



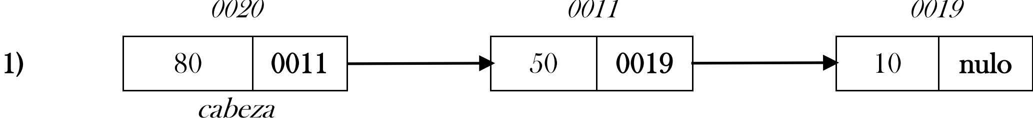
Al apuntador de *siguiente* de *aux* se le asigna la dirección del nuevo nodo.



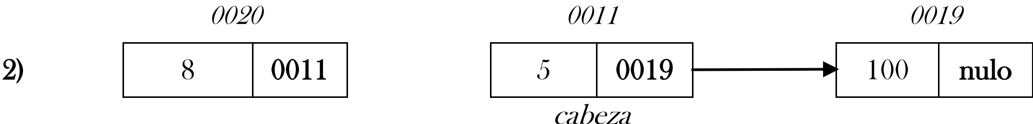
La lista final es



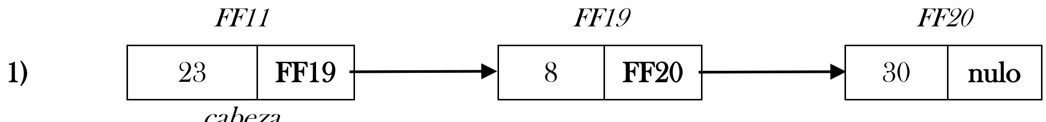
ELIMINACIÓN AL INICIO (*EliminarPrim)*



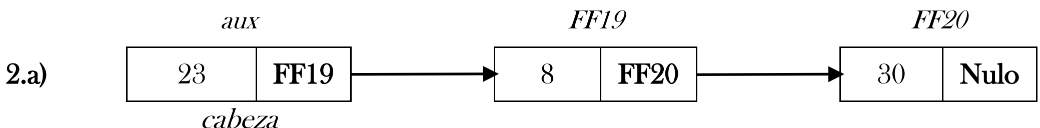
Se asigna a *cabeza* la dirección del apuntador al *siguiente* de *cabeza* y se elimina la liga del primer elemento.

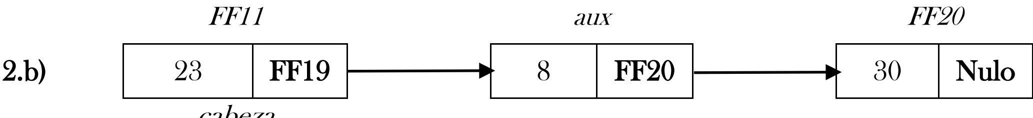


ELIMINACIÓN AL FINAL (*EliminarFin)*

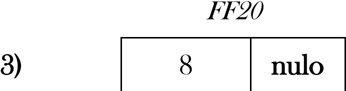


Se define *aux* como apuntador del tipo Nodo para recorrer toda la lista hasta que encuentre el **penúltimo** elemento.

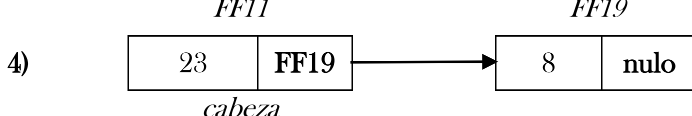




Al apuntador de *siguiente* de *aux* se le asigna nulo.



La lista final es

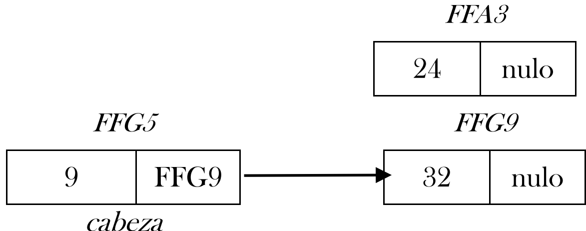


INSERCIÓN Y ELIMINACIÓN ENTRE DOS ELEMENTOS

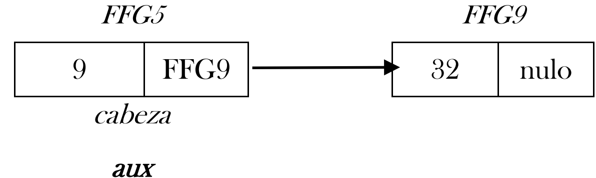
Una variante más que existe en las operaciones de *inserción y eliminación* del elemento de una lista es cuando se inserta un nuevo nodo en una posición específica o cuando se elimina un elemento en específico y que además puede encontrarse entre dos elementos y no precisamente al inicio o al final de la lista. Las representaciones esquemáticas para la implementación de estas dos operaciones se describen a continuación.

Inserción entre dos elementos (*Insertar2*)

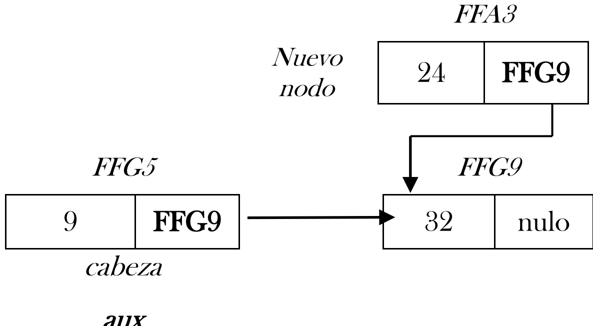
El nodo *FFA3* se inserta entre *FFG5* y *FFG9*



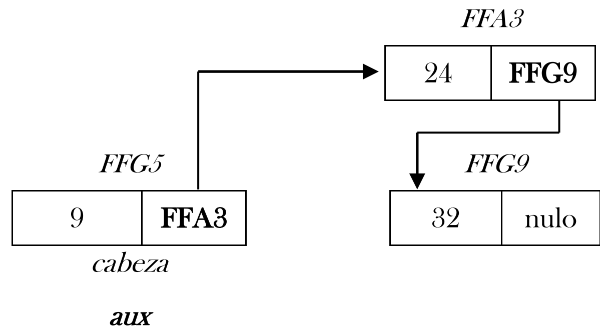
1) Se recorre la lista hasta un elemento antes de la posición en que se quiere insertar el nodo nuevo, esto se realiza con una variable auxiliar a la cual se le debió asignar previamente la dirección de *cabeza.* (En este ejemplo *aux* permanece con la dirección de *cabeza* ya que éste es una posición antes del lugar donde se desea insertar el nuevo elemento)



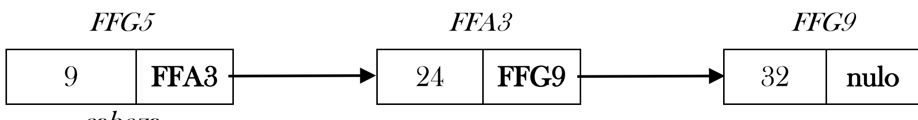
2) Se asigna la dirección del apuntador a *siguiente* de *aux* al apuntador a *siguiente* del nodo a insertar, esto permite enlazar el nuevo nodo a la lista.



3) Se asigna la dirección del apuntador a *siguiente* de *aux* la dirección del nuevo nodo.

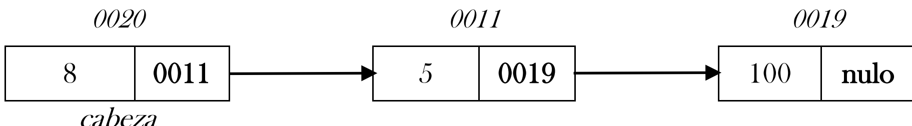


4) La lista final es

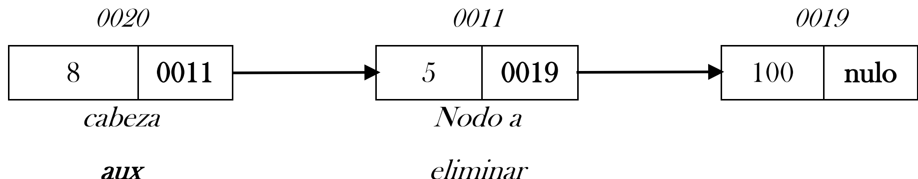


Eliminación entre dos elementos (*Eliminar2*)

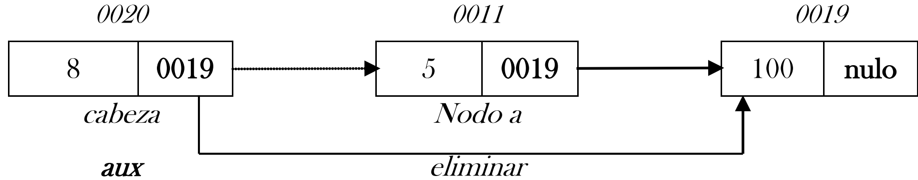
El nodo a eliminar es el que tiene la dirección *0011* del siguiente esquema



5) Se recorre la lista hasta un elemento antes del nodo a eliminar con una variable auxiliar a la cual se le asignó previamente la dirección de *cabeza.* (En este ejemplo *aux* permanece con la dirección de *cabeza* ya que éste es el elemento anterior al nodo a eliminar)

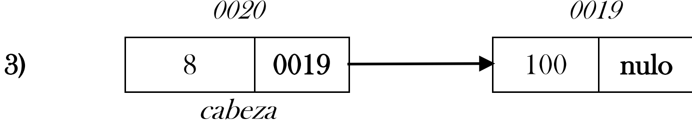


6) Se asigna la dirección del apuntador a *siguiente* de *aux* la dirección del apuntador a *siguiente* del nodo a eliminar.



De esta manera la liga del *siguiente* de *aux* "desaparece" quedando sólo la liga del nodo a eliminar con el de la dirección 0019, sin embargo como la dirección del nodo eliminar ya no la "apunta" ningún otro elemento se puede decir que ya "no es parte" de la lista, quedando el esquema final de la siguiente manera:

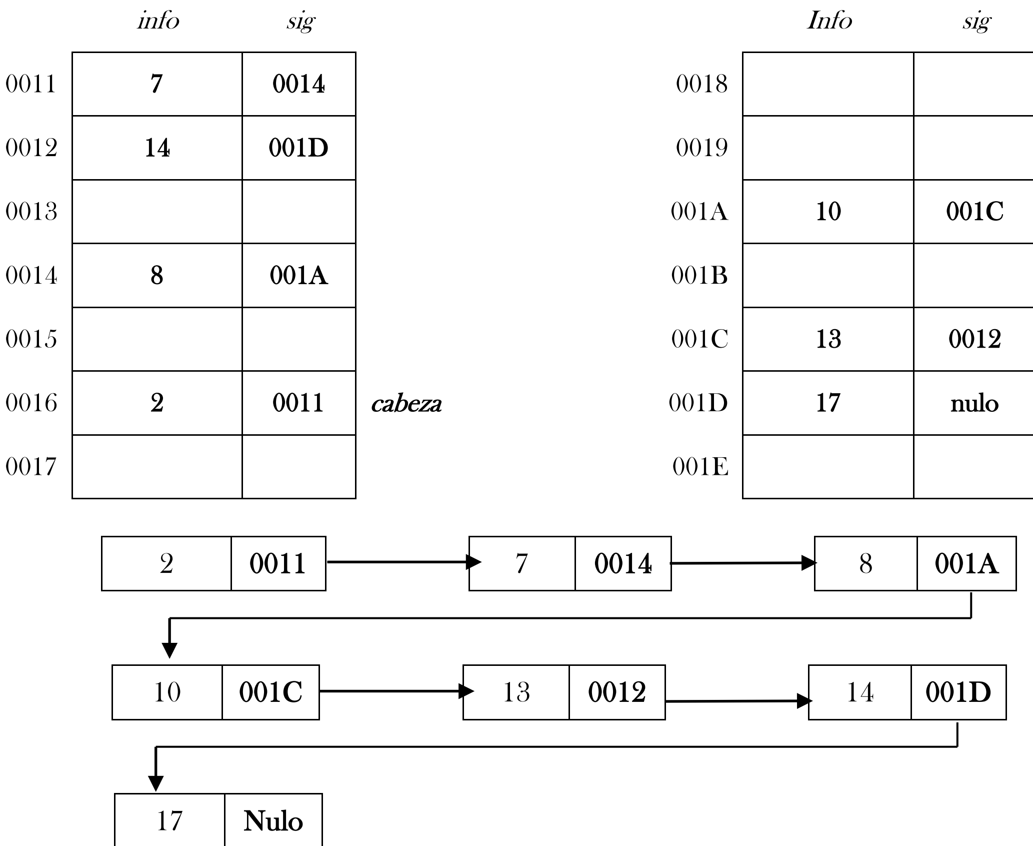
3)



Ejercicio Nº 5

Imagina que la siguiente cuadrícula es la memoria de la computadora en la cual se almacenan los nodos de una lista simplemente ligada. Simular el conjunto de operaciones de la siguiente tabla indicando el resultado final en el cuadriculado de la memoria, realizar su esquema final e identificar el lugar de *cabeza.* Considerar que para cualquier inserción ésta debe ser ***ordenada***, eso implica identificar qué tipo de inserción se debe realizar asimismo el tipo de eliminación.

Operación	Nodo	
	Dirección	Valor
Insertar	0012	8
Insertar	0019	9
Insertar	0018	5
Insertar	001A	10
Insertar	0011	7
Eliminar	-	8
Insertar	0013	15
Eliminar	-	5
Insertar	001C	13
Insertar	0014	8
Insertar	001D	17
Eliminar	-	15
Insertar	0012	14
Insertar	0016	2
Eliminar	-	9



Actividad 25

Implementación de las operaciones básicas de una lista simplemente enlazada.
Las implementaciones de las operaciones de una lista simplemente enlazada se presentan en la Tabla 7, considerando la siguiente declaración

Nodo: Registro
x: E
**sig: Nodo*
Fin Registro

**cabeza: Nodo*

Tabla 7. Implementación de las operaciones de una lista simplemente enlazada

Insertar	InsertarFin
Módulo Insertar (valor:E, *L: Nodo) ***Inicio*** **N: Nodo* *N ← Asignación de memoria* *Si (N ≠ Nulo) entonces* *Si (L = Nulo) entonces* *N->x ← valor* *N->sig ← Nulo* *L ← N*	***Módulo InsertarFin (valor:E, *N: Nodo)*** ***Inicio*** **aux: Nodo* *aux ← Ultimo( )* *N->x ← valor* *N->sig ← Nulo* *aux-> sig ← N* ***Termina***

cabeza ← L *Otro* *InsertarFin(valor,N)* *FinSi* *Otro* *Escribir("Error en Memoria")* *FinSi* ***Termina***	
InsertarPrim	Eliminar
Módulo InsertarPrim (valor:E, *N: Nodo) ***Inicio*** *N->x ← valor* *N->sig ← cabeza* *cabeza ← N* ***Termina***	***Módulo Eliminar (valor:E, *L:Nodo)*** ***Inicio*** **aux: Nodo* *Si (L ≠ Nulo) entonces* *aux ← Buscar(valor)* *Si (aux ≠ Nulo) entonces* *Si (aux = cabeza) entonces* *EliminarPrim()* *Otro* *Si (aux->sig = Nulo) entonces* *EliminarFin(aux)* *Otro* *Eliminar2(aux)* *FinSi* *FinSi* *Otro* *Escribir ("Elemento no encontrado")* *FinSi* *Otro* *Escribir("Lista Vacía")* *FinSi* ***Termina***
EliminarPrim	EliminarFin
Módulo EliminarPrim () ***Inicio*** **aux: Nodo* *aux ← cabeza* *cabeza ← cabeza->sig* *Liberar (aux)* ***Termina***	***Módulo EliminarFin (*N:Nodo)*** ***Inicio*** **aux: Nodo* *aux ← Anterior(N)* *aux->sig ← Nulo* *Liberar (N)* ***Termina***
EsVacia	ListaVacia
Módulo EsVacía () ***Inicio*** *Si (cabeza = Nulo) entonces* *Regresa 1* *Otro* *Regresa 0* *FinSi* ***Termina***	***Módulo ListaVacía ()*** ***Inicio*** *cabeza ← Nulo* ***Termina***
Ultimo(Recorrer)	Buscar
Módulo Ultimo( ): Nodo ***Inicio*** **aux: Nodo* *aux ← cabeza* *Mientras (aux->sig ≠ Nulo)* *aux ← aux->sig* *FinMientras* *Regresa aux* ***Termina***	***Módulo Buscar (valor:E): Nodo*** ***Inicio*** **aux: Nodo* *aux ← cabeza* *Mientras(aux ≠ nulo)* *Si (aux->x ≠ valor) entonces* *aux ← aux->sig* *Otro* *Regresa aux* *FinSi* *FinMientras* *Regresa nulo* ***Termina***

Anterior	Siguiente
Módulo Anterior(*N:Nodo): Nodo ***Inicio*** **aux: Nodo* *aux ← cabeza* *Mientras (aux->sig ≠ N)* *aux ← aux->sig* *FinMientras* *Regresa aux* ***Termina***	***Módulo Siguiente (*N:Nodo): Nodo*** ***Inicio*** *Regresa N->sig* ***Termina***

Primero	Visualiza
Módulo Primero( ): Nodo ***Inicio*** *Regresa cabeza* ***Termina***	***Módulo Visualiza( )*** ***Inicio*** **N: Nodo* *N ← cabeza* *Mientras (N ≠ Nulo)* *Escribir(N->x)* *N ← N->sig* *FinMientras* ***Termina***

Insertar2	Eliminar2
Módulo Insertar2(*N:Nodo, *NIzq:Nodo, *NDer:Nodo) ***Inicio*** *N->sig ← NDer* *NIzq->sig ←N* ***Termina***	***Módulo Eliminar2(*N:Nodo)*** ***Inicio*** **aux: Nodo* *aux ← Anterior(N)* *aux->sig ← N->sig* *Liberar(N)* ***Termina***

Actividades 26 y 27

2.4.3.3 Aplicaciones

Una de aplicaciones principales de una lista simplemente enlazada es la pila, recordando que en el apartado **2.1** se mencionó que esta estructura puede implementarse de manera dinámica y de manera estática. La implementación estática fue mediante vectores y la dinámica mediante el uso de apuntadores a través de una lista simplemente enlazada. A continuación se presenta la implementación de ésta.

- **Implementación dinámica de una pila**

Esta implementación, se basa en definir cada elemento de la pila como un *nodo* de la lista simplemente enlazada, cada inserción se realiza en la cima de la pila (al inicio), quedando su declaración de la siguiente manera:

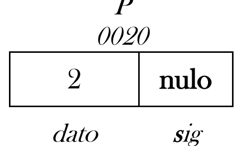
Pila: Registro
dato: tipo de datos
****sig: Pila***
FinRegistro

Sólo por ejemplificar, el siguiente esquema muestra la inserción y eliminación de un nodo en la pila.

P:Pila

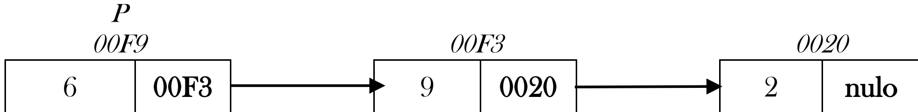
Inserción de un elemento

1. Primer elemento (Pila vacía), apuntador a *siguiente* es nulo



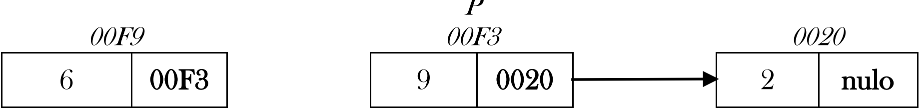
2. Se inserta un segundo elemento, el apuntador a *siguiente* del nuevo elemento apunta a la pila y la pila pasa a ser el nuevo nodo.

3. Se inserta un tercer elemento, el apuntador a *siguiente* del nuevo elemento apunta a la pila



Eliminación de un elemento

1. La pila pasa a ser el elemento *siguiente* que apunta a la misma pila



El elemento eliminado en este caso fue el 6 que fue el último en entrar, recordar que la pila es una estructura **LIFO** último que entra primero que sale.

Los algoritmos de implementación de estas dos operaciones son los mostrados en la Tabla 8

Pila: Registro
dato: E
**sig: Pila*
FinRegistro

Tabla 8. Inserción y eliminación de una pila con estructuras dinámicas

Insertar	Eliminar
Módulo Insertar (valor:E, **P: Pila) ***Inicio*** **N: Pila* *N ← Asignación de memoria* *Si (N ≠ Nulo) entonces* *N->dato ← valor* *N->sig ← (*P)* **P ← N* *Otro* *Escribir("Error en Memoria")* *FinSi* ***Termina***	***Módulo Eliminar (*P: Pila)*** ***Inicio*** **aux: Pila* *Si (EsVacia(*P) = 1) entonces* *Escribir("Pila Vacía")* *Termina* *FinSi* *aux ← (*P)* *(*P) ← aux->sig* *Liberar(aux)* ***Termina*** ***Módulo EsVacía (*P: Pila)*** ***Inicio***

	Si (*P = NULO) entonces Regresa 1 Otro Regresa 0 FinSi Termina

Actividad 28

Aplicación de una Pila en el Tratamiento de expresiones aritméticas (Notación Infija, Prefija, Postfija).

Una de las aplicaciones típicas de una *pila* es almacenar los caracteres de que consta una *expresión aritmética* con el fin de evaluar el valor numérico de dicha expresión.

Una *expresión aritmética* consta de operandos y operadores [1], es decir, la expresión

*A+B-C*D*

tiene como:

Operandos: **A B C D**
Operadores: **+ - ***

Se dice que todo *operador* se encuentra en medio de dos *operandos*.

Es importante recordar que la evaluación y cálculo de una expresión depende de la precedencia o prioridad de los operadores. Esta prioridad se muestra en la Tabla 9 considerando sólo los operadores más básicos.

Tabla 9: Prioridad de operadores

Operador	Descripción
( )	Paréntesis
^	Potencia
* y /	Multiplicación y división
+ y -	Suma y resta

Esto es:

a) $5 * 7 + 3 = 38$ no es lo mismo que $5 * (7 + 3) = 50$

b) $16 - 2 * 4 = 8$ no es lo mismo que $(16 - 2) * 4 = 56$

En ambos incisos, se resuelve primero la multiplicación debido a la precedencia de los operadores, es decir, la prioridad de la multiplicación es superior a la de la suma o en su caso la resta.

Ejercicio N° 6:

Calcular el resultado de las siguientes expresiones (auxíliate de la Tabla 9 para la prioridad de los operandos):

a) **5 * 4 – 6 * 2 = 8**
   a. 20 – 12   */*Primero se realizan las dos multiplicaciones y */*
   b. 8   */*después la resta*/*
b) **4 + 6 * 7 = 46**
   a. 4 + 42   */*Primero se realiza la multiplicación y */*
   b. 46   */*después la suma*/*
c) **5 + 3 – 8 / 4 = 6**
   a. 5 + 3 – 2   */*Se realiza la división de 8/4 */*
   b. 8 – 2   */*Como la + y la – tienen la misma prioridad puede resolverse de cualquier manera */*
   c. 6   */*Finalmente la resta*/*

Si a los incisos del ejercicio anterior se le colocan paréntesis para generar una expresión diferente, entonces el resultado cambia, recordar que por prioridad primero se resuelve la operación especificada dentro de los paréntesis.

a) **5 * (4 – 6) * 2 = -20**
   a. 5 * (-2) * 2   */*Se realiza la resta (4-6) */*
   b. (-10) * 2   */*después la multiplicación*/*
   c. -20
b) **(4 + 6) * 7 = 70**
   a. 10 * 7   */*Se realiza la suma de 4 + 6 */*
   b. 70   */*después la multiplicación*/*
c) **(5 + 3 – 8) / 4 = 0**
   a. 0 / 4   */*Se realiza la suma y la resta de 5 + 3 - 8*/*
   b. 0   */*Se realiza finalmente la división */*

El uso de paréntesis además de asociar operaciones también facilita y enfatiza la lectura y el cálculo del resultado final de una expresión, por ejemplo:

*A + B * C* expresión enfatizada *A + (B * C)*

Ejercicio Nº 7:

Calcular el resultado de las siguientes expresiones, colocar paréntesis a las expresiones en donde se pueda enfatizar la operación.

a) 1 + 4 * 6 – 3 =
   a. 1 + (4 * 6) – 3 =
   b. 1 + 24 – 3 =
   c. **22**
b) 2 ^ 2 * 3 – 4 + 20 / 2 / (2 + 3) =
   a. (((2 ^ 2) * 3) – 4) + ((20 / 2) / (2 + 3)) =
   b. ((4 * 3) – 4) + (10 / 5) =
   c. (12 – 4) + 2 =
   d. 8 + 2 =
   e. **10**

Actividad 29

Como se mencionó anteriormente, unas de las aplicaciones principales de la *pila* es el de calcular el resultado de expresiones aritméticas. Estas expresiones pueden representarse en distintas notaciones. Las más usadas son: [1, 3, 4]

- **Infija:** los operadores aparecen en el centro de los operandos.
- **Postifija (Polaca Inversa):** los operadores aparecen detrás y los operandos enfrente.
- **Prefija (Polaca):** los operadores aparecen delante de los operandos.

Y en el orden que deben ser evaluados, esto quiere decir, considerando la precedencia operadores.

Ejemplo:

*(a+b)*c-d/(e+f)*

expresión *infija: a + b * c – d / e + f*
expresión *postfija: a b + c * d e f + / -*
expresión *prefija: - * + a b c / d + e f*

Cabe mencionar que para representar cualquier tipo de expresión en *infija, posfija o prefija,* no se escriben los paréntesis en el resultado.

Más a detalle, para representar en *postfijo* la expresión anterior *(a+b)*c-d/(e+f)* se siguieron los siguientes pasos:

- Primero se recomienda colocarle a la expresión original los paréntesis necesarios para enfatizar la prioridad de las operaciones (en caso de que así se requiera):

  ((a+b)*c) – (d/(e+f))

- Segundo, uno vez enfatizada la expresión, ir colocando primero los operandos y después los operadores dependiendo de la procedencia de la operación más interna.

Paso	Expresión	Descripción
0	***((a+b)*c) – (d/(e+f))***	*Expresión inicial*
1	*(**(ab+)** *c) – (d/(e+f))*	*El primer operando ((a+b)*c) se divide en otro primer operando (a+b), de éste el primero es **a**, el segundo operando es **b** y el operador es **+**, entonces quedaría **ab+***
2	*(ab+**c***) – (d/(e+f))*	*Teniendo **c** como segundo operando y el operador ***** entonces la expresión es **(ab+c*)**. Aquí se completa el primer operando de la expresión inicial.*
3	*(ab+c*) – (d/**(ef+)**)*	*El segundo operando de la expresión inicial (d/(e+f)) se divide en otro primer operando **d** y uno segundo **(e+f)**, de este el primero es **e**, el segundo operando es **f** y el operador es **+**, entonces quedaría **ef+***
4	*(ab+c*) – **(def+/)***	*Teniendo **d** como primer operando y el operador / entonces la expresión es **(def+/)**. Aquí se completa el segundo operando de la expresión inicial.*
5	***ab+c*def+/-***	*Finalmente colocando los dos operandos de la expresión inicial obtenidos en los pasos **1** y **4**, y considerando como operador el signo **–** la expresión resultante es **ab+c*def+/-***

Bajo el mismo tratamiento, se realiza el *prefijo* de la expresión, considerando el orden correcto: primero operador y después operandos.

Paso	Expresión
0	*((a+b)*c) – (d/(e+f))*
1	*-((a+b)*c) (d/(e+f))*
2	*-((+ab)*c) (d/(e+f))*
3	*-*+abc (d/(e+f))*
4	*-*+abc (d/(+ef))*
5	*-*+abc (/d+ef)*
6	***-*+abc/d+ef***

Actividad 30

Algoritmo para evaluación de una expresión aritmética
A la hora de evaluar una expresión aritmética escrita, normalmente, en notación *infija* la computadora sigue los siguientes pasos: [1]

1. Transformar la expresión de infija a postfija
2. Evaluar la expresión en postfija

En el algoritmo anterior es fundamental la utilización de *pilas.* Para realizar el algoritmo de evaluación, es necesario primero contar con el algoritmo de transformación de expresión infija a postfija.

Postfija

- Se crea inicialmente la pila
- Se utiliza la pila para almacenar los operadores y los paréntesis izquierdos de la expresión a transformar.
- La expresión se va leyendo caracter por caracter.
- Los operandos se pasan directamente a formar parte de la expresión postfija.
- Los operadores se meten en la pila siempre que esté vacía o cuando el operador que contiene la cima de la pila sea menor que el operador que se leyó de la expresión.

Para comprender el funcionamiento de la obtención de una expresión posfija realizar la **actividad 31.**

Actividad 31

Una vez obtenida la expresión posfija y realizada la actividad 32 se procede al cálculo del resultado de la expresión.

Evaluación de una expresión

Utilizando en la expresión postfija obtenida con la actividad anterior, entonces se evalúa la expresión con el siguiente algoritmo

```
Módulo Evaluar (post[80]: S) : R
Inicia
        cadaux[10]: S
        valor1, valor2, valor3: R
        i: E
        *P: Pila
        *P ← Nulo
        Para (i ← 1; i<= longitud de (post); i ← i+1)
                Si (post[i] es operador) entonces
                        valor2 ← PrimeroP(P)
                        Eliminar(&P)
                        valor1 ← PrimeroP(P)
                        Eliminar(&P)
                        Caso(post[i])
                                post[i] = '*': valor3 ← valor1 * valor2
                                post[i] = '+': valor3 ← valor1 + valor2
                                post[i] = '/': valor3 ← valor1 / valor2
                                post[i] = '-': valor3 ← valor1 - valor2
                        FinCaso
                        Insertar(valor3,&P)
                Otro
                        valor3 ← Transformar a número (post[i])
                        Insertar(valor3,&P)
                FinSi
        FinPara
        valor1 ← PrimeroP(P)
        Eliminar(&P)
        Regresa valor1
Termina
```

Nota: el Módulo *PrimeroP* regresa el valor del tope de la pila.

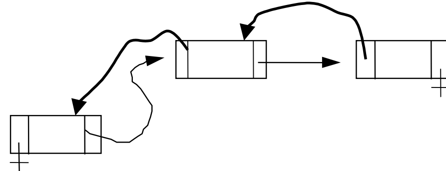
Actividad 32

Resumen

- Una lista es un conjunto de nodos.
- Un nodo está compuesto por dos partes: *información almacenada y un apuntador de enlace.*
- Una lista simplemente enlazada (LSE) está compuesta por un conjunto de nodos enlazados únicamente por un apuntador al siguiente elemento.
- Una LSE está coordinada por un único apuntador que el *inicio o cabeza* de la lista.
- Una LSE está vacía cuando el valor de su apuntador de *inicio o cabeza* es Nulo.
- El último nodo o elemento de una LSE siempre será Nulo.
- La implementación dinámica de una Pila hace uso de una estructura de Lista, donde la inserción de elementos se realiza siempre al inicio considerando la analogía de que *Cabeza* es *Tope.*
- Una aplicación de la estructura Pila dinámica es la evaluación de expresiones, para lo cual se debe obtener primero la expresión posfija de ésta y posteriormente evaluarla.

2.4.4 Lista Doblemente Enlazada

Es un conjunto de elementos en el que cada uno contiene las posiciones o direcciones del elemento siguiente y del elemento anterior. Esquemáticamente se puede representar como se muestra en la siguiente figura.



2.4.4.1 Representación y declaración

Recordando el esquema de definición de un nodo, esta lista esquematizada se representa en la siguiente figura

NODO

DIR	INFO	DIR

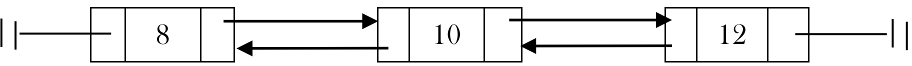
y cuya declaración es:

Nodo: Registro
Info /*Conjunto de campos a definir con su tipo de dato*/
**ant: Nodo* /*Apuntador al elemento anterior de un nodo*/
**sig: Nodo* /*Apuntador al elemento siguiente de un nodo*/
Fin Registro

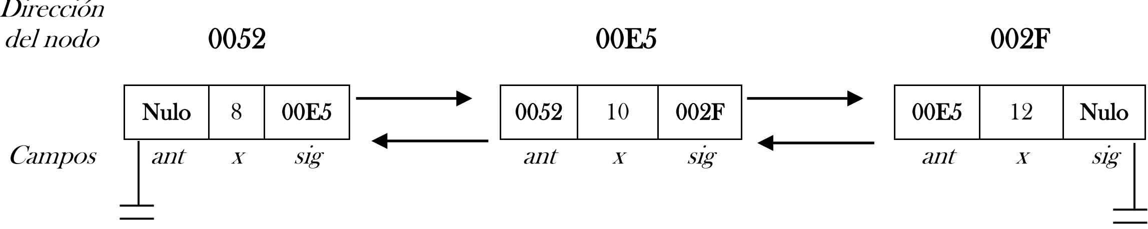
Considerando lo anterior, un ejemplo es:

Nodo: Registro
**ant: Nodo*
x: E
**sig: Nodo*
Fin Registro

Dando 3 valores cualesquiera a 3 nodos para crear una lista doblemente enlazada podría esquematizarse de la siguiente manera:



Recordando que a cada nodo se le asigna una dirección de memoria para poder hacer referencia a éste, entonces se tendría:



Lo anterior significa que el nodo **0052** apunta en su campo de *sig* al nodo **00E5**, éste a su vez al **002F** y éste finalmente a **Nulo.** De la misma manera el nodo **0052** apunta a nulo, el nodo **00E5** a **0052** y **002F** a **00E5** en sus elementos anteriores respectivamente.

Cualquier operación que se quiera implementar de una lista doblemente enlazada debe manejar un puntero de cabeza (inicio) y uno de cola (final) para identificar el inicio y fin de la lista, los cuales se definen del tipo del nodo de la lista.

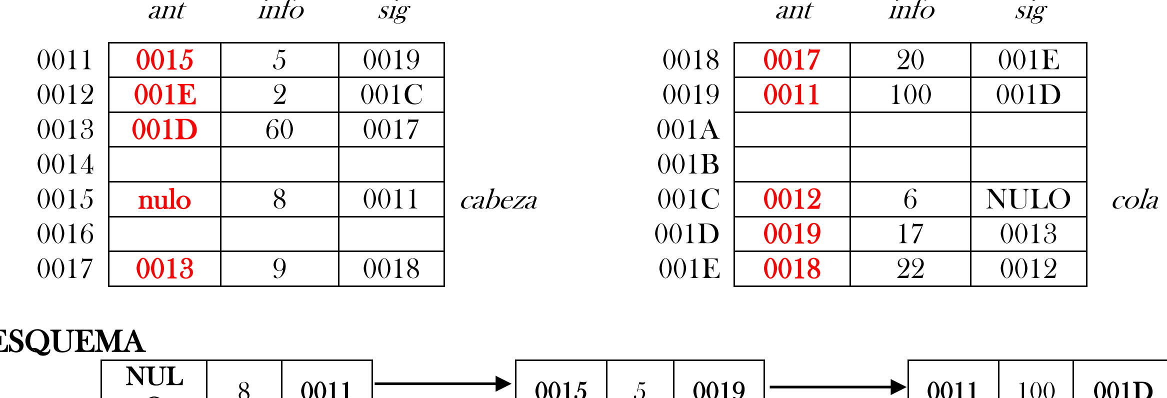
Nodo: Registro
**ant: Nodo*
x: E
**sig: Nodo*
Fin Registro

***cabeza: Nodo**
***cola: Nodo**

Cuando la lista está vacía, por consiguiente, estos punteros (cabeza y cola) deben ser nulos, de la misma manera cuando la lista tenga elementos el puntero de enlace al *siguiente* elemento del último nodo y el puntero de *anterior* del primer nodo también deben ser nulos.

Ejercicio Nº 8:

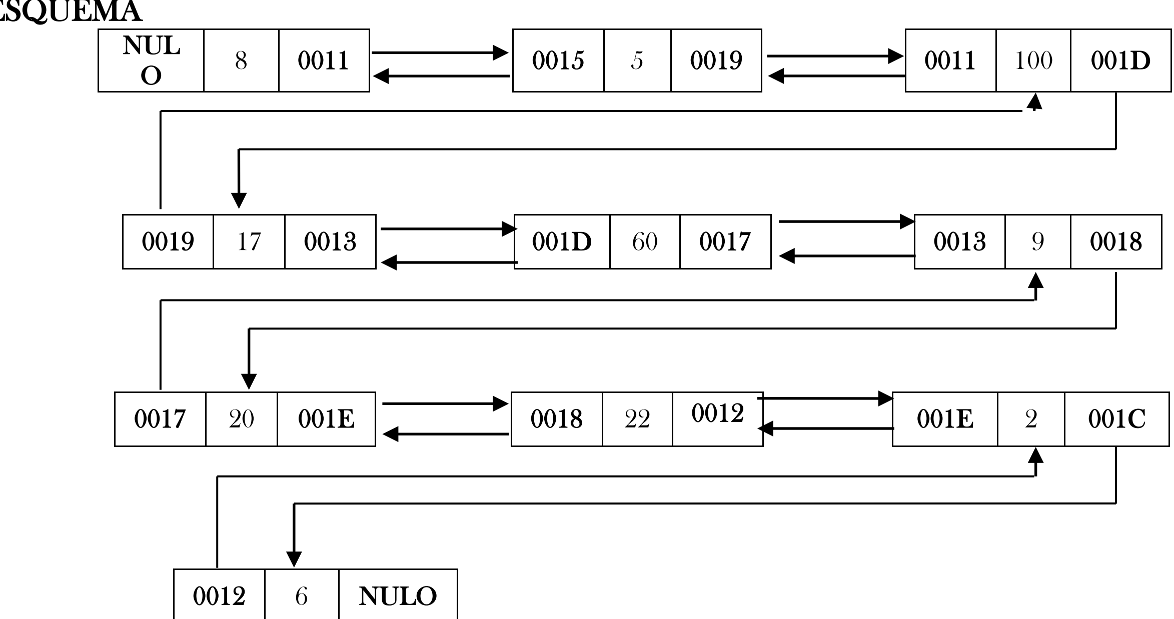
Imagina que la siguiente cuadrícula es la memoria de la computadora en la cual se almacenan los nodos de una lista simplemente ligada, convertirla en doblemente enlazada colocando en la columna *ant* la dirección que representaría el elemento anterior al nodo en cuestión, identificar *cabeza y cola*. Realizar el esquema que representaría la lista.



	ant	*info*	*sig*	
0011	0015	5	0019	
0012	001E	2	001C	
0013	001D	60	0017	
0014				
0015	nulo	8	0011	*cabeza*
0016				
0017	0013	9	0018	

	ant	*info*	*sig*	
0018	0017	20	001E	
0019	0011	100	001D	
001A				
001B				
001C	0012	6	NULO	*cola*
001D	0019	17	0013	
001E	0018	22	0012	

ESQUEMA



Actividad 33

2.4.4.2 Operaciones básicas

Como ya se mencionó, cualquier operación que se quiera implementar con una lista doblemente ligada debe poder manejar un puntero de cabeza para identificar el inicio y un puntero de final de la lista. Para recordar:

Nodo: Registro
**ant: Nodo*
Info
**sig: Nodo*
Fin Registro

**cabeza: Nodo*
**final: Nodo*

Considerando lo anterior, en la Tabla 10 se describen las operaciones principales de una lista doblemente enlazada.

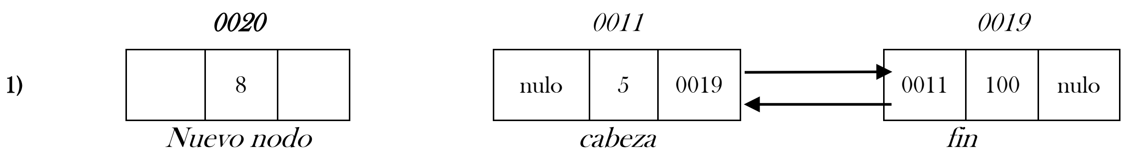
Tabla 10. Operaciones de una lista doblemente enlazada

Operación	Algoritmo	Especificaciones
Insertar	1. Asignarle al nuevo nodo una dirección de memoria 2. Asignarle al nuevo nodo los valores de *información* 3. Verificar correcta asignación de memoria 4. Si la lista está vacía, asignarle a *cabeza* y a *final* la dirección del nuevo nodo 5. Si la lista no está vacía, verificar si la inserción es al inicio, al final o en medio de dos elementos. 6. Realizar los enlaces correspondientes al tipo de inserción.	Verificar que la lista está vacía antes de intentar insertar un elemento. Verificar si la asignación de memoria es correcta, en caso contrario termina ejecución de esta operación. Considerar las ligas a cada uno de los elementos dependiendo del tipo de inserción.
Eliminar	1. Verificar si la lista no está vacía 2. Realizar las ligas correspondientes según sea la eliminación al inicio, al final o entre dos elementos. 3. Liberar la dirección de memoria del elemento a eliminar	Verificar que la lista no esté vacía antes de intentar quitar un elemento. Si está vacía el programa debe enviar un mensaje de error y el programa debe terminar su ejecución
EsVacía	1. Verificar si el apuntador a *cabeza* es nulo.	Devuelve 1 (verdadero) si la lista está vacía y 0 (falso) en caso contrario.
ListaVacía	1. Asignarle a los punteros *cabeza* y *final* un nulo.	Se limpia o vacía la lista, creándola o dejándola sin elementos.
Buscar	1. Declarar un apuntador a la lista 2. Asignarle la dirección de la cabeza de la lista 3. Recorrer la lista hasta encontrar el elemento buscado o bien hasta el fin de la lista	Devuelve nulo si no encontró el elemento, en caso contrario la dirección del elemento.
InsertarPrim	1. Asignar al apuntador de enlace al *siguiente* elemento del nodo nuevo el valor del apuntador de cabeza de la lista 2. Asignar al apuntador de enlace al *anterior* elemento del nodo nuevo el valor de nulo 3. Asignar al apuntador *anterior* de *cabeza* la dirección del nuevo nodo 4. Asignar al apuntador de *cabeza* la dirección del nuevo nodo	Previamente se identificó en la operación *Insertar* si la lista no estaba vacía y que la asignación de memoria del nuevo nodo fuera correcta.
InsertarFin	1. Asignar al apuntador de enlace al *siguiente* elemento del nodo nuevo el valor de nulo 2. Asignar al apuntador de enlace al *anterior* elemento del nodo nuevo el valor de *final* 3. Asignar al apuntador *anterior* de *final* la dirección del nuevo nodo 4. Asignar al apuntador de *final* la dirección del nuevo nodo	Previamente se identificó en la operación *Insertar* si la lista no estaba vacía y que la asignación de memoria del nuevo nodo fuera correcta.
EliminarPrim	1. Asignar al apuntador *cabeza* la dirección del apuntador *siguiente* de la *cabeza* de la lista.	Previamente se identificó en la operación *Eliminar* si la lista no estaba vacía.

Operación	Algoritmo	Especificaciones
	2. Al apuntador de *anterior* de la nueva *cabeza* se le asigna nulo 3. Liberar el espacio de memoria ocupado por el elemento eliminado	Liberar el espacio del elemento eliminado utilizando una variable auxiliar que antes de cambiar *cabeza* asigne la dirección de ésta.
EliminarFin	1. Asignarle al apuntador de *final* la dirección del enlace *anterior* del *final* 2. Al enlace *siguiente* del nuevo final asignarle el valor de nulo 3. Liberar la memoria del elemento eliminado	Previamente se identificó en la operación *Eliminar* si la lista no estaba vacía. Liberar el espacio del elemento eliminado utilizando una variable auxiliar que antes de cambiar *cabeza* asigne la dirección de ésta.
Anterior	1. Devuelve la dirección del apuntador *anterior* del elemento especificado en el argumento	Devuelve la dirección del apuntador *anterior* del elemento especificado en el argumento
Siguiente	1. Devuelve la dirección del apuntador al *siguiente* elemento del nodo buscado	Devuelve la dirección del apuntador al *siguiente* elemento del nodo buscado
Primero	1. Regresa el valor de *cabeza*.	Devuelve la dirección del nodo *cabeza*. Tomar en cuenta que esta dirección puede ser Nula cuando la lista está vacía.
Ultimo	1. Regresa el valor de *final*.	Devuelve la dirección del nodo *final*. Tomar en cuenta que esta dirección puede ser Nula cuando la lista está vacía.
Visualiza	1. Verificar que la lista no esté vacía 2. Para cada elemento de la lista desplegar cada uno de los campos que constituyen a la información del nodo de la lista	Si la lista está vacía desplegar un mensaje de lista vacía y termia la ejecución de esta operación.

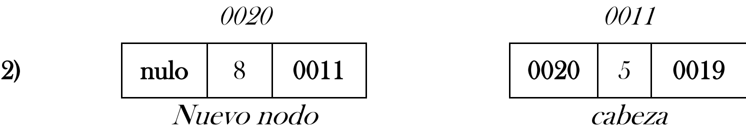
Para un mayor entendimiento las operaciones de inserción y eliminación de una lista se representan esquemáticamente a continuación.

INSERCIÓN AL INICIO (*InsertarPrim)*

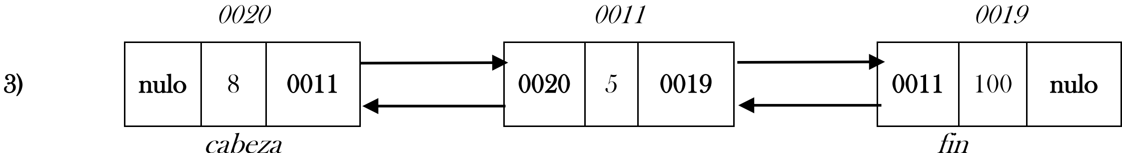


Se realizan las asignaciones:

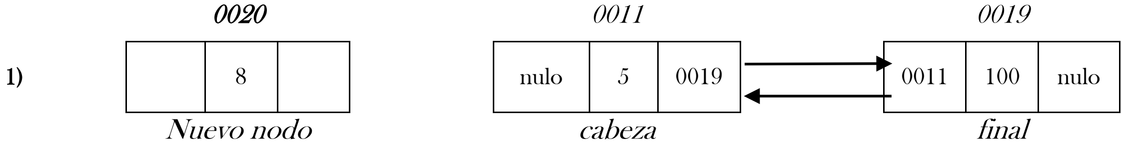
- al apuntador de *siguiente* del nodo nuevo la dirección de *cabeza*
- al apuntador de *anterior* del nodo nuevo se le asigna *nulo*
- al apuntador de *anterior* de cabeza la dirección del nuevo nodo



- a *cabeza* se le asigna la dirección del nodo nuevo.

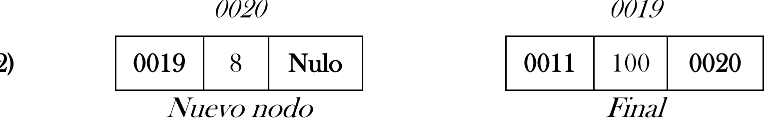


INSERCIÓN AL FINAL (*InsertarFin)*

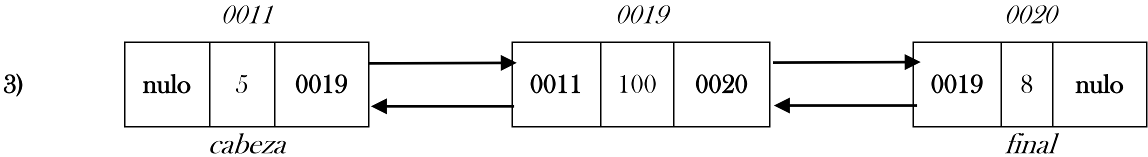


Se realizan las asignaciones:

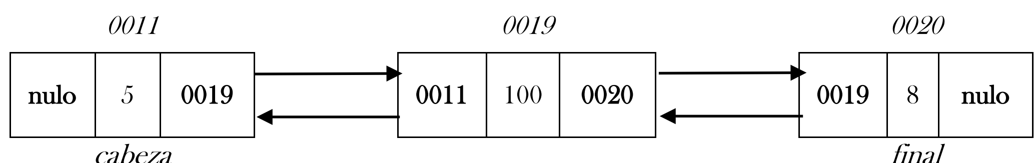
- al apuntador de *anterior* del nodo nuevo la dirección de *final*
- al apuntador de *siguiente* del nodo nuevo se le asigna nulo
- al apuntador de *siguiente* de *final* la dirección del nuevo nodo



- a *final* se le asigna la dirección del nodo nuevo.

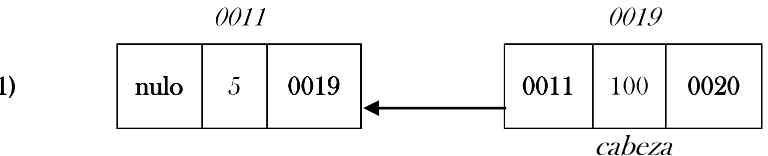


ELIMINACIÓN AL INICIO (*EliminarPrim)*

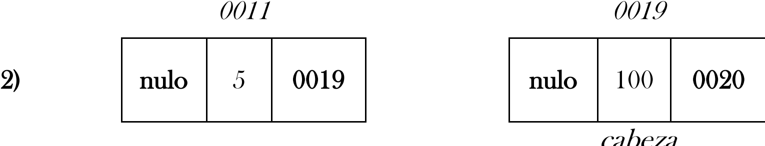


Se asigna:

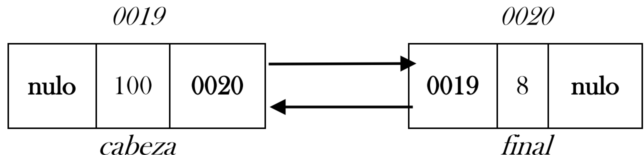
- a *cabeza* la dirección del apuntador al *siguiente* de *cabeza* y se elimina la liga del primer elemento.



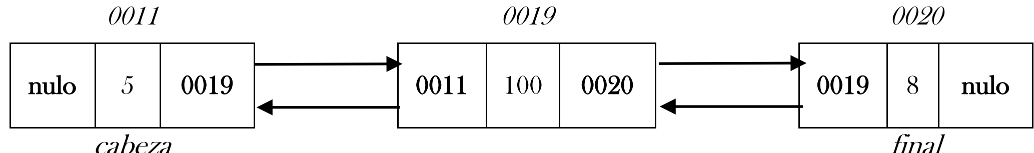
- al apuntador de *anterior* de la ahora *cabeza* el valor de nulo y se “rompe” con la liga al elemento.



Lista final

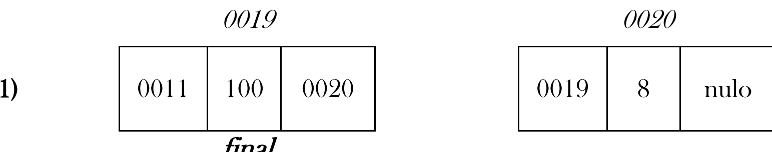


ELIMINACIÓN AL FINAL (*EliminarFin)*

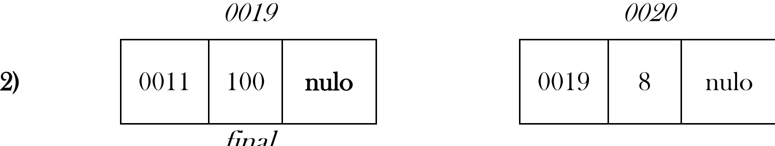


Se asigna:

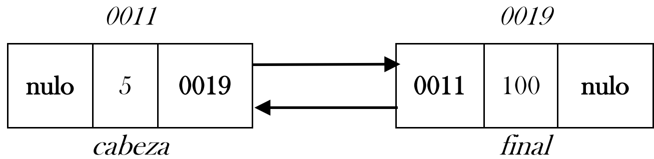
- a *final* la dirección del apuntador al *anterior* de *final* y se elimina la liga del primer elemento.



- al apuntador de *siguiente* del ahora *final* el valor de nulo y se “rompe” con la liga al elemento.

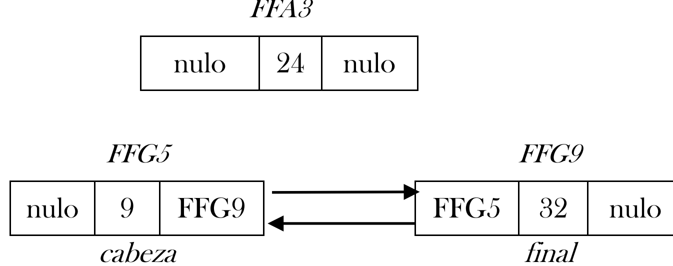


La lista final es

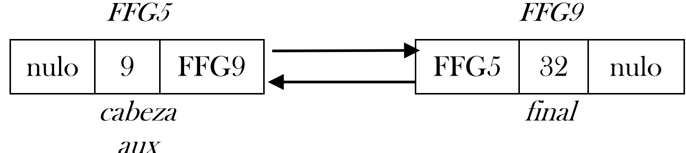


INSERCIÓN ENTRE DOS ELEMENTOS (*INSERTAR2*)

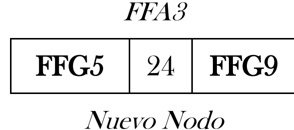
El nodo *FFA3* se inserta entre *FFG5* y *FFG9*



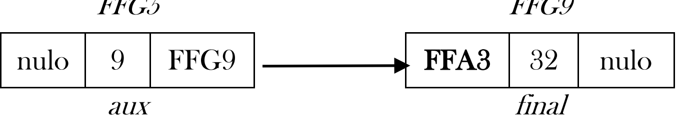
1) Se recorre la lista hasta un elemento antes de la posición en que se quiere insertar el nodo nuevo, esto se realiza con una variable auxiliar a la cual se le debió asignar previamente la dirección de *cabeza.* (En este ejemplo *aux* permanece con la dirección de *cabeza* ya que éste es una posición antes del lugar donde se desea insertar el nuevo elemento)



2) Posteriormente se realizan las asignaciones siguientes:
   - Al apuntador de *anterior* del nodo nuevo se le asigna la dirección de *aux*.
   - Al apuntador de *siguiente* del nuevo nodo se le asigna la dirección del apuntador de *siguiente* de *aux*.

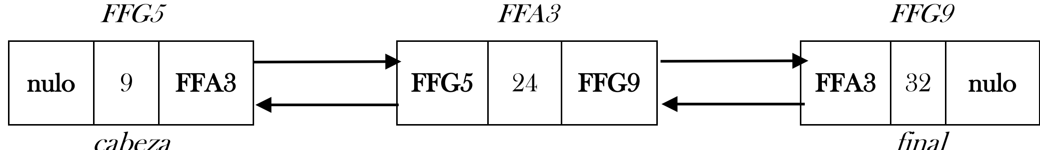


   - Al apuntador de *siguiente* de *anterior* de *aux* (FFG9) se le asigna la dirección del nuevo nodo.

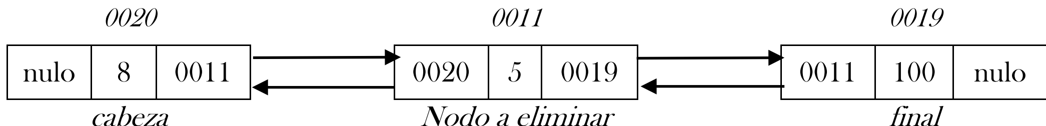


   - Al apuntador de *siguiente* de *aux* se le asigna la dirección del nuevo nodo.

La lista final es

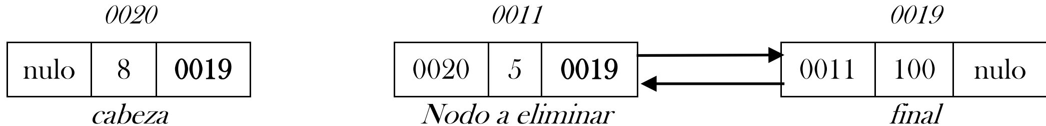


Eliminación entre dos elementos (*Eliminar2*)
El nodo a eliminar es el que tiene la dirección *0011* del siguiente esquema

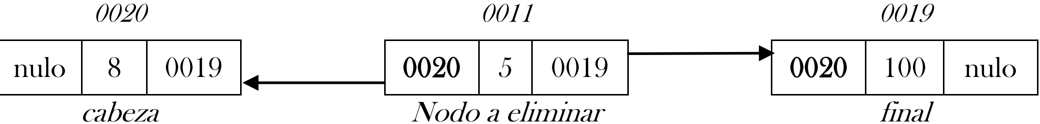


1) Se recorre la lista hasta encontrar el elemento a eliminar

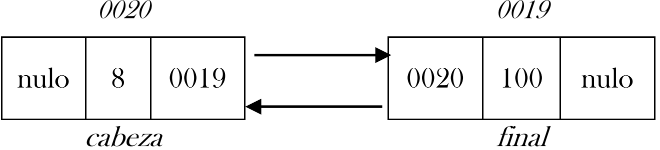
2) Se realizan las asignaciones siguientes:
   - Al apuntador de *anterior* de *siguiente* del nodo a eliminar se le asigna el apuntador de *siguiente* del mismo nodo



   - Al apuntador de *siguiente* de *anterior* del nodo a eliminar se le asigna el apuntador de *anterior* del mismo nodo



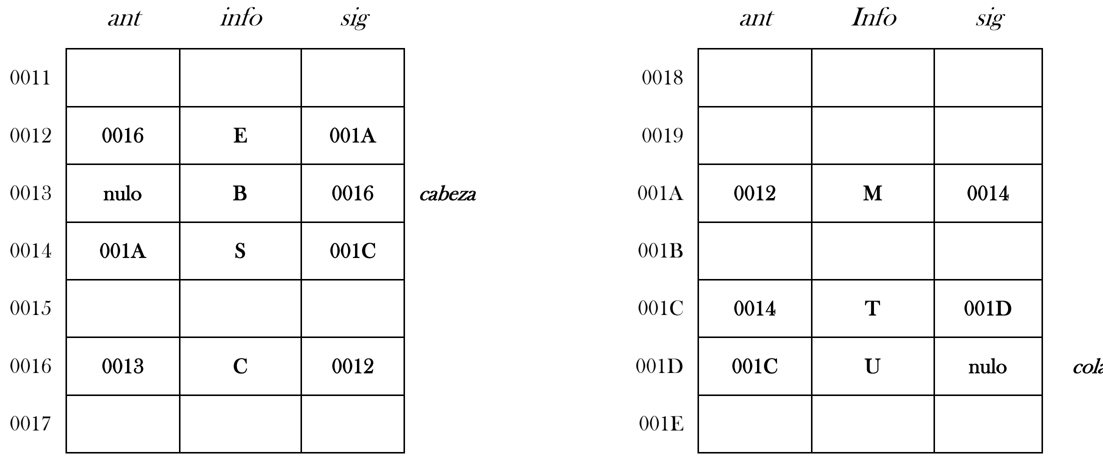
La lista final es



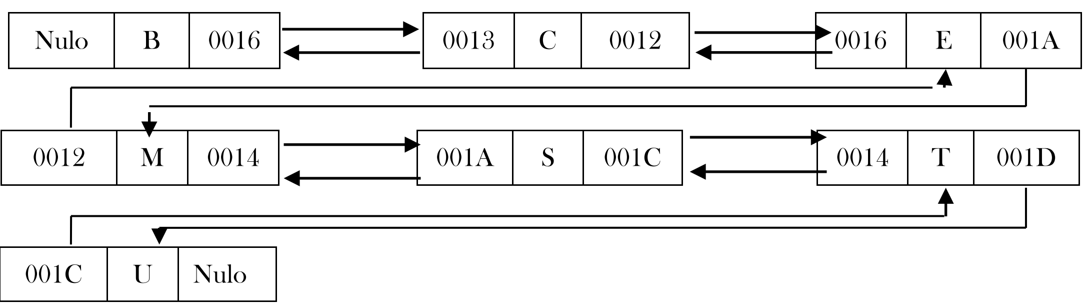
Ejercicio Nº 9

Imagina que la siguiente cuadrícula es la memoria de la computadora en la cual se almacenan los nodos de una lista simplemente ligada, simular el conjunto de operaciones de la siguiente tabla indicando el resultado final en el cuadriculado de la memoria realizando la tabulación de una lista doblemente enlazada, asimismo el esquema final e identificar el lugar de *cabeza y cola.* Considerar que para cualquier inserción ésta debe ser ordenada, eso implica identificar qué tipo de inserción se debe realizar asimismo el tipo de eliminación.

Operación	Nodo	
	Dirección	Valor
Insertar	0012	A
Insertar	0019	G
Insertar	0018	J
Insertar	001A	M
Insertar	0011	R
Eliminar	-	A
Insertar	0013	B
Eliminar	-	J
Insertar	001C	T
Insertar	0014	S
Insertar	001D	U
Eliminar	-	G
Insertar	0012	E
Insertar	0016	C
Eliminar	-	R



ESQUEMA



Implementación de las operaciones básicas de una lista doblemente enlazada.

Las implementaciones de las operaciones de una lista doblemente enlazada se presentan en la Tabla 11, considerando la siguiente declaración.

Variables globales:

Nodo: Registro
**ant: Nodo*
x: E
**sig: Nodo*
Fin Registro

**cabeza: Nodo*
**final: Nodo*

Tabla 11. Implementación de las operaciones de una lista doblemente enlazada

Insertar	InsertarFin
Módulo Insertar (valor:E, *L: Nodo) ***Inicio*** **N: Nodo* *N ← Asignación de memoria* *Si (N ≠ Nulo) entonces* *Si (L = Nulo) entonces* *N->x ← valor* *N->sig ← Nulo* *N->ant ← Nulo* *L ← N* *cabeza ← L* *final ← L* *Otro* *InsertarFin(valor,N)* *FinSi* *Otro* *Escribir("Error en Memoria")* *FinSi* ***Termina***	***Módulo InsertarFin (valor:E, *N: Nodo)*** ***Inicio*** *N->x ← valor* *N->sig ← Nulo* *N->ant ← final* *final-> sig ← N* *final ← N* ***Termina***
InsertarPrim	**Eliminar**
Módulo InsertarPrim (valor:E, *N: Nodo) ***Inicio*** *N->x ← valor* *N->ant ← Nulo* *N->sig ← cabeza* *cabeza-> anterior ← N* *cabeza ← N* ***Termina***	***Módulo Eliminar (valor:E, *L:Nodo)*** ***Inicio*** **aux: Nodo* *Si (L ≠ Nulo) entonces* *aux ← Buscar(valor)* *Si (aux ≠ Nulo) entonces* *Si (aux = cabeza) entonces* *EliminarPrim()* *Otro* *Si (aux = final) entonces* *EliminarFin()* *Otro* *Eliminar2(aux)* *FinSi* *FinSi* *Otro* *Escribir ("Elemento no encontrado")* *FinSi* *Otro* *Escribir("Lista Vacía")* *FinSi* ***Termina***
EliminarPrim	**EliminarFin**
Módulo EliminarPrim () ***Inicio*** **aux: Nodo* *aux ← cabeza* *cabeza ← cabeza->sig* *cabeza->anterior ← Nulo* *Liberar (aux)* ***Termina***	***Módulo EliminarFin ()*** ***Inicio*** **aux: Nodo* *aux ← final* *final ← final->anterior* *final->siguiente ← Nulo* *Liberar (aux)* ***Termina***

EsVacía	ListaVacía
Módulo EsVacía () ***Inicio*** *Si (cabeza = Nulo) entonces* *Regresa 1* *Otro* *Regresa 0* *FinSi* ***Termina***	***Módulo ListaVacía ()*** ***Inicio*** *cabeza ← Nulo* *final ← Nulo* ***Termina***
Ultimo	**Primero**
Módulo Ultimo( ): Nodo ***Inicio*** *Regresa final* ***Termina***	***Módulo Primero( ): Nodo*** ***Inicio*** *Regresa cabeza* ***Termina***
Anterior	**Siguiente**
Módulo Anterior(*N:Nodo): Nodo ***Inicio*** *Regresa N->ant* ***Termina***	***Módulo Siguiente (*N:Nodo): Nodo*** ***Inicio*** *Regresa N->sig* ***Termina***
Buscar	**Visualiza**
Módulo Buscar (valor:E): Nodo ***Inicio*** **aux: Nodo* *aux ← cabeza* *Mientras(aux ≠ nulo)* *Si (aux->x ≠ valor) entonces* *aux ← aux->sig* *Otro* *Regresa aux* *FinSi* *FinMientras* *Regresa nulo* ***Termina***	***Módulo Visualiza()*** ***Inicio*** **N: Nodo* *N ← cabeza* *Mientras (N ≠ Nulo)* *Escribir(N->x)* *N ← N->sig* *FinMientras* ***Termina***
Insertar2	**Eliminar2**
Módulo Insertar2(*N:Nodo, *NIzq:Nodo, *NDer:Nodo) ***Inicio*** *N->sig ← NDer* *N->ant ← NIzq* *NIzq->sig ← N* *NDer->ant ← N* ***Termina***	***Módulo Eliminar2(*N:Nodo)*** ***Inicio*** *N->ant->sig ← N->sig* *N->sig->ant ← N->ant* *Liberar(N)* ***Termina***

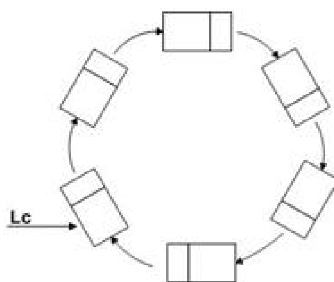
Actividades 35

Resumen

- Una lista doblemente enlazada (LDE) está compuesta por un conjunto de nodos enlazados por dos apuntadores, uno al elemento siguiente y otro al elemento anterior.
- Una LDE está coordinada por dos apuntadores que son *Inicio o Cabeza* y *Final o Cola* de la lista.
- Una LDE está vacía cuando el valor de sus apuntadores *Cabeza* y de *Final* son Nulos.
- El apuntador a *siguiente* del último nodo de una LDE siempre será Nulo.
- El apuntador a *anterior* del primer elemento de una LDE siempre será Nulo.

2.4.5 Lista Circular Simplemente Enlazada

En las listas lineales siempre hay un último elemento que tiene el campo de enlace a *nulo [1]*. En el caso de las listas no lineales, el puntero del último elemento apunta al primero de la lista [3] haciendo de ésta una estructura circular. La cual se representa en la siguiente figura.



Una lista circular por naturaleza no tiene ni principio ni fin [1, 2] sin embargo resulta útil establecer un nodo a partir del cual se accede a la lista y así poder acceder a sus elementos. Todos los recorridos y operaciones de una lista circular se realizan tomando como referencia el último nodo.

2.4.5.1 Representación

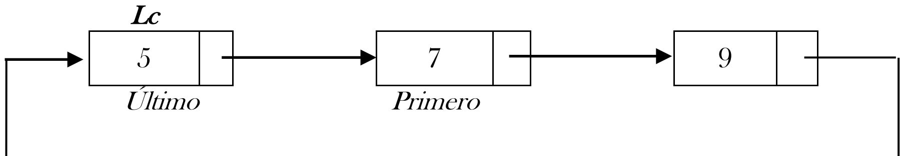
De igual manera el esquema la declaración de un nodo se especifica como:

ListaC: Registro
Info
**sig:ListaC*
Fin Registro

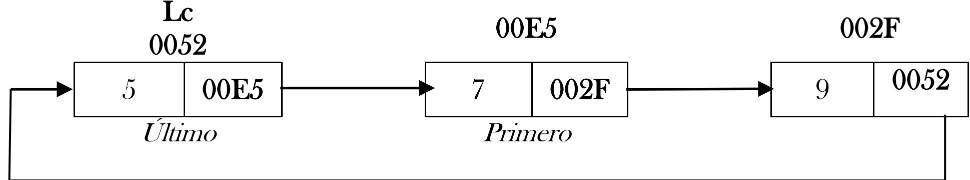
Además de definir siempre un apuntador *cabeza-final (Lc)* que permita identificar el ***último nodo*** insertado de la lista y el siguiente de éste es el primero [1-4].

***Lc: ListaC**

Esta estructura puede esquematizarse de la siguiente manera:



Recordando que a cada nodo se le asigna una dirección de memoria para poder hacer referencia a éste, entonces se tendría:



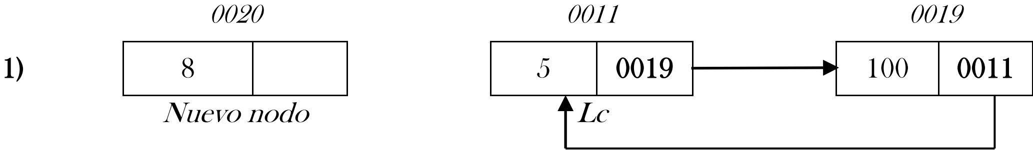
Cuando la lista está vacía el puntero *Lc* debe ser nulo. Cuando es el primer elemento de la lista el apuntador al *siguiente* apuntará a sí mismo.

2.4.5.2 Operaciones básicas

Las operaciones de este tipo de lista son muy similares a las de una lista simplemente ligada o enlazada, por lo que sólo se implementan en este apartado las operaciones que reflejen algún cambio.

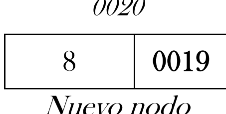
Para poder implementar los algoritmos de las operaciones primero se representan las operaciones de una manera esquemática.

INSERCIÓN AL FINAL (*InsertarFin)*

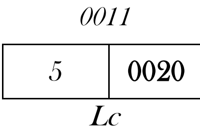


Se realizan las asignaciones siguientes:

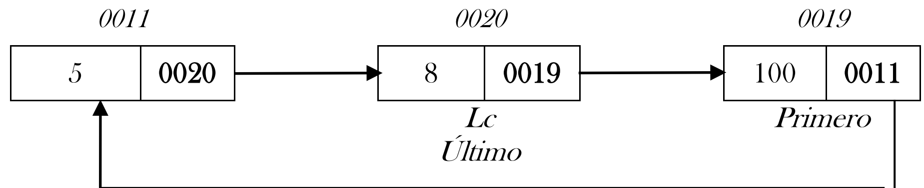
- Al apuntador de *siguiente* del nodo nuevo, la dirección del apuntador *siguiente* de *Lc*



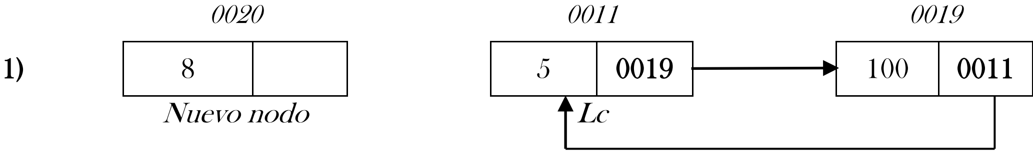
- *A Lc* de *siguiente,* la dirección del nuevo nodo



- Finalmente a *Lc,* la dirección del nodo nuevo.

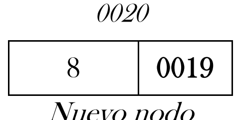


INSERCIÓN AL INICIO (*InsertarPrim)*

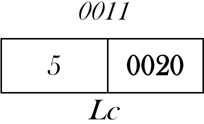


Se realizan las asignaciones siguientes:

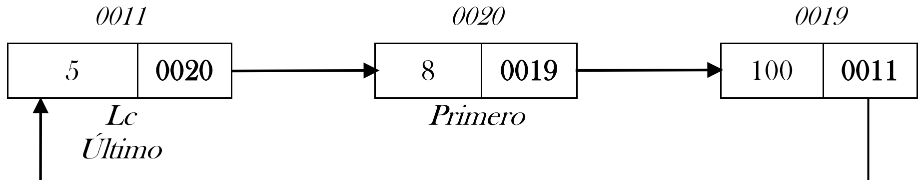
- Al apuntador de *siguiente* del nodo nuevo, la dirección del apuntador *siguiente* de *Lc*



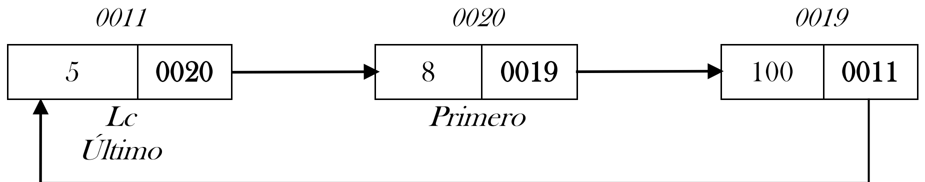
- *A Lc* de *siguiente,* la dirección del nuevo nodo



La lista final es:

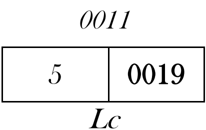


ELIMINACIÓN AL INICIO (*EliminarPrim)*

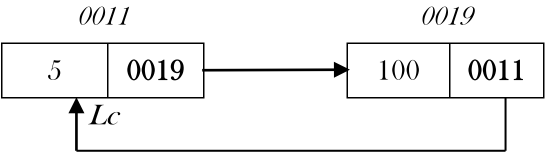


Se asigna:

- Al apuntador *siguiente* de *Lc,* el apuntador de *siguiente* de *siguiente.*



La lista final



Implementación de las operaciones básicas de una lista circular simplemente enlazada.

Considerando lo anterior, en la Tabla 12 se describen las operaciones principales de una lista circular simplemente enlazada que tengan alguna diferencia con las de la lista simplemente enlazada.

ListaC: Registro
x: E
**sig: ListaC*
Fin Registro

***Lc: ListaC**

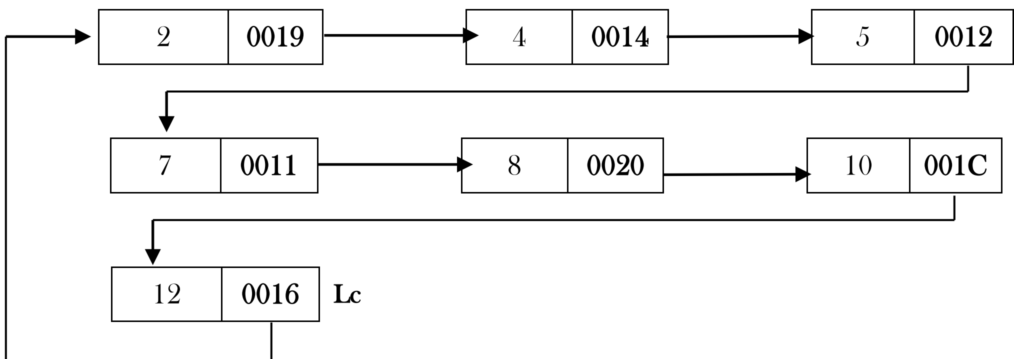
Tabla 12. Operaciones de una lista circular simplemente enlazada

Insertar	InsertarFin
Módulo Insertar (valor:E, *L: ListaC) ***Inicio*** **N: ListaC* *N ← Asignación de memoria* *Si (N ≠ Nulo) entonces* *Si (L = Nulo) entonces* *N->x ← valor* *N->sig ← N* *L ← N* *Lc ← L* *Otro* *InsertarFin(valor,N)* *FinSi* *Otro* *Escribir("Error en Memoria")* *FinSi* ***Termina***	***Módulo InsertarFin (valor:E, *N: ListaC)*** ***Inicio*** *N->x ← valor* *N->sig ← Lc->sig* *Lc-> sig ← N* *Lc ← N* ***Termina***
InsertarPrim	**Eliminar**
Módulo InsertarPrim (valor:E, *N: ListaC) ***Inicio*** *N->x ← valor* *N->sig ← Lc->sig* *Lc-> sig ← N* ***Termina***	***Módulo Eliminar (valor:E, *L:ListaC)*** ***Inicio*** **aux: ListaC* *Si (L ≠ Nulo) entonces* *aux ← Buscar(valor)* *Si (aux ≠ Nulo) entonces* *Si (aux = Lc->sig) entonces* *EliminarPrim()* *Otro* *Si (aux = Lc) entonces* *EliminarFin()* *Otro* *Eliminar2(aux)* *FinSi* *FinSi* *Otro* *Escribir ("Elemento no encontrado")* *FinSi* *Otro* *Escribir("Lista Vacía")* *FinSi* ***Termina***
EliminarPrim	**EliminarFin**
Módulo EliminarPrim () ***Inicio*** **aux: ListaC* *aux ← Lc->sig* *Lc->sig ← Lc->sig->sig* *Liberar (aux)* ***Termina***	***Módulo EliminarFin ()*** ***Inicio*** **aux: ListaC* *aux ← Anterior(Lc)* *aux->sig ← Lc->sig* *Liberar (Lc)* *Lc ← aux* ***Termina***

Ejercicio Nº 10

Imagina que la siguiente cuadrícula es la memoria de la computadora en la cual se almacenan los nodos de una lista circular simplemente enlazada. Realizar el esquema final e identificar el lugar de *Lc.* Considerar que para cualquier inserción ésta debe ser ordenada, eso implica identificar qué tipo de inserción se debe realizar asimismo el tipo de eliminación.

Operación	Nodo	
	Dirección	Valor
Insertar	0012	7
Insertar	0019	4
Insertar	0018	6
Insertar	001A	9
Insertar	0011	8
Eliminar	-	9
Insertar	0013	3
Eliminar	-	6
Insertar	001C	12
Insertar	0014	5
Insertar	001D	1
Eliminar	-	3
Insertar	0020	10
Insertar	0016	2
Eliminar	-	1



Actividad 36 y 37

- **Implementación dinámica de una cola**

Esta implementación es una variante de la realización de listas enlazadas. Al realizar una lista circular se estableció, por conveniencia, que el puntero de acceso a la lista referenciaba al último nodo y que éste al primero [1]. Esta implementación, se basa en definir cada elemento de la cola como un *nodo* de la lista circular simplemente enlazada, cada inserción se realiza en el último nodo referenciado por *Lc* y la eliminación por el siguiente (primer elemento), quedando su declaración de la siguiente manera:

```
Cola: Registro
        dato: tipo de datos
        *sig: Cola
FinRegistro
*Lc: Cola
```

Considerando que:

- *Final:* Es *Lc*
- *Frente: E*s *Lc->sig*

Bajo estas consideraciones, la implementación de las operaciones se presenta en la Tabla 13.

Tabla 13. Operaciones de una cola implementadas con estructuras dinámicas

Operación	Implementación

Insertar	***Modulo Insertar (*c: cola, valor: E)***
	Inicio
	  Si(ColaLlena(c)=1) entonces
	    Escribir("Cola llena")
	  Otro
	    c->dato ← valor
	    c->sig ← Lc->sig
	    Lc->sig ← c
	    Lc ← c
	  FinSi
	Termina

Quitar	***Modulo Quitar(*Lc: cola)***
	Inicio
	  Si (ColaVacia(c) = 1) entonces
	    Escribir ("Cola Vacía")
	  Otro
	    Lc->sig ← Lc->sig->sig
	  FinSi
	Termina

Cola Vacía	***Modulo ColaVacia(*c: cola) : E***
	Inicio
	  Si (Lc = Nulo) entonces
	    Inicializar( )
	    Regresa 1
	  Otro
	    Regresa 0
	  FinSi
	Termina

Inicializar	***Modulo Inicializar()*** *Inicio* *Lc ← Nulo* *Termina*

Frente	***Modulo Frente(*c: cola)***
	Inicio
	  Si (ColaVacia(c) = 1) entonces
	    regresa 0
	  Otro
	    regresa Lc->sig
	  FinSi
	Termina

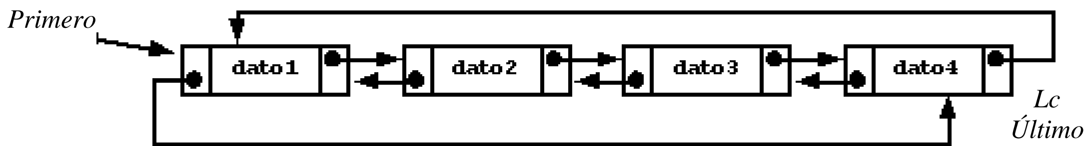
Actividad 38

Resumen

- Una lista circular simplemente enlazada (LCSE) está compuesta por un conjunto de nodos enlazados por un apuntador al elemento siguiente.
- La característica principal de una LCSE es que está coordinada por un apuntador denominado *Lc* que corresponde al último elemento de la lista circular.
- El apuntador *LC* apunta siempre al primer nodo o elemento de una LCSE.
- Considerando *LC* como el apuntador principal de la lista y haciendo una analogía entonces:
    - *Frente = LC->sig*
    - *Final = Lc*
- El primer elemento de la lista siempre apunta a sí mismo.

2.4.6 Lista Circular Doblemente Enlazada

Esta implementación de listas circulares con nodos que tienen dos punteros, permite recorrer la lista circular en sentido del avance del reloj, o bien en sentido contrario [1, 3]. Recordar que en las lista circulares siempre hay un puntero del último elemento apunta al primero de la lista [3] haciendo de ésta una estructura circular. La cual se representa en la siguiente figura.



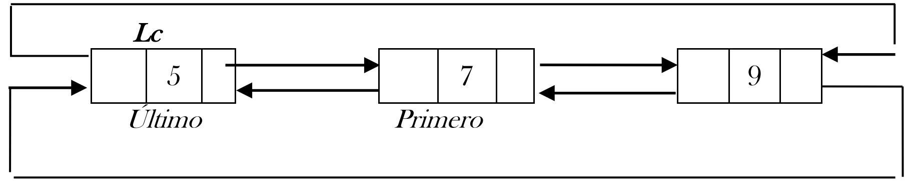
Una lista circular por naturaleza no tiene ni principio ni fin [1, 2] sin embargo resulta útil establecer un nodo a partir del cual se accede a la lista y así poder acceder a sus elementos. Todos los recorridos y operaciones de una lista circular se realizan tomando como referencia el último nodo que es el apuntador *Lc* que permite identificar el ***último nodo*** insertado de la lista y el siguiente de éste es el primero [1-4].

2.4.6.1 Representación

De igual manera el esquema la declaración de un nodo se especifica como:

ListaCD: Registro
**ant: ListaCD*
Info
**sig: ListaCD*
Fin Registro
****Lc: ListaCD***

Esta estructura puede esquematizarse de la siguiente manera:



Cualquier operación que se realice con una lista circular doblemente enlazada debe considerar las ligas de los punteros de siguiente y anterior. Bajo este esquema la implementación de estas operaciones son las presentadas en la Tabla 14.

ListaCD: Registro
**ant: ListaCD*
x: E
**sig: ListaCD*
Fin Registro

****Lc: ListaCD***

Tabla 14. Operaciones de una lista circular doblemente enlazada

Insertar	InsertarFin
Módulo Insertar (valor:E, *L: ListaCD) ***Inicio*** **N: ListaCD* *N ← Asignación de memoria* *Si (N ≠ Nulo) entonces* *Si (L = Nulo) entonces* *N->x ← valor* *N->sig ← N* *N->ant ← N* *L ← N* *Lc ← L* *Otro* *InsertarFin(valor,N)* *FinSi* *Otro* *Escribir("Error en Memoria")* *FinSi* ***Termina***	***Módulo InsertarFin (valor:E, *N: ListaCD)*** ***Inicio*** *N->x ← valor* *N->ant ← Lc* *N->sig ← Lc->sig* *Lc-> sig ← N* *Lc ← N* ***Termina***
InsertarPrim	**Eliminar**
Módulo InsertarPrim (valor:E, *N: ListaCD) ***Inicio*** *N->x ← valor* *N->ant ← Lc* *N->sig ← Lc->sig* *Lc-> sig ← N* ***Termina***	***Módulo Eliminar (valor:E, *L:ListaCD)*** ***Inicio*** **aux: ListaCD* *Si (L ≠ Nulo) entonces* *aux ← Buscar(valor)* *Si (aux ≠ Nulo) entonces* *Si (aux = Lc->sig) entonces* *EliminarPrim()* *Otro* *Si (aux = Lc) entonces* *EliminarFin()* *Otro* *Eliminar2(aux)* *FinSi* *FinSi* *Otro* *Escribir ("Elemento no encontrado")* *FinSi* *Otro* *Escribir("Lista Vacía")* *FinSi* ***Termina***
EliminarPrim	**EliminarFin**
Módulo EliminarPrim () ***Inicio*** **aux: ListaCD* *aux ← Lc->sig* *Lc->sig->ant ← Lc* *Lc->sig ← Lc->sig->sig* *Liberar (aux)* ***Termina***	***Módulo EliminarFin ()*** ***Inicio*** **aux: ListaCD* *aux ← Lc->ant* *Lc->ant->sig ← Lc->sig* *Lc->sig->ant ← Lc->ant* *Liberar (Lc)* *Lc ← aux* ***Termina***

Resumen

- Una lista circular doblemente enlazada (LCDE) está compuesta por un conjunto de nodos enlazados por dos apuntadores, uno al elemento siguiente y otro al elemento anterior.
- La característica principal de una LCDE, al igual que una LCSE, es que está coordinada por un apuntador denominado *Lc* que corresponde al último elemento de la lista circular.
- El apuntador *LC* apunta siempre al primer nodo o elemento de una LCSE.
- Considerando *LC* como el apuntador principal de la lista y haciendo una analogía entonces:
    - *Frente = LC->sig*
    - *Final = Lc*
- El primer elemento de la lista siempre apunta a sí mismo.
- Recordar que el registro que representa al nodo de este tipo de lista está compuesto por campos que representan información y dos campos más que hacen referencia a los elementos *anterior* y *siguiente* de cada uno de los nodos.

REFERENCIAS

1. Joyanes, L. and I. Zahonero, *Estructura de Datos. Algoritmos, Abstracción y Objetos*. 1998, Madrid: McGraw-Hill.
2. Joyanes, L. and I. Zahonero, *Programación en C. Metodología, algoritmos y estructuras de datos*. 2a. ed. 2005, España: Mc Graw Hill.
3. Joyanes, L., et al., *Estructuras de datos en C*. 2005, Madrid: Mc Graw Hill, Serie Schaum.
4. Tenembaum, et al., *Estructuras de datos en C*. 1997, México: Prentice-Hall.
5. Cairó, O. and S. Guardati, *Estructuras de Datos*. 2a. Edición ed. 2002, México: Mc Graw Hill.
6. Albarrán, S. and M. Salgado, *Programación Estructurada*. 2010, México: UAEM.
7. Criado, M.A., *Programación en Lenguajes Estructurados*. 2006, México: Alfaomega.

UNIDAD DE COMPETENCIA III
Aplicar la estructura de datos árbol

20 HRS.
4 SEMANAS

Como se mencionó en las unidades de competencia anteriores, existen 2 tipos de estructuras de datos dinámicas: *las lineales y las no lineales.* Esta unidad se enfoca al uso, manejo y aplicación de las estructuras de datos dinámicas no lineales como son: *los árboles* en sus distintas modalidades.

3.1 RECURSIVIDAD DIRECTA

Un subprograma, procedimiento o función que se llama directamente a sí mismo se dice que es *recursivo* [1, 2], es decir, un objeto recursivo es aquel que forma parte de sí mismo [2].

La recursión puede darse de dos maneras diferentes [1]:

1. ***Directa***: el subprograma se llama directamente a sí mismo.
2. ***Indirecta***: el subprograma llama a otro segundo subprograma y éste a su vez llama al primero.

Un procedimiento o función recursivos deben de cumplir con dos propiedades generales para no dar lugar a un ciclo infinito con las sucesivas llamadas [1, 2]:

- Cumplir una cierta condición o criterio del que dependa la llamada recursiva, es decir, establecer un **estado básico** en el cual la solución no se presente de manera recursiva sino directamente.
- Cada vez que el procedimiento o función se llamen a sí mismos, directa o indirectamente, debe estar más cerca del incumplimiento de la condición de que depende la llamada, es decir, que la entrada (datos) del problema debe ir acercándose al estado básico.

El siguiente ejercicio permite ejemplificar la función de recursión.

Ejercicio N° 1:

El factorial de un número entero positivo se define como el producto de los números comprendidos entre 1 y *n*. La expresión **n!** simboliza el factorial de *n.* por ejemplo:

$0! = 1$
$1! = 1$
$2! = 1 * 2$
$3! = 1 * 2 * 3$
$4! = 1 * 2 * 3 * 4$
$n! = 1 * 2 * \ldots * (n-1) * n$

En este ejercicio el estado básico es $n = 0$ entonces el factorial sería 1, para cualquier otro valor mayor que 0 estaría dado el factorial como $n * (n-1)!$, esto se puede ver como:

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

Finalmente el algoritmo recursivo para el cálculo del factorial de un número es:

```
Módulo Factorial ( N:E ): E
Inicia
        Si (N = 0) entonces
                Factl ← 1               /*estado base*/
        Otro
                Fact ← N * Factorial(N-1)
        FinSi
        Regresa Fact
FinMódulo
```

La prueba de escritorio de este algoritmo se puede representar como pila de instrucciones. Si se considera a **N=5** la pila de instrucciones sería

Factorial(0)
Factorial(1)
Factorial(2)
Factorial(3)
Factorial(4)
Factorial(5)
Principal

Lo anterior, se detuvo en el *Factorial(0)* debido a que el estado base es N=0, de ahí empiezan a realizarse los cálculos de forma inversa (como la pila LIFO). Lo cual significa que se ejecuta primero el factorial de 0, después el de 1, y así sucesivamente.

Factorial(0)	*regresa 1*	
Factorial(1)	*regresa 1*1*	
Factorial(2)	*regresa 1*2*	
Factorial(3)	*regresa 2*3*	
Factorial(4)	*regresa 6*4*	
Factorial(5)	*regresa 24*5*	*factorial = 120*
Principal		

Un ejemplo de la codificación de este algoritmo recursivo en lenguaje de programación C es:

```
int Factorial(int N){
        int fact;

        if (N == 0) fact = 1;
        else{
                fact = N*Factorial(N-1);
        }
        return fact;
}
```

Actividad 39

Resumen

- Recursividad es cuando un subprograma, procedimiento o función se llama directamente a sí mismo.
- La recursividad permite ejecutar de una manera más rápida los programas o módulos que la utilizan.
- Un módulo recursivo debe contar siempre con un estado básico en el cual la solución no se presente de manera recursiva sino directamente y permita terminar con la recursión del módulo.

3.2 ÁRBOLES: USOS, CARACTERÍSTICAS, REPRESENTACIÓN Y CONSTRUCCIÓN

El concepto de *árbol* implica una estructura en la que los datos se organizan de modo que los elementos de información están relacionados entre sí a través de ramas [3], surgiendo el concepto de estructura de ramificación entre nodos [1]. El árbol genealógico es un ejemplo de éstos.

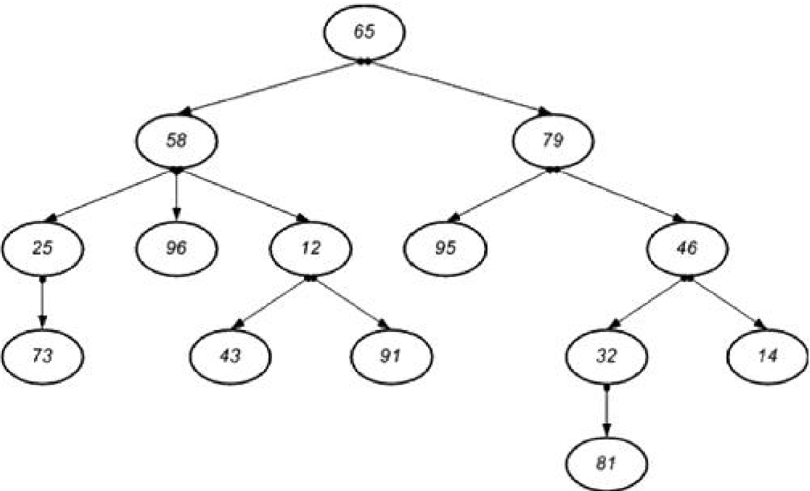
Las estructuras de tipo árbol se usan para representar datos con una relación jerárquica entre sus elementos [3].

Los árboles son las estructuras de datos más importantes en computación [1-4]. Se pueden usar para representar fórmulas matemáticas, para organizar adecuadamente la información, para construir un árbol genealógico, para el análisis de circuitos eléctricos [1]; como método eficiente para búsquedas grandes y complejas en aplicaciones diversas de inteligencia artificial y algoritmos de cifrado; los sistemas operativos almacenan sus archivos en árboles, en diseño de compiladores y procesadores de texto [3].

3.2.1 Concepto de árbol

Un *árbol* es una estructura jerárquica aplicada a una colección de elementos u objetos llamados *nodos.* Uno de los cuales es conocido como *raíz,* creando una relación con los demás elementos lo cual da lugar a términos como *padre, hijo, hermano, antecesor, sucesor, ancestro, etc. [1].*

Esquemáticamente la siguiente figura es la representación más usual y común de un árbol.



3.2.2 Características y propiedades de los árboles

Las características, propiedades y terminología más importantes de un árbol son los presentados en la Tabla 15 [1, 3]:

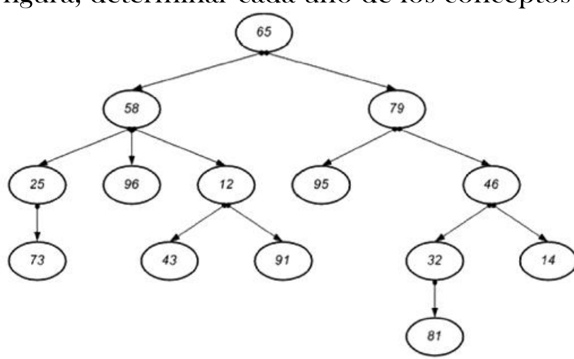
Tabla 15: Terminología de un árbol

Término	Descripción
Nodos	Elementos o *vértices* de un árbol
Raíz	Todo árbol que no es vacío, tiene un único nodo raíz, del cual descienden los demás elementos del árbol
Padre	Antecesor o ascendiente de un nodo, excepto nodo *raíz*
Hijos	Descendientes de un nodo, puede ser varios.
Grado	Número de hijos que salen de un nodo, es decir el número de descendientes directos
Nodo terminal u hoja	Todo nodo que no tiene ramificaciones (hijos) o con grado 0
Hermanos	Todos los nodos que son descendientes directos de un mismo nodo
Nivel	Número de antecesores que tiene un nodo desde la raíz, considerando que el nivel de la raíz es 1.
Profundidad o altura	Es el máximo de los niveles de los nodos de un árbol.
Peso de un árbol	Es el número de nodos terminales.
Nodo interior	Todo nodo que no es raíz, ni terminal u hoja.
Grado de un árbol	Es el máximo grado de todos los nodos del árbol.

Para un mayor entendimiento de la terminología de un árbol, a continuación se realiza el ejercicio 2.

Ejercicio N° 2:

Haciendo referencia a la figura, determinar cada uno de los conceptos de la Tabla 15.



Término	Descripción	Algunos Resultados
Nodos	Elementos o *vértices* de un árbol	**65, 58, 79, 25, 96, 12, 95, 46, 73, 43, 91, 32, 14, 81**
Raíz	Todo árbol que no es vacío, tiene un único nodo raíz, del cual descienden los demás elementos del árbol	**65**
Padre	Antecesor o ascendiente de un nodo, excepto nodo *raíz*	**65** es padre de **58** **58** es padre de **96** **46** es padre de **14** **12** es padre de **91**
Hijos	Descendientes de un nodo, puede ser varios.	**81** es hijo de **32** **73** es hijo de **25**

Término	Descripción	Algunos Resultados
		79 es hijo de **65** **43** es hijo de **12**
Grado	Número de hijos que salen de un nodo, es decir el número de descendientes directos	El grado de **46** es **2** El grado de **58** es **3** El grado de **25** es **1** El grado de **12** es **2**
Nodo terminal u hoja	Todo nodo que no tiene ramificaciones (hijos) o con grado 0	**73, 96, 43, 91, 95, 81, 14**
Hermanos	Todos los nodos que son descendientes directos de un mismo nodo	**58** y **79** son hermanos **95** y **46** son hermanos **43** y **91** son hermanos **25, 96** y **12** son hermanos
Nivel	Número de antecesores que tiene un nodo desde la raíz, considerando que el nivel de la raíz es 1.	El nivel de **46** es **3** El nivel de **81** es **5** El nivel de **91** es **4** El nivel de **79** es **2**
Profundidad o altura	Es el máximo de los niveles de los nodos de un árbol.	**5**
Peso de un árbol	Es el número de nodos terminales.	**7**
Nodo interior	Todo nodo que no es raíz, ni terminal u hoja.	**58, 79, 25, 12, 46, 32**
Grado del árbol	Es el máximo grado de todos los nodos del árbol.	**3**

Actividad 40

Resumen

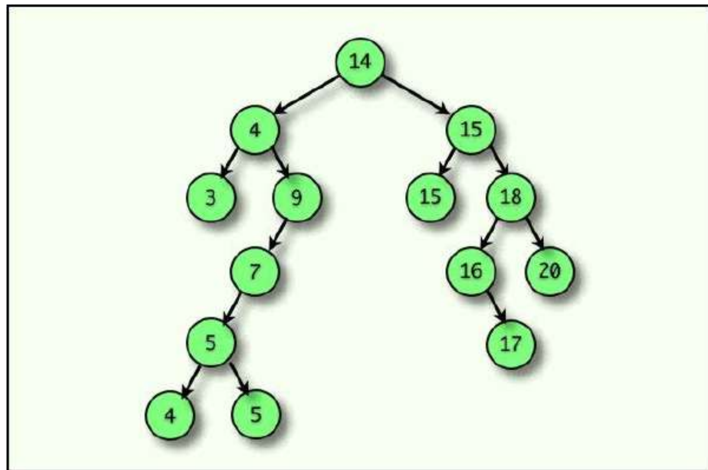
- Un *árbol* es una estructura jerárquica aplicada a una colección de elementos u objetos llamados *nodos*. Uno de los cuales es conocido como *raíz,* creando una relación con los demás elementos lo cual da lugar a términos como *padre e hijo*.
- *Padre* es el nodo que tiene nodos *Hijos*.
- Cada nodo está compuesto por dos partes, una de información y otra de los enlaces o apuntadores de los nodos hijos.
- El nodo principal de un árbol se denomina *Raíz*.
- El grado de un árbol representa el número máximo de hijos que puede tener un padre o nodo.

3.3 ÁRBOLES BINARIOS (REPRESENTACIÓN. RECORRIDO EN PREORDEN, INORDEN, POSTORDEN)

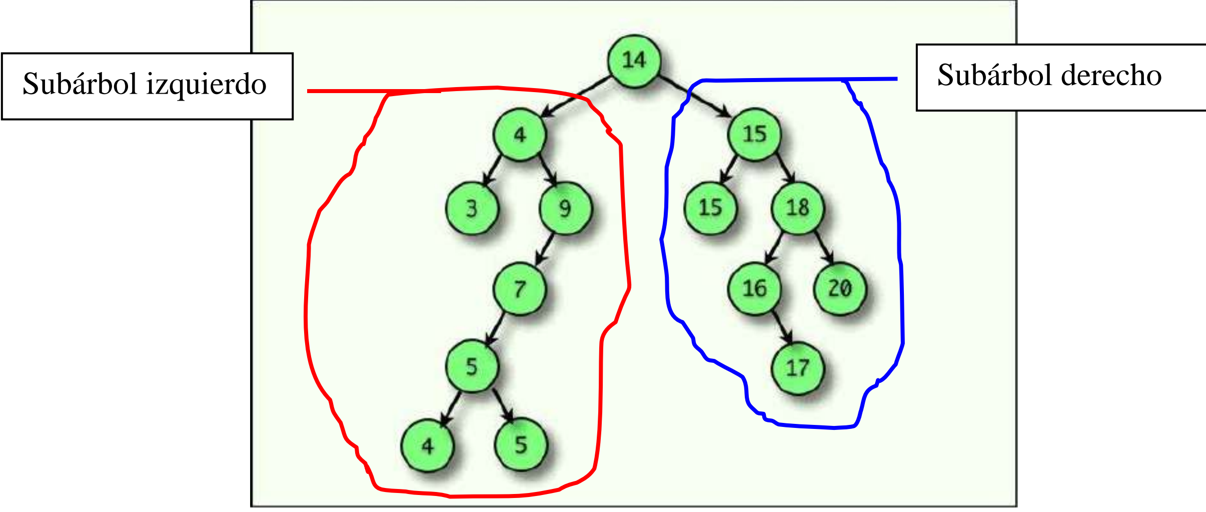
Un *árbol binario* es aquél en el que cada uno de sus nodos no puede tener más de dos nodos hijos, es decir, dos subárboles identificados como *izquierdo y derecho* respectivamente [1-3], es un árbol que tiene como máximo el grado 2.

Un *árbol binario* es un conjunto de nodos que constan de un nodo ***raíz*** enlazado a dos árboles binarios disjuntos (que ningún nodo puede estar en ambos subárboles) denominados **subárbol izquierdo** y **subárbol derecho**.

Su representación esquemática se muestra en la siguiente figura.



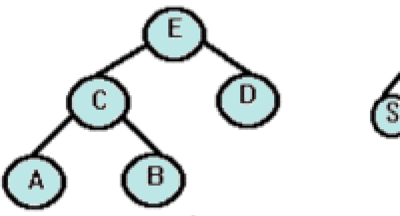
La terminología de un árbol binario es la misma para que para cualquier árbol, sólo se integran dos términos más: ***subárbol izquierdo y subárbol derecho***, los cuales se identifican en la siguiente figura.



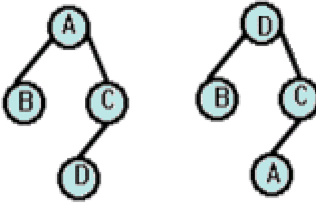
3.3.1 Árboles binarios distintos, similares y equivalentes

Dos árboles son:

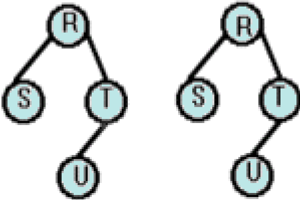
- *Distintos:* cuando sus estructuras son diferentes.



- *Similares:* cuando sus estructuras son idénticas pero la información que contienen sus nodos difieren entre sí.

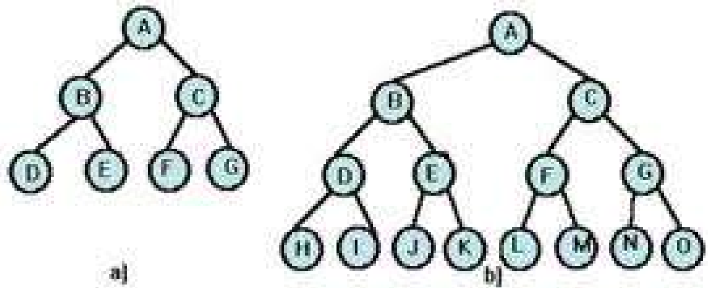


- *Equivalentes:* aquellos que son similares y además los nodos contienen la misma información.



3.3.2 Árbol binario completo

Es un árbol en el que todos sus nodos excepto los del último nivel tienen dos hijos.



Actividad 41

3.3.3 Implementación de un árbol binario

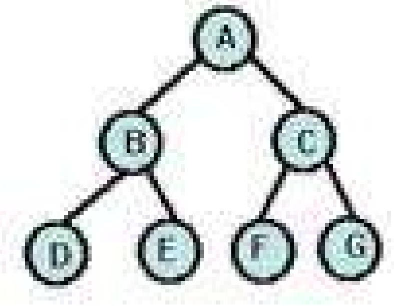
Los nodos de un árbol binario están principalmente representados por registros, los cuales contienen como mínimo tres campos: *Izq, Info y Der*.

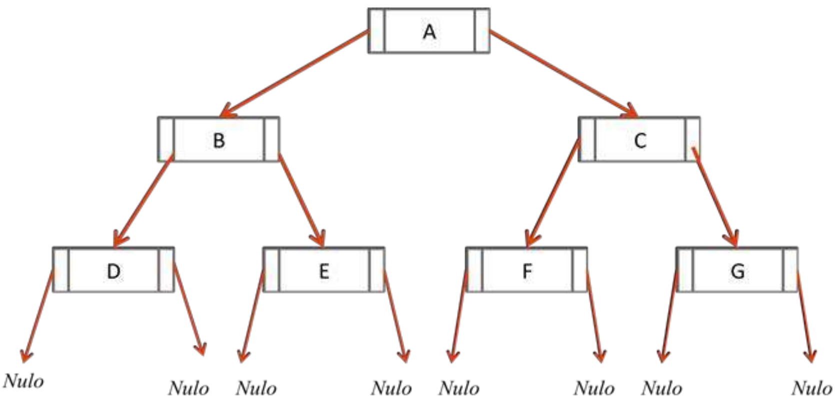
Izq	Info	Der

en lenguaje algorítmico es:

Nodo: Registro
**Izq: Nodo*
Info: tipo de dato
**Der: Nodo*
Fin Registro

Por ejemplo, la representación de la siguiente figura con la definición del nodo anterior sería:





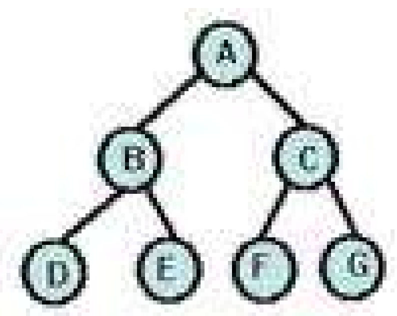
3.3.4 Recorridos

Se denomina ***recorrido*** al proceso que permite acceder una sola vez a cada uno de los nodos del árbol. Existen diferentes formas de hacer el recorrido de un árbol binario, en *anchura* y en *profundidad [3]*.

3.3.4.1 Recorrido en Anchura

Consiste en recorrer los distintos niveles y, dentro de cada nivel, los diferentes nodos de izquierda a derecha. Por ejemplo, de la figura siguiente el recorrido en *anchura* sería:

A B C D E F G



3.3.4.2 Recorrido en Profundidad

El árbol puede ser recorrido en diferentes órdenes, de acuerdo al momento en que visita su *raíz*. De esta manera, se clasifican en tres tipos de recorridos:

- *PreOrden:* **Raíz**- Izquierdo-Derecho (**R**ID)
- *InOrden*: Izquierdo-**Raíz**-Derecho (I**R**D)
- *PostOrden*: Izquierdo-Derecho-**Raíz** (ID**R**)

Donde Izquierdo y Derecho se refieren a los subárboles de cada nueva raíz.

Por ejemplo, de la figura anterior, los nodos visitados de acuerdo a su recorrido son:

PreOrden (RID): A B D E C F G
InOrden (IRD): D B E A F C G
PostOrden (IDR): D E B F G C A

Actividad 42

3.3.4.3 Implementación de los Recorridos en profundidad

La implementación de los recorridos de un árbol se presenta en la Tabla 16.

Tabla 16: Implementación de recorridos en profundidad

PreOrden	EnOrden	PostOrden
Módulo PreOrden (*N: Nodo) ***Inicia*** *Si (N ≠ Nulo) entonces* *Escribir (N->info)* *PreOrden(N->Izq)* *PreOrden(N->Der)* *FinSi* ***FinMódulo***	***Módulo InOrden (*N: Nodo)*** ***Inicia*** *Si (N ≠ Nulo) entonces* *InOrden(N->Izq)* *Escribir (N->info)* *InOrden(N->Der)* *FinSi* ***FinMódulo***	***Módulo PostOrden (*N: Nodo)*** ***Inicia*** *Si (N ≠ Nulo) entonces* *PostOrden(N->Izq)* *PostOrden(N->Der)* *Escribir (N->info)* *FinSi* ***FinMódulo***

3.3.5 Árbol binario de expresión

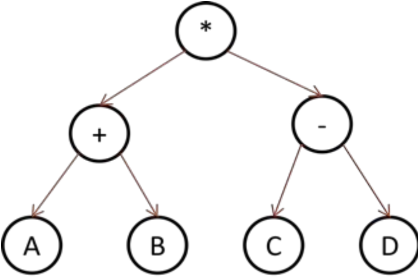
Una de las principales aplicaciones de un árbol binario es la de almacenar expresiones aritméticas en memoria. Ésta se puede representar considerando que toda expresión tiene operadores y operandos, considerando esto, el ***operador*** se representa en la *raíz* y los ***operandos*** en los nodos *izquierdo* y *derecho* según sea la expresión, lo cual quiere decir que

los árboles de expresión son árboles binarios, cuyas hojas contienen operandos y los otros nodos operadores.

Por ejemplo, la siguiente expresión

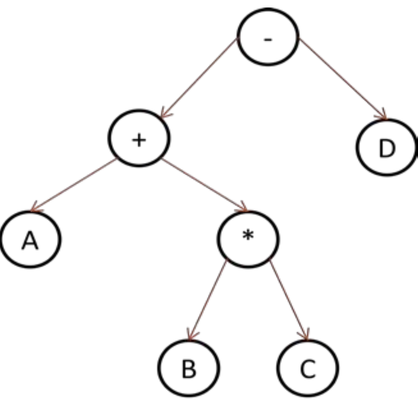
$$(A + B) * (C - D)$$

Tiene su presentación en árbol:



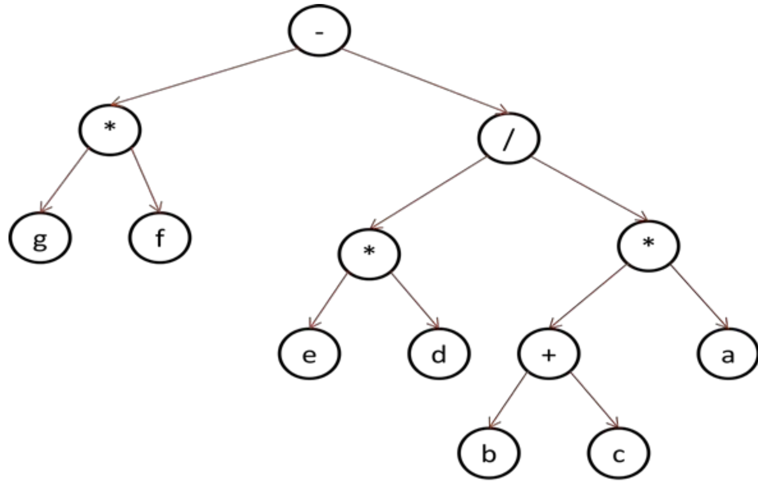
Esto se realiza utilizando la procedencia de los operadores y la asociación de los paréntesis, lo que significa que la siguiente expresión (sin paréntesis) no es lo mismo que la anterior.

$$A + B * C - D$$



Así como a partir de una expresión se obtiene el árbol, se puede también de un árbol obtener su expresión.

Ejercicio Nº 3:
Se tiene el siguiente árbol, obtener su expresión.



Primero se parte de la raíz principal, en este caso **-**, después rama izquierda y luego derecha, considerando primero nuevamente la raíz de cada subárbol, esto es:

Paso	Árbol o subárbol	Expresión
Realizando por partes:		
1		(g*f) -
2		(e*d) /
3		(b+c) *
Uniendo las partes anteriores:		
4		(b+c)* a

Paso	Árbol o subárbol	Expresión
5		(e*d) / [(b+c)*a]
6		(g*f) – [(e*d) / [(b+c)*a]]
	Expresión final:	**(g*f) – [(e*d) / [(b+c)*a]]**

Ejercicio N° 4:

Haciendo referencia a las expresiones obtener el árbol o viceversa según sea el caso.

Expresión	Árbol
[(A+B) – (C*D)] / (E-F)	
a*(b+c) / (d*e) – (f*g)	

Actividad 43

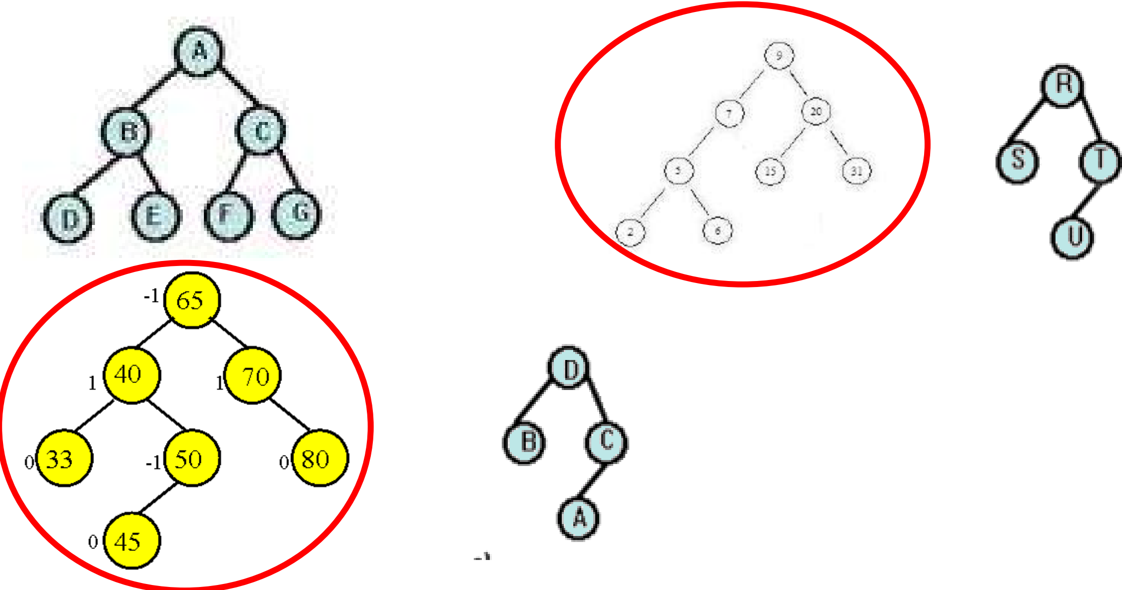
Resumen

- Un *árbol binario* es un árbol de grado de 2.
- Cada nodo padre de un árbol binario tiene a su vez dos subárboles, uno izquierdo y otro derecho.
- Para poder visitar cada uno de los nodos de un árbol se hace uso de los recorridos de profundidad y anchura.
- El recorrido de profundidad hace referencia a los recorridos de orden, preorden y postorden.
- Un recorrido por anchura se realiza visitando los nodos, nivel por nivel del árbol.
- Una de las aplicaciones principales de un árbol binario es la de un árbol de expresión.

3.4 ÁRBOLES BINARIOS DE BÚSQUEDA (OPERACIONES – INSERCIÓN, ELIMINACIÓN, BÚSQUEDA).

Un árbol binario de búsqueda es aquel en el que para cualquier nodo su valor es *superior* a los valores de los nodos de su subárbol izquierdo e *inferior* a los de su subárbol derecho [3], esto significa que este tipo de árbol es un árbol bien ordenado [2].

Ejercicio N° 5:
Con base en la definición anterior, encerrar en un círculo los árboles que son binarios de búsqueda.



✍ **Actividad 44**

3.4.1 Operaciones: inserción, eliminación y búsqueda

La utilidad de este tipo de árbol se refiere a la eficiencia en la búsqueda de un nodo, similar al método de búsqueda binaria, y a la ventaja que presentan los árboles sobre los arreglos en cuanto a que la inserción y borrado de un elemento no requiere el desplazamiento de todos los demás [3].

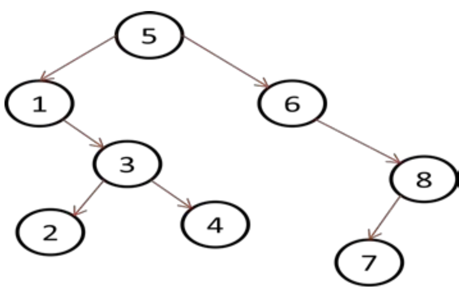
3.4.1.1 Búsqueda de un nodo

La búsqueda de un elemento comienza en el nodo *raíz* y sigue los siguientes pasos: [3]

A. El dato buscado se compara con el dato del nodo raíz.
B. Si los datos son iguales, la búsqueda se detiene.
C. Si el dato buscado es mayor que el dato de la raíz, la búsqueda se reanuda en el subárbol derecho. Si el dato buscado es menor, la búsqueda se reanuda en el subárbol izquierdo.

Ejercicio N° 6:

Marcar el recorrido del árbol con base en el elemento buscado considerando el esquema siguiente.



Buscar	Instrucción	Recorrido
4	4 se compara con 5 4 < 5 entonces subárbol izquierdo 4 se compara con 1 4 > 1 entonces subárbol derecho 4 se compara con 3 4 > 3 entonces subárbol derecho 4 se compara con 4 4 =4 entonces se detiene la búsqueda	
8	8 se compara con 5 8 > 5 entonces subárbol derecho 8 se compara con 6 8 >6 entonces subárbol derecho 8 se compara con 8 8 =8 entonces se detiene la búsqueda	
2	2 se compara con 5 2 < 5 entonces subárbol izquierdo 2 se compara con 1 2 >1 entonces subárbol derecho 2 se compara con 3 2 < 3 entonces subárbol izquierdo 2 se compara con 2 2 =2 entonces se detiene la búsqueda	

Actividad 45

3.4.1.2 Inserción de un nodo

El algoritmo que se sigue para la inserción de un nodo es: [3]

A. Comparar el dato del elemento a insertar con el dato del nodo raíz, si es mayor avanzar hacia el subárbol derecho, si es menor hacia el izquierdo.
B. Repetir el paso anterior hasta encontrar un elemento con el dato igual o llegar al final del subárbol donde debiera situarse el nuevo elemento.
C. Cuando se llega al final es porque no se ha encontrado, por tanto se deberá reservar memoria para una nueva estructura de nodo, introducir los datos y asignar nulo a los punteros izquierdo y derecho del mismo. A continuación se colocará el nuevo nodo como hijo izquierdo o derecho el anterior según sea el valor del dato.

Ejercicio N° 7:

Del siguiente conjunto de números obtener su árbol binario de búsqueda.

5 1 3 6 4 8 2 7

Insertar	Instrucción	Recorrido
5	**Primer elemento,** entonces nodo central	
1	**1 se compara con 5** **1 < 5** entonces subárbol izquierdo No más elementos se **inserta 1**	
3	**3 se compara con 5** **3 < 5** entonces subárbol izquierdo **3 se compara con 1** **3 > 1** entonces subárbol derecho No más elementos se **inserta 3**	
6	**6 se compara con 5** **6 > 5** entonces subárbol derecho No más elementos se **inserta 6**	
4	**4 se compara con 5** **4 < 5** entonces subárbol izquierdo **4 se compara con 1** **4 > 1** entonces subárbol derecho **4 se compara con 3** **4 > 3** entonces subárbol derecho No más elementos se **inserta 4**	

Insertar	Instrucción	Recorrido
8	**8 se compara con 5** **8 > 5** entonces subárbol derecho **8 se compara con 6** **8 > 6** entonces subárbol derecho No más elementos se **inserta 8**	
2	**2 se compara con 5** **2 < 5** entonces subárbol izquierdo **2 se compara con 1** **2 > 1** entonces subárbol derecho **2 se compara con 3** **2< 3** entonces subárbol derecho No más elementos se **inserta 2**	
7	**7 se compara con 5** **7 > 5** entonces subárbol derecho **7 se compara con 6** **7 > 6** entonces subárbol derecho **7 se compara con 8** **7 < 8** entonces subárbol izquierdo No más elementos se **inserta 7**	

Actividad 46

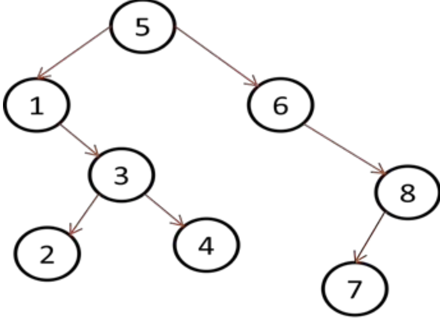
3.4.1.3 Eliminación de un nodo

La eliminación o borrado de un nodo, después de haberlo encontrado requiere considerar las siguientes posibilidades: [1, 3]

A. Que no tenga hijos, es decir que sea hoja.
   a. Se suprime, asignando nulo al puntero de su antecesor que lo apuntaba a éste.
B. Que tenga un único hijo.
   a. El elemento anterior se enlaza con el hijo del que se requiere borrar.
C. Que tenga dos hijos.
   a. Se sustituye por el elemento más próximo inmediato superior o inmediato inferior.
   b. Para localizar estos elementos debe el elemento eliminado sustituirse por el nodo que se encuentra más a la izquierda en el subárbol derecho o por el nodo que se encuentra más a la derecha en el subárbol izquierdo.

Ejercicio N° 8:

Con base en el siguiente esquema eliminar los nodos que se piden en la tabla que a continuación se presenta, considerar que las eliminaciones son instrucciones consecutivas, es decir que se basan en el esquema nuevo que se va generando.



Eliminar	Instrucción	Recorrido
7	**Se Busca 7** **Caso A**: No tiene hijos, sólo se suprime	
6	**Se Busca 6** Con base en el esquema generado en la eliminación anterior. **Caso B**: Un único hijo, ser recorre.	
5	**Se Busca** *5* Con base en el esquema generado en la eliminación anterior. **Caso C**: Dos hijos, se sustituye por el más a la derecha del subárbol izquierdo.	

Actividad 47

3.4.2 Implementación de las operaciones de inserción, eliminación y búsqueda en un árbol binario de búsqueda.

La implementación de los algoritmos de las operaciones de inserción, eliminación y búsqueda se presentan en la Tabla 17, basadas en la declaración del siguiente nodo.

Nodo: Registro
**I: Nodo*
Num: E
**D: Nodo*
FinRegistro

**N: Nodo*

Tabla 17: Implementación de las operaciones de inserción, eliminación y búsqueda.

Operación	Implementación
Búsqueda	***Módulo Buscar (*N: Nodo, dato:E)*** ***Inicia*** *Si (N ≠ Nulo) entonces* *Si (dato < N->Num) entonces* *Buscar (N->I, dato)* *Otro* *Si (dato > N->Num) entonces* *Buscar (N->D, dato)* *Otro* *Escribir ("Elemento Encontrado")* *FinSi* *FinSi* *Otro* *Escribir ("Elemento No Encontrado")* *FinSi* ***FinMódulo***
Inserción	***Módulo Insertar (*N: Nodo, dato:E)*** ***Inicia*** *Si (N ≠ Nulo) entonces* *Si (dato < N->Num) entonces* *Insertar (N->I, dato)* *Otro* *Si (dato > N->Num) entonces* *Insertar (N->D, dato)* *Otro* *Escribir ("El Nodo ya se encuentra")* *FinSi* *FinSi* *Otro* *Crea (Nuevo)* *Nuevo->I ← Nulo* *Nuevo->D ← Nulo* *Nuevo->Num ← dato* *N ← Nuevo* *FinSi* ***FinMódulo***
Eliminación	***Módulo Borrar (*N: Nodo)*** ***Inicia***

Operación	Implementación
	Si (N ≠ Nulo) entonces *Si (dato < N->Num) entonces* *Borrar (N->I, dato)* *Otro* *Si (dato > N->Num) entonces* *Borrar (N->D, dato)* *Otro* *Otro ← N* *Si (Otro->D = Nulo) entonces* *N ← Otro->I* *Otro* *Si (Otro->I = Nulo) entonces* *N ← Otro->D* *Otro* *Aux ← N->I* *Band ← 0* *Mientras (Aux->D ≠ Nulo)* *Aux1 ← Aux* *Aux ← Aux->D* *Band ← 1* *FinMientras* *N->Num ← Aux->Num* *Otro ← Aux* *Si (Band = 1) entonces* *Aux1->D ← Aux->I* *Otro* *N->I ← Aux->I* *FinSi* *FinSi* *FinSi* *Libera memoria (Otro)* *FinSi* *FinSi* *Otro* *Escribir ("Elemento No Encontrado")* *FinSi* ***FinMódulo***

Actividad 48

Resumen

- Un árbol binario de búsqueda es aquel en el que para cualquier nodo su valor es *superior* a los valores de los nodos de su subárbol izquierdo e *inferior* a los de su subárbol derecho.
- Para la inserción y búsqueda de elementos en un árbol binario sólo es necesario comparar el valor a insertar/buscar y considerar si es mayor insertarlo/buscarlo a la derecha y en otro caso a la izquierda, esto para cada subárbol encontrado.
- En la eliminación de un elemento se deben considerar 3 casos con base en el nodo a eliminar:
  - Si tiene 0 hijos: se elimina nada más.
  - Si tiene 1 hijo: se sube el hijo al padre del nodo a eliminar.
  - Si tiene 2 hijos: se sube el hijo más a la derecha del subárbol izquierdo.

REFERENCIAS

1. Cairó, O. and S. Guardati, *Estructuras de Datos*. 2a. Edición ed. 2002, México: Mc Graw Hill.

2. Joyanes, L. and I. Zahonero, *Estructura de Datos. Algoritmos, Abstracción y Objetos*. 1998, Madrid: McGraw-Hill.

3. Joyanes, L., et al., *Estructuras de datos en C*. 2005, Madrid: Mc Graw Hill, Serie Schaum.

4. Tenembaum, et al., *Estructuras de datos en C*. 1997, México: Prentice-Hall.

UNIDAD DE COMPETENCIA IV
Aplicar la estructura de datos grafo

25 HRS.
5 SEMANAS

4.1 GRAFOS: CARACTERÍSTICAS Y CLASIFICACIÓN

Un grafo se define como un conjunto de vértices o nodos *V* y un conjunto de arcos *A* [1, 2]. Se utilizan para representar redes de alcantarillado, redes de comunicaciones, circuitos eléctricos, etc.

Un grafo se representa con el par:

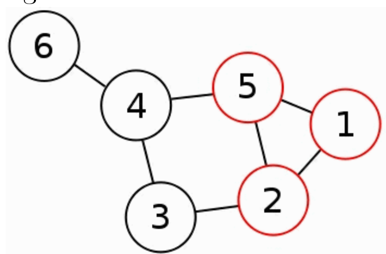
G = (V, A)

De esta manera los dos elementos básicos de un grafo son los *vértices o nodos* y los *arcos o aristas [3]*.

Donde:

- *Vértices*: Son los nodos que almacenan la información.
- *Arcos:* Representa la relación entre los nodos, es decir, la relación entre la información del nodo.

La siguiente figura representa un grafo.



Donde:

G = {V, A}
Vértices: V = {1, 2, 3, 4, 5, 6}
Arcos: A = {(6,4), (4,5), (5,1), (5,2), (4,3), (3,2), (4,6), (5,4), (1,5), (2,5), (4,3), (2,3)}

Un arco o arista está formado por un par de nodos y se escribe *(v1,v2)* siendo *v1 y v2* el par de nodos [2]. Se dice que hay un arco (*v1, v2*) que va del vértice *v1* al vértice *v2* cuando el nodo *v1* apunta a otro nodo *v2 [1].*

4.1.1 Características de los grafos

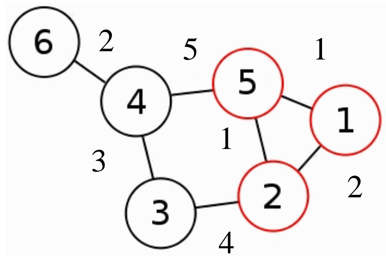
Las características y conceptos principales de los grafos se presentan en la Tabla 18. Considerando que *v* es un nodo:

Tabla 18: Características de un grafo

Características	Descripción
Vértices	Son los nodos que almacenan la información
Aristas	Representa la relación entre los nodos.
Factor de peso	Número que se asigna a las aristas, es decir que están ponderadas.
Grado	Número de aristas que contienen a *v* (nodo)
Nodo aislado	No tiene aristas, es decir, tiene grado 0, *Grado(v) = 0*
Lazo o bucle	Arista que conecta a un nodo consigo mismo. *a = (v, v)*
Camino	Un camino denominado *P* de longitud *n* desde un vértice *v* a un vértice *w* se define como la secuencia de *n* vértices que se debe seguir para llegar del nodo origen al nodo destino. *P = (v1, ..., vn)*
Camino cerrado	Si el primer y último nodo son iguales
Camino simple	Si todos sus nodos son distintos, con excepción de que el primero y el último si lo pueden ser.
Ciclo	Camino simple cerrado de longitud 3 o mayor.
Grafo conexo	Si existe un camino simple entre dos de sus nodos cualesquiera.
Grafo árbol	Si es un grafo conexo sin ciclos
Grafo completo	Si cada vértice *v* es adyacente a todos los demás de *G*. *N vértices tendrá n(n-1)/2 aristas*
Grafo etiquetado	Si sus aristas tienen asignado un valor llamado *peso o longitud.*
Multígrafo	Si al menos dos de sus vértices están conectados por dos aristas, éstas reciben el nombre de *aristas múltiples o paralelas.*

Ejercicio N° 1:

Con base en la Tabla 18 y el grafo siguiente identificar cada uno de sus términos.



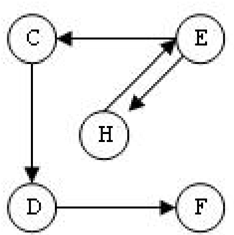
Término	Resultado
Vértices	**6, 4, 5, 1, 2, 3**
Aristas	**(6,4), (4,5), (5,1), (5,2), (2,3), (3,4), (4,6), (5,4), (1,5), (2,5), (3,2), (4,3)**
Factor de peso	(6,4) y (4,6): **2** (4,3) y (3,4): **3** (4,5) y (5,4): **5** (5,1) y (1,5): **1** (1,2) y (2,1): **2** (3,2) y (2,3): **4**
Grado	Grado **1**: Nodo **6** Grado **2**: Nodos **1, 3, 4** Grado **3**: Nodos **2, 5**
Nodo aislado	**No hay**
Lazo o bucle	**No hay**
Camino	Caminos de 6 a 2 pueden ser: **6-4-5-1-2** **6-4-3-2** **6-4-5-2**
Camino cerrado	Un camino cerrado es: **4-5-2-3-4** Otro es: **5-2-1-5** Un camino no cerrado es: **6-4-5-1**
Camino simple	Un camino simple es: **5-1-2-5** Un camino que no es simple: **6-4-5-2-1-5-3-4**
Ciclo	**4-5-2-4** **5-2-1-5**
Grafo conexo	**Es conexo porque todos los nodos tienen al menos un camino a otro nodo**
Grafo completo	**No es completo porque los nodos no llevan a todos los nodos**
Grafo valorado	**Sí es grafo valorado**

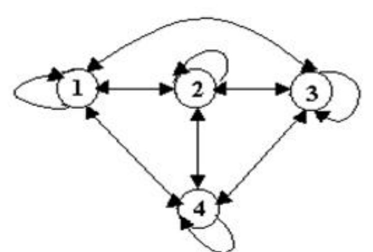
Actividad 49

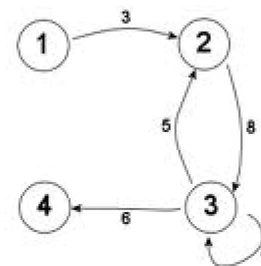
4.1.2 Clasificación de los grafos.

Los grafos se clasifican en *dirigidos y no dirigidos.*

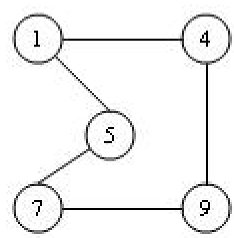
- **Grafos Dirigidos:** Un grafo dirigido se identifica cuando las relaciones de un vértice con otro se representan con una → ya que tienen asociada una dirección. Ejemplos de éstos se presentan en las figuras siguientes

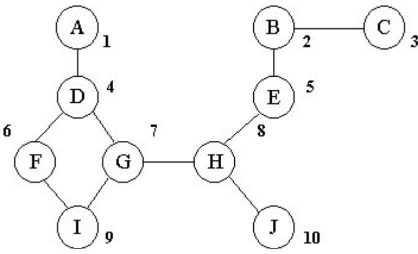


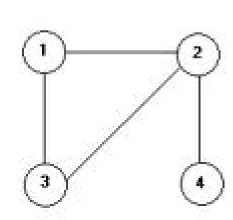




- **Grafos No Dirigidos:** Un grafo no dirigido es identificado cuando las relaciones de un vértice con otro se representan con una **—**. Ejemplos de este tipo de grafo se presentan en las figuras siguientes







Actividad 50

Resumen

- Un grafo se define como un conjunto de vértices o nodos *V* y un conjunto de arcos *A* [1, 2]. Se utilizan para representar redes de alcantarillado, redes de comunicaciones, circuitos eléctricos.
- Un *Vértice o Nodo*: es el nodo que almacena la información.
- El *Arco o Arista* de un grafo representa la relación entre los nodos, es decir, la relación entre la información de los nodos.
- El *Factor de peso* de grafo se refiere al número que se asigna a las aristas, es decir, aristas que están ponderadas.
- Existen dos tipos de grafos: *Dirigidos y No Dirigidos.*
- Un grafo *Dirigido* es aquel cuya relación entre vértices está representada con una flecha, es decir tiene una dirección.
- Un grafo *No Dirigido* es aquel cuya relación entre vértices está representada con una línea continua, es decir no tiene una dirección.

4.2 GRAFOS DIRIGIDOS

Llamados también *digrafos.* Un grafo es dirigido si los pares de nodos que forman los arcos son ordenados.

Este tipo de grafo se caracteriza porque arista *a* tiene una dirección asignada; es decir, cada arista está asociada a un par ordenado (*u, v)* de nodos de G (grafo). Para las aristas dirigidas se aplica la siguiente terminología [3]:

a) *a* empieza en *u* y termina en *v.*
b) *u* es el origen o punto inicial de *a,* y *v* es el destino o punto terminal de *a.*
c) *u* es un predecesor de *v* y *v* es un sucesor o vecino de *u.*
d) *u* es adyacente hacia *v* y *v* es adyacente desde *u.*

4.2.1 Representación

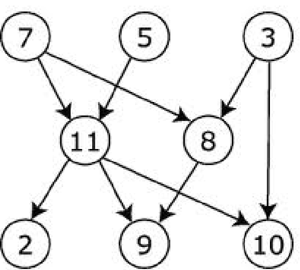
Un grafo se puede representar mediante una implementación estática (*matriz de adyacencia*) o una dinámica (*lista de adyacencia*).

4.2.1.1 Matriz de adyacencia

Un grafo puede ser representado utilizando una matriz denominada *matriz de adyacencia*, donde las filas y las columnas son los vértices del grafo y los elementos indican si entre un determinado par de vértices existe un arco que los una.

Ejercicio Nº 2:

Elaborar la matriz de adyacencia que representa el siguiente *grafo dirigido*.



- **Paso 1**, se procede a calcular el tamaño de la matriz de adyacencia considerando que ésta debe ser una matriz cuadrada de orden *VxV*, donde *V* es el número de vértices, en este ejercicio el tamaño de la matriz es de orden:

  8 x 8

- **Paso 2**, se identifican los vértices del grafo de manera ordenada.

  Vértices: V = {2, 3, 5, 7, 8, 9, 10, 11}

- **Paso 3**, representar cada columna y cada renglón con los vértices del grafo identificados en el paso anterior como se muestra a continuación y se llena la matriz con los valores de 0.

vértices	2	3	5	7	8	9	10	11
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0

- **Paso 4,** una vez identificado el tamaño de la matriz y los vértices, se procede a identificar los arcos considerando la dirección que entre éstos existe; para este ejercicio, entonces se tiene:

  Arcos: A = {(7,11), (7,8), (5,11), (3,8), (3,10), (8,9), (11,2), (11,9), (11, 10)}

- **Paso 5,** para cada arco existente entre vértices se coloca el valor **1** en la intersección de la columna y renglón que corresponde al arco en cuestión, considerando un par ordenado **(R, C)** donde **R** es corresponde al valor del renglón y **C** al valor de la columna que representan respectivamente.

  Por ejemplo, para este ejercicio y para el arco **(7, 11)** se tiene:

R: 7
C: 11

vértices	2	3	5	7	8	9	10	11
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	**1**
8	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0

Nota: *recordar que no es lo mismo el arco (7,11) que el (11,7) debido a que se hace referencia a diferente posición dentro de la matriz.*

El llenado de la matriz de la adyacencia de los pasos 4 y 5 anteriores, se puede representar con la siguiente función:

$$aij = \begin{cases} 1, & si\ hay\ un\ arco \\ 0, & si\ no\ hay\ arco \end{cases}$$

Así para cada uno de los arcos del grafo, finalmente la matriz de adyacencia es:

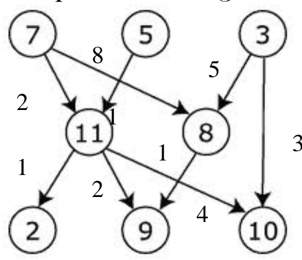
vértices	2	3	5	7	8	9	10	11
2	0	0	0	0	0	0	0	0
3	0	0	0	0	**1**	0	**1**	0
5	0	0	0	0	0	0	0	**1**
7	0	0	0	0	**1**	0	0	**1**
8	0	0	0	0	0	**1**	0	0
9	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0
11	**1**	0	0	0	0	**1**	**1**	0

Grafo con factor de peso

Una variación que puede haber en un grafo es que éste tenga un *factor de peso.* Los grafos con factor de peso pueden representarse de tal forma que si existe arco, el elemento A[i, j] es el factor de peso, la no existencia de peso supone que A[i, j] es 0 (cero) [1], donde A es la matriz de adyacencia.

Ejercicio Nº 3:

Elaborar la matriz de adyacencia que representa el siguiente *grafo con factor de peso.*



- **Paso 1**, el tamaño de la matriz es de orden: *8x8*
- **Paso 2**, se identifican los vértices del grafo de manera ordenada.

Vértices: V = {2, 3, 5, 7, 8, 9, 10, 11}

- **Paso 3**,

Vértices	2	3	5	7	8	9	10	11
2								
3								
5								
7								
8								
9								
10								
11								

- **Paso 4,**
  Arcos: A = {(7,11), (7,8), (5,11), (3,8), (3,10), (8,9), (11,2), (11,9), (11, 10)}

- **Paso 5,**

$$aij = \begin{cases} factor\ de\ peso, & si\ hay\ un\ arco \\ 0, & si\ no\ hay\ arco \end{cases}$$

Se obtienen los factores de peso de cada uno de los arcos y sus vértices, los cuales se muestran a continuación.

Arco	Factor de Peso
(7, 11)	2
(7, 8)	8
(5, 11)	1
(3, 8)	5
(3, 10)	3
(8, 9)	1
(11, 2)	1
(11, 9)	2
(11, 10)	4

Los factores de peso obtenidos se representan en la matriz de adyacencia para cada uno de los arcos del grafo, quedando finalmente:

Vértices	2	3	5	7	8	9	10	11
2	0	0	0	0	0	0	0	0
3	0	0	0	0	**5**	0	**3**	0
5	0	0	0	0	0	0	0	**1**
7	0	0	0	0	**8**	0	0	**2**
8	0	0	0	0	0	**1**	0	0
9	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0
11	**1**	0	0	0	0	**2**	**4**	0

Actividad 52

4.2.1.2. Lista de adyacencia

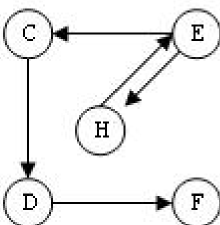
Es una estructura multienlazada formada por una lista directorio, cuyos nodos representan los vértices del grafo y además, de cada uno de estos nodos se puede emerger otra lista enlazada cuyos nodos a su vez representan los arcos cuyo vértice origen es el del nodo de la lista directorio [1].

La lista de adyacencia para un vértice *a* es una lista ordenada de todos los vértices adyacentes de *a,* para su representación se puede utilizar un arreglo *HEAD* donde *HEAD[i]* es un

apuntador a la lista de vértices adyacentes al vértice *i [2]* o bien una lista mutienlazada donde *HEAD* sería la lista principal.
Lo anterior, se ejemplifica con el ejercicio N° 4.

Ejercicio N° 4

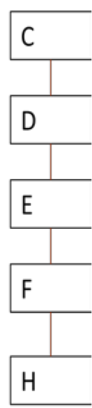
Obtener la lista de adyacencia del siguiente grafo.



- **Paso 1**, se procede a calcular el tamaño del arreglo *HEAD considerando* el número de vértices, en este ejercicio el tamaño es: *5*

- **Paso 2**, se identifican los vértices del grafo de manera ordenada.
  Vértices: V = {C, D, E, F, H}

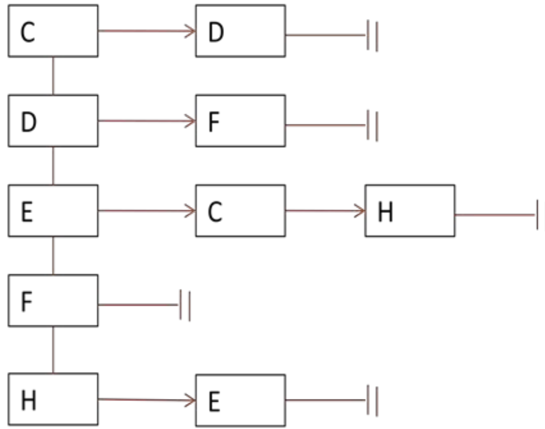
- **Paso 3**, se esquematiza con el arreglo HEAD.



- **Paso 4,** con base en los vértices se identifican los arcos
  Arcos: A = {(C, D), (E, H), (E, C), (D, F), (H, E)}

- **Paso 5,** se crean los nodos de la lista de cada uno de los elementos del arreglo HEAD, con base en los vértices obtenidos en el paso 2 y los arcos que los relacionan:
  Vértice C: (C, D)
  Vértice D: (D, F)
  Vértice E: (E, C) y (E, H)
  Vértice F: Nulo
  Vértice H: (H, E)

- **Paso 6,** se esquematiza el paso 5, creando la lista de adyacencia final.



Actividad 53

Resumen

- Un grafo dirigido es llamado también *digrafo*.
- Un grafo se puede representar de manera estática por medio de una matriz de adyacencia y de manera dinámica por medio de una lista de adyacencia.
- En una matriz de adyacencia las columnas y los renglones representan los vértices del grafo.
- Los arcos en una matriz de adyacencia se representan colocando un 1 si existe arco entre vértices y un 0 si no existe.
- El factor de peso de un arco es la ponderación que tiene ese arco entre los vértices que se relacionan.
- Cuando los arcos en un grafo tienen factor de peso (ponderación) éste es colocado en la matriz de adyacencia en la intersección de los vértices que tienen el arco.
- En una lista de adyacencia los vértices se representan como los nodos principales de una lista simplemente enlazada y los vértices con los cuales tiene relación se enlazan en cada uno de los vértices nodos.
- Para considerar el factor de peso de un grafo en una lista de adyacencia éste se incluye como campo en nodo del vértice al cual se liga.

4.3 GRAFOS NO DIRIGIDOS.

Un grafo no dirigido es aquel en el que los arcos están formados por pares de nodos no ordenados, no apuntados.

4.3.1 Representación

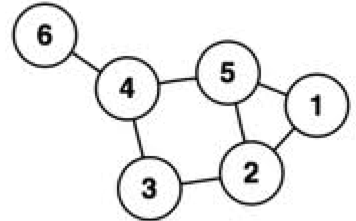
Este tipo de grafo también se puede representar mediante una implementación estática (*matriz de adyacencia*) o una dinámica (*listas multienlazadas o de adyacencia*).

4.3.1.1 Matriz de adyacencia

Bajo el mismo concepto de matriz de adyacencia, para los grafos no dirigidos se sigue la misma secuencia de pasos sólo considerando que la matriz de adyacencia de este tipo de grafos es una matriz simétrica [1], esto debido a que como no existe dirección entre los vértices se considera bidireccional.

Ejercicio Nº 5:

Elaborar la matriz de adyacencia que representa el siguiente *grafo no dirigido*.



- **Paso 1**, el tamaño de la matriz es de orden:

  6 x 6

- **Paso 2**, se identifican los vértices del grafo de manera ordenada.

  Vértices: V = {1, 2, 3, 4, 5, 6}

- **Paso 3**,

Vértices	1	2	3	4	5	6
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0

- **Paso 4,**

  Arcos: A = {(6,4), (4,5), (5,1), (5,2), (4,3), (3,2), (4,6), (5,4), (1,5), (2,5), (4,3), (2,3)}

- **Paso 5,**

$$aij = \begin{cases} 1, & si\ hay\ un\ arco \\ 0, & si\ no\ hay\ arco \end{cases}$$

Así para cada uno de los arcos del grafo, finalmente la matriz de adyacencia es:

Vértices	1	2	3	4	5	6

1	0	0	0	0	**1**	0
2	0	0	**1**	0	**1**	0
3	0	**1**	0	**1**	0	0
4	0	0	**1**	0	**1**	**1**
5	**1**	**1**	0	**1**	0	0
6	0	0	0	**1**	0	0

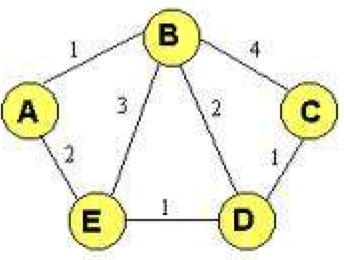
Actividad 54

Grafo con factor de peso

Una variación que puede haber en un grafo es que éste tenga un *factor de peso.* Los grafos con factor de peso pueden representarse de tal forma que si existe arco, el elemento A[i, j] es el factor de peso, la no existencia de peso supone que A[i, j] es 0 (cero) [1], donde A es la matriz de adyacencia y además es simétrica.

Ejercicio Nº 6:

Elaborar la matriz de adyacencia que representa el siguiente *grafo no dirigido con factor de peso.*



- **Paso 1**, el tamaño de la matriz es de orden: *5x5*

- **Paso 2**, se identifican los vértices del grafo de manera ordenada.

  Vértices: V = {A, B, C, D, E}

- **Paso 3**,

Vértices	**A**	**B**	**C**	**D**	**E**
A					
B					
C					
D					
E					

- **Paso 4,**

  Arcos: A = {(A,E), (E,A), (A,B), (B,A), (B,E), (E,B), (B,D), (D,B), (D, E), (E,D), (D,C), (C,D), (C,B), (B,C)}

- **Paso** *5,*

$$aij = \begin{cases} factor\ de\ peso, & si\ hay\ un\ arco \\ 0, & si\ no\ hay\ arco \end{cases}$$

Se obtienen los factores de peso de cada uno de los arcos y sus vértices, los cuales se muestran a continuación.

Arco	Factor de Peso
(A,E)	2
(A,B)	1
(B,E)	3
(B,D)	2
(D,E)	1
(D,C)	1
(C,B)	4

Y los cuales se representan en la matriz de adyacencia para cada uno de los arcos del grafo, finalmente la matriz de adyacencia es:

Vértices	A	B	C	D	E
A	0	**1**	0	0	**2**
B	**1**	0	**4**	**2**	**3**
C	0	**4**	0	**1**	0
D	0	**2**	**1**	0	**1**
E	**2**	**3**	0	**1**	0

Actividad *55*

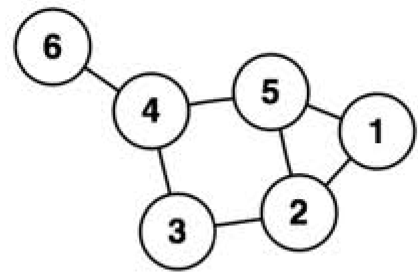
4.3.1.2. Lista de adyacencia

Recordando que la lista de adyacencia para un vértice *a* es una lista ordenada de todos los vértices adyacentes de *a,* para su representación se puede utilizar un arreglo *HEAD* donde *HEAD[i]* es un apuntador a la lista de vértices adyacentes al vértice *i [2]* o bien una lista multienlazada donde *HEAD* sería la lista principal.

Para entender lo anterior, se ejemplifica con el ejercicio N° 25.

Ejercicio N° 7:

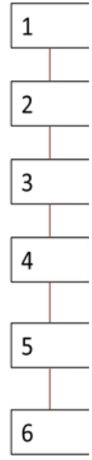
Obtener la lista de adyacencia del siguiente grafo.



- **Paso 1**, se procede a calcular el tamaño del arreglo *HEAD considerando* el número de vértices, en este ejercicio el tamaño es: *6*

- **Paso 2**, se identifican los vértices del grafo de manera ordenada.
Vértices: V = {1, 2, 3, 4, 5, 6}

- **Paso 3**, se esquematiza con el arreglo HEAD.

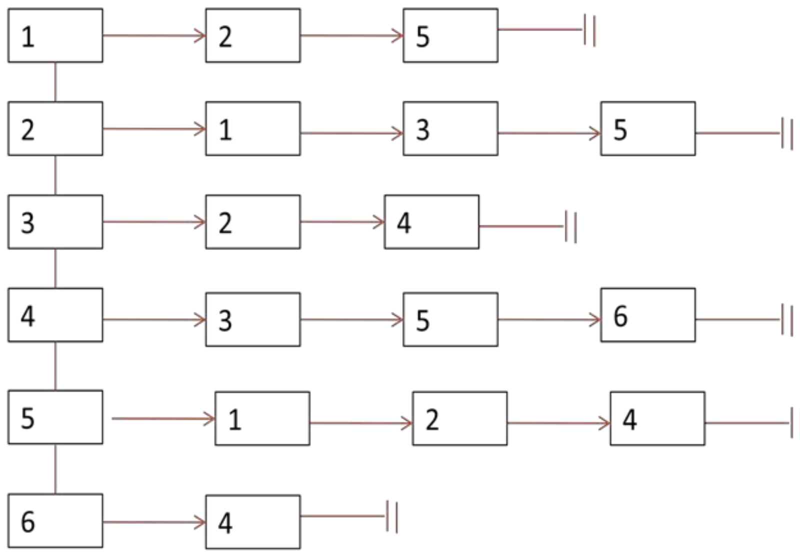


- **Paso 4,** con base en los vértices se identifican los arcos
Arcos: {(6,4), (4,5), (5,1), (5,2), (4,3), (3,2), (4,6), (5,4), (1,5), (2,5), (4,3), (2,3)}

- **Paso 5,** se crean los nodos de la lista de cada uno de los elementos del arreglo HEAD, con base en los vértices obtenidos en el paso 2 y los arcos que los relacionan:

Vértice 6: (6, 4)
Vértice 4: (4, 5), (4, 6) y (4, 3)
Vértice 5: (5, 4), (5,1) y (5, 2)
Vértice 1: (1, 5) y (1, 2)
Vértice 2: (2, 3), (2, 5) y (2, 1)
Vértice 3: (3, 4) y (3, 2)

- **Paso 6,** se esquematiza el paso 5, creando la lista de adyacencia final. A diferencia de la lista de adyacencia de un grafo dirigido es que en la de este tipo los vértices se replican en cada nodo cuando hay una relación, esto es, el vértice *u* estará en la lista de adyacencia del vértice *v* y viceversa.



Actividad 56

Resumen

- Un grafo no dirigido también se puede representar de manera estática por medio de una matriz de adyacencia y de manera dinámica por medio de una lista de adyacencia.
- En una matriz de adyacencia las columnas y los renglones representan los vértices del grafo.
- Los arcos en una matriz de adyacencia se representan colocando un 1 si existe arco entre vértices y un 0 si no existe, considerando que el grafo no dirigido tienen doble dirección.
- Cuando los arcos en un grafo tienen factor de peso (ponderación) éste es colocado en la matriz de adyacencia en la intersección de los vértices que tienen el arco, considerando que el grafo no dirigido tienen doble dirección.
- En una lista de adyacencia los vértices se representan como los nodos principales de una lista simplemente enlazada y los vértices con los cuales tiene relación se enlazan en cada uno de los vértices nodos, considerando que el grafo no dirigido tienen doble dirección.
- Para considerar el factor de peso de un grafo en una lista de adyacencia éste se incluye como campo en nodo del vértice al cual se liga.

4.4 TAD GRAFO (ESTÁTICO Y DINÁMICO)

Los grafos al igual que las pilas, colas, listas, etc. son tipos abstractos de datos y sus operaciones básicas son:

- *InicializaGrafo(G):* El grafo no tiene vértices ni arcos.
- *Union(X, Y):* Añade el arco (X, Y) al grafo G.
- *BorrarArco(X, Y):* Elimina del grafo G el arco (X, Y)
- *Adyacente(X, Y):* Función que devuelve un cero si X y Y no forman un arco
- *NuevoVertice(X):* Añade el vértice X al grafo G.
- *BorrarVertice(X):* Elimina del grafo G el vértice X.

La implementación de estas operaciones dependen de su representación (estática o dinámica) [2], es decir, por medio de la matriz de adyacencia o de la lista de adyacencia. A continuación se describen cada una de ellas realizando un ejercicio de operaciones de grafos.

4.4.1 Implementación con matriz de adyacencia

Para la implementación de las operaciones de un grafo se requiere primero definir cuál es la estructura de éste. La siguiente es la definición con la que se pretende trabajar.

```
Arco: Registro
        u: TipodeDato
        v: TipodeDato
FinRegistro

Grafo: Registro
        V[N]: TipodeDato
        A[N][N]: TipodeDato
FinRegistro

*G: Grafo, Arc: Arco
```

Donde:

u y *v:* son los vértices del arco
V[ ]: es el vector de vértices del grafo
A[ ][ ]: es la matriz de adyacencia
N: valor máximo que pueden tomar los vértices (enteros)

Las operaciones implementadas con una representación estática son las que se muestran en la Tabla 19, basadas en la definición del TAD grafo anterior.

Tabla 19: Operaciones TAD Grafo con representación estática.

Operación	Descripción	Implementación

InicializaGrafo(G)	El grafo no tiene vértices ni arcos	***Modulo IniciaGrafo(*G: Grafo)***
		Inicia
		    I, J: E
		    Para I←1, I<N, I←I+1
		        G->V[I] ← 0
		        Para J←1, J<=N, J←J+1
		            G->A[I][J] ← 0
		        FinPara
		    FinPara
		Termina

Operación	Descripción	Implementación
Union(u,v)	Añade el arco (u, v) al grafo G. Añadir los dos vértices del arco	***Modulo AnadeArco(*G: Grafo, Arc: Arco)*** ***Inicia*** *AnadeVertice(G, Arc.u)* *AnadeVertice(G, Arc.v)* *G->A[Arc.u][Arc.v] ← 1* ***Termina***
BorrarArco(u,v)	Elimina del grafo G el arco	***Modulo ElimArco(*G: Grafo, Arc: Arco)*** ***Inicia*** *G->A[Arc.u][Arc.v] ← 0* ***Termina***
Adyacente(u,v)	Función que devuelve cero (0) si no forman un arco los vértices u, v	***Modulo PerteneceArco(*G: Grafo, Arc: Arco)*** ***Inicia*** *Sw: E* *Sw ← PerteneceVertice(G, Arc.u) y* *PertenceVertice(G, Arc.v)* *Si (Sw = 1) entonces* *Regresa G->A[Arc.u][Arc.v]* *Otro* *Regresa 0* *FinSi* ***Termina***
	Decide si un vértice está en el grafo	***Modulo PerteneceVertice(*G: Grafo, Arc: Arco)*** ***Inicia*** *Si (v >= 1 y v<=N) entonces* *Regresa G->V[v]* *Otro* *Regresa 0* *FinSi* ***Termina***
NuevoVertice(v)	Añade el vértice *v* al grafo	***Modulo AnadeVertice(*G: Grafo, v:E)*** ***Inicia*** *Si (v >= 1 y v<=N) entonces* *G->V[v] ← 1* *FinSi* ***Termina***
BorrarVertice(v)	Elimina del grafo G el vértice v	***Modulo ElimVertice(*G: Grafo, v:E)*** ***Inicia*** *I: E* *Si (v >= 0 y v<N) entonces* *G->V[v] ← 0* *Para I←1, I<=N, I←I+1* *G->A[I][v] ← 0* *G->A[v][I] ← 0* *FinPara* *FinSi* ***Termina***

Para dejar en claro la funcionalidad de cada una de éstas se presenta el ejercicio siguiente.

Ejercicio Nº 8:

V= {1, 4, 2, 3}
A= {(1, 4), (1,2), (4,3), (3,2), (3, 1)}
N= 4 (el número mayor de los vértices es 4)

<table>
<tr><th>Operación</th><th>Representación G</th><th>Procesos</th><th>Grafo</th></tr>
<tr><td>IniciaGrafo(G)</td><td>V:<table><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>A:<table><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table></td><td></td><td></td></tr>
<tr><td>AnadeVertice(1)</td><td>V:<table><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td></tr></table></td><td>v← 1
G->V[1]← 1</td><td>1</td></tr>
<tr><td>AnadeVertice(4)</td><td>V:<table><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr></table></td><td>v:4
G->V[4]← 1</td><td>4 1</td></tr>
<tr><td>AnadeArco(G,1,4)</td><td>V:<table><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>A:<table><tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>3</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table></td><td>Arc.u: 1
Arc.v: 4
AnadeVertice(G, 1)
AnadeVertice(G, 4)
G->A[1][4]← 1</td><td>1
4</td></tr>
<tr><td>Adyacente(G,4,1)</td><td>A:<table><tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>3</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table></td><td>Arc.u: 4
Arc.v: 1
Sw← PerteneceVertice(G, 4) y
PertenceVertice(G, 1)
Sw: 0</td><td>Regresa 0 entonces
No son adyacentes</td></tr>
</table>

<table>
<tr><th>Operación</th><th>Representación G</th><th>Procesos</th><th>Grafo</th></tr>
<tr>
<td>AnadeArco(G,1,2)</td>
<td>V:
<table>
<tr><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td></tr>
</table>
A:
<table>
<tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><th>1</th><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><th>2</th><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><th>3</th><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><th>4</th><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>
</td>
<td>Arc.u: 1
Arc.v: 2
AnadeVertice(G, 1)
AnadeVertice(G, 2)
G->A[1][2] ← 1</td>
<td>1
4 2</td>
</tr>
<tr>
<td>AnadeArco(G,4,3)</td>
<td>V:
<table>
<tr><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>
A:
<table>
<tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><th>1</th><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><th>2</th><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><th>3</th><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><th>4</th><td>0</td><td>0</td><td>1</td><td>0</td></tr>
</table>
</td>
<td>Arc.u: 4
Arc.v: 3
AnadeVertice(G, 4)
AnadeVertice(G, 3)
G->A[4][3] ← 1</td>
<td>1
2
4
3</td>
</tr>
<tr>
<td>AnadeArco(G,3,2)</td>
<td>V:
<table>
<tr><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>
A:
<table>
<tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><th>1</th><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><th>2</th><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><th>3</th><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><th>4</th><td>0</td><td>0</td><td>1</td><td>0</td></tr>
</table>
</td>
<td>Arc.u: 3
Arc.v: 2
AnadeVertice(G, 3)
AnadeVertice(G, 2)
G->A[3][2] ← 1</td>
<td>1
2
4
3</td>
</tr>
</table>

<table>
<tr><th>Operación</th><th>Representación G</th><th>Procesos</th><th>Grafo</th></tr>
<tr>
<td>AnadeArco(G,3,1)</td>
<td>V:
<table><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>A:
<table><tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>3</td><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>4</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table></td>
<td>Arc.u: 3
Arc.v: 1
AnadeVertice(G, 3)
AnadeVertice(G, 1)
G->A[3][1] ← 1</td>
<td></td>
</tr>
<tr>
<td>ElimVertice(G,2)</td>
<td>V:
<table><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>A:
<table><tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>3</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>4</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table></td>
<td>v: 2
N: 4
Si (v >= 0 y v<N) entonces
G->V[2] ← 0
Para I←1, I<=N, I←I+1
G->A[I][2] ← 0
G->A[2][I] ← 0
FinPara
FinSi</td>
<td></td>
</tr>
<tr>
<td>AnadeArco(G,3,4)</td>
<td>V:
<table><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>A:
<table><tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>3</td><td>1</td><td>0</td><td>0</td><td>1</td></tr><tr><td>4</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table></td>
<td>Arc.u: 3
Arc.v: 4
AnadeVertice(G, 3)
AnadeVertice(G, 4)
G->A[3][4] ← 1</td>
<td></td>
</tr>
</table>

<table>
<tr><th>Operación</th><th>Representación G</th><th>Procesos</th><th>Grafo</th></tr>
<tr>
<td>BorrarArco(G,4,3)</td>
<td>V:
<table>
<tr><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td></tr>
</table>
A:
<table>
<tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>3</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>
</td>
<td>Arc.u: 4
Arc.v: 3
G->A[4][3] ← 0</td>
<td></td>
</tr>
</table>

Actividades 57 y 58

4.4.2 Implementación con lista de adyacencia

Para esta implementación se hace uso de una estructura multienlazada de la siguiente forma [1]:

- Una lista enlazada general mantiene información de todos los vértices del grafo.
- Cada nodo de la lista contiene un puntero *sig* al siguiente nodo de la lista y un campo *el* que es un registro que contiene un campo *v* donde almacena la información del vértice y un puntero *ady* (lista de adyacencia) que apunta a una lista enlazada donde se almacenan lo distintos arcos adyacentes al vértices.
- Cada lista de adyacencia se implementa como una lista enlazada cuyos nodos contienen los campos *sig* que apunta al siguiente nodo, y el campo *el* que a su vez es un registro que contienen el campo *v* para almacenar el otro vértice del arco y el campo *valor* que contiene el valor del arco.

Además de las operaciones *IniciarGrafo, Adyacencia, ElimVertice, BorrarArco, AnadeVertice, AnadeArco*, se tienen que considerar las siguientes operaciones:

- *AnadeConjuntoAdy:* Añade un nodo de la lista de adyacencia de un vértice.
- *PertenceConjuntoAdy:* Decide si un vértice se encuentra en la lista de adyacencia de otro vértice.
- *BorraConjuntoAdy:* Elimina un nodo de la lista de adyacencia.
- *BorraListaAdy:* Elimina todos los nodos de una lista de adyacencia.
- *EncuentraPosAdy:* Encuentra dos punteros, uno al nodo que contiene un vértice y otro al nodo inmediatamente anterior.

- *Inicializa:* Crea el grafo vacío poniendo la lista general a *Nulo*.
- *AnadeArco:* Añade un arco, para ello se asegura de que los vértices que lo componen estén en la lista general añadiéndolos previamente.
- *BorrarArco:* Borra un arco eliminando el vértice destino de la lista de adyacencia que emerge del vértice fuente
- *AnadeVertice:* Añade un vértice a la lista general de vértices.
- *BorrarVertice:* Borra un vértice de la lista general así como todos los arcos que llegan o emergen de él.
- *PerteneceVertice:* Decide si un vértice está en el grafo comprobando su pertenencia a lista general.
- *PerteneceArco:* Decide si un arco está en el arco.

Para la implementación de las operaciones de un grafo se declaran las estructuras necesarias con las que se realiza la representación.

```
Arco: Registro       ItemAdy: Registro    ListaAdy: Registro    ItemG: Registro       ListaG: Registro
   u: E                 v: E                 el: ItemAdy           v: E                  el: ItemG
   v: E                 valor: R             *sig: ListaAdy        *Ady: ListaAdy        *sig: ListaG
   valor: R          FinRegistro          FinRegistro           FinRegistro           FinRegistro
FinRegistro
```

Las operaciones implementadas con una representación dinámica son las que se muestran en la Tabla 20, basadas en la definición del TAD grafo anterior.

Tabla 20: Operaciones TAD Grafo con representación dinámica.

Operación	Implementación
AñadeConjuntoAdy	***Modulo AñadeConjuntoAdy (*Primero: ListaAdy, dato:ItemAdy)*** ***Inicia*** **Nuevo: ListaAdy* *Si (PerteneceConjuntoAdy(Primero.dato) ≠ 1) entonces* *Nuevo ← Asignar memoria* *Nuevo->el ← dato* *Nuevo->sig ← Primero* *Primero ← Nuevo* *FinSi* ***Termina***
PerteneceConjuntoAdy	***Modulo PerteneceConjuntoAdy(*Primero: ListaAdy, dato: ItemAdy)*** ***Inicia*** **ptr: ListaAdy* *Para ptr←Primero, ptr ≠ nulo, ptr ← ptr->sig* *Si (ptr->el.v = dato.v) entonces* *Regresa 1* *FinSi* *FinPara* *Regresa 0* ***Termina***
BorraConjuntoAdy	***Modulo BorraConjuntoAdy(*Primero: ListaAdy, dato: ItemAdy)*** ***Inicia*** **ptr, *ant: ListaAdy* *EncuentraPosAdy(Primero, dato, ant, ptr)* *Si (ptr ≠ nulo) entonces* *Si (ptr = primero) entonces* *Primero ← ptr->sig* *Otro* *ant->sig ←ptr->sig* *FinSi* *FinSi* *Liberar (ptr)* ***Termina***
BorraListaAdy	***Modulo BorraListaAdy(*Primero: ListaAdy)*** ***Inicia*** **l, *ll: ListaAdy* *l ← Primero* *Mientras (l ≠ nulo)* *ll ← l* *l ← l->sig* *Liberar (ll)* *FinMientras* ***Termina***
EncuentraPosAdy	***Modulo EncuentraPosAdy(*Primero: ListaAdy, dato: ItemAdy, *Ant: ListaAdy, *Pos: ListaAdy)*** ***Inicia*** **ptr, *antt: ListaAdy* *enc: E* *enc ← 0* *ptr ← Primero* *antt ← nulo* *Mientras (enc ≠ 1 y ptr ≠ nulo)* *Si (ptr->el-v = dato.v) entonces* *enc ← 1* *Otro* *enc ← 0* *FinSi* *Si (enc ≠ 1) entonces* *antt ←ptr*

Operación	Implementación
	ptr ← ptr->sig FinSi FinMientras Ant ← antt Pos ← ptr **Termina**
Inicializa	**Modulo Inicializa (*Primero: ListaG)** **Inicia** Primero ← Nulo **Termina**
AnadeArco	**Modulo AñadeArco(*Primero: ListaG, arc: Arco)** **Inicia** Dato: ItemG Dato1:ItemAdy *Ant, *Pos: ListaG Dato.v ← arc.u Dato.Ady ← nulo AñadeVertice(Primero, Dato) Dato.v ← arc.v AñadeVertice(Primero, Dato) Dato.v ← arc.u EncuentraPosGrafo(Primero, Dato,Ant,Pos) Dato1.v ← arc.v Dato1.valor ← arc.valor AñadeConjuntoAdy(Pos->el.Ady, Dato1) **Termina**
BorraArco	**Modulo BorraArco(*Primero: ListaG, arc: Arco)** **Inicia** Dato: ItemG Dato1:ItemAdy *Ant, *Pos: ListaG Dato.v ← arc.u Dato.Ady ← nulo EncuentraPosGrafo(Primero, Dato,Ant,Pos) Dato1.v ← arc.v Dato1.valor ← arc.valor BorraConjuntoAdy(Pos->el.Ady, Dato1) **Termina**
AñadeVértice	**Modulo AñadeVertice(*Primero: ListaG, dato: ItemAdy)** **Inicia** *Nuevo: ListaG Si (PertenceVertice(Primero,dato) ≠ 1) entonces Nuevo ← Asignar memoria Nuevo->el ← dato Nuevo->sig ← Primero Primero ← Nuevo FinSi **Termina**
BorraVértice	**Modulo BorraVertice(*Primero: ListaG, dato:ItemG)** **Inicia** *ptr, *ant: ListaG Dato1: ItemAdy EncuentraPosGrafo(Primero, dato, ant, ptr) Si (ptr ≠ nulo) entonces Si (ptr = primero) entonces Primero ← ptr->sig Otro ant->sig ←ptr->sig FinSi

Operación	Implementación
	BorraListaAdy(ptr->el.Ady) *Liberar (ptr)* *ptr ← primero* *dato1.v ← dato.v* *Mientras (ptr ≠ nulo)* *BorraConjuntoAdy(ptr->el.Ady),dato1)* *ptr ← ptr->sig* *FinMientras* *FinSi* ***Termina***
PerteneceVertice	***Modulo PerteneceVertice(*Primero: ListaG, dato: ItemAdy)*** ***Inicia*** **ptr: ListaG* *Para ptr←Primero, ptr ≠ nulo, ptr ← ptr->sig* *Si (ptr->el.v = dato.v) entonces* *Regresa 1* *FinSi* *FinPara* *Regresa 0* ***Termina***
PerteneceArco	***Modulo PerteneceArco (*Primero: ListaG, dato: ItemAdy)*** ***Inicia*** *Dato: ItemG* *Dato1:ItemAdy* **Ant, *Pos: ListaG* **Ant1, *Pos1: ListaAdy* *Dato.v ← arc.u* *Dato.Ady ← nulo* *EncuentraPosGrafo(Primero, Dato,Ant,Pos)* *Dato1.v ← arc.v* *Dato1.valor ← arc.valor* *Si (Pos ≠ nulo) entonces* *EncuentraPosAdy(Pos->el.Ady, Dato1,Ant1,Pos1)* *Si (Pos1 ≠ Nulo) entonces* *Regresa 1* *Otro* *Regresa 0* *FinSi* *FinSi* *Regresa 0* ***Termina***

Actividad 59

Resumen

- La implementación de una matriz de adyacencia en cualquier lenguaje de programación se realiza utilizando arreglos bidimensionales (matrices)
- La implementación de una lista de adyacencia en cualquier lenguaje de programación se realiza utilizando listas simplemente ligadas considerando que cada nodo vuelve a ser una lista simplemente enlazada, es decir, listas de listas.

4.5 RECORRIDOS DE UN GRAFO

La operación de recorrer un grafo consiste en visitar (procesar) cada uno de los nodos a partir de uno dado [2]. Al igual que los árboles, hay dos formas en recorrer un grafo: *recorrido en anchura y recorrido en profundidad* [1].

4.5.1 Recorrido en anchura

Este recorrido visita el vértice partida para después visitar todos los nodos adyacentes que no estuvieran ya visitados y así sucesivamente, utiliza dos estructuras auxiliares [1]:

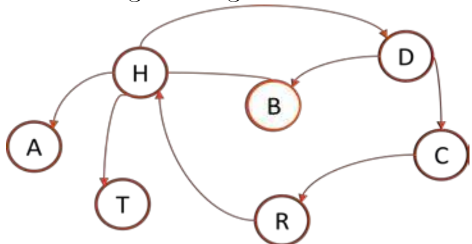
- una *cola o una pila* para colocar los vértices adyacentes hasta que les corresponda ser procesados
- y una *lista* con todos los vértices del grafo ya visitados.

Lo anterior se define en los siguientes pasos [2]:

1. Visitar el vértice de partida A
2. Meter en la cola el vértice de partida y marcarle como procesado
3. Repetir los pasos 4 y 5 hasta que la cola esté vacía
4. Retirar el nodo frente (*W*) de la cola, visitar *W*
5. Meter en la cola todos los vértices adyacentes a W que no estén procesados y marcarlos como procesados.
6. Recorrido final son los elementos que fueron saliendo de la cola en ese mismo orden.

Ejercicio N° 9

Realizar el recorrido en anchura del siguiente grafo considerando los pasos anteriores.



Paso	Instrucción	Resultado
1	Vértice de partida	*D*
2	Meter a la cola y marcar como procesado	H D B A C T R *Frente* D
3	Repetir los pasos 4 y5 hasta que la cola esté vacía	
4	Retirar el vértice frente (*D*) de la cola, visitar *D*	*Frente* *Elemento Retirado* D

Paso	Instrucción	Resultado
5	Meter en la cola todos los vértices adyacentes a **D** que no estén procesados y marcarlos como procesados	Fren: B C
4 y 5	Para el vértice frente (*B*)	Fren: C H; Elemento Retirado: B
4 y 5	Para el vértice frente (*C*)	Fren: H R; Elemento Retirado: C
4 y 5	Para el vértice frente (*H*)	Fren: R A T; Elemento Retirado: H
4 y 5	Para el vértice frente (*R*) Todos los vértices marcados	Fren: A T; Elemento Retirado: R
4 y 5	Para el vértice frente (*A*) Todos los vértices marcados	Fren: T; Elemento Retirado: A
4 y 5	Para el vértice frente (*T*) Todos los vértices marcados y cola vacía.	Fren: (vacía); Elemento Retirado: T
6	**Recorrido final**	***D B C H R A T***

Actividad 60

4.5.1.1 Implementación del recorrido en Anchura

```
Algoritmo
Inicia
        Mientras la cola no esté vacía
                Sacar vértice u de la cola
                Para cada vértice v adyacentes a u
                        Si el vértice v no está en el conjunto de visitados entonces
                                Añadirlo al conjunto de visitados y a la cola
                        FinSi
                FinPara
        FinMientras
Termina


Modulo RecorridoAnchura(*G: ListaG, v:E, *C: Cola)
Inicio
        Cl: Cola
        Dato: ItemG
        *Ady: ListaAdy
        *Visitados, *Ant, *Pos: ListaG
        InicializaCola(C)
        InicializaCola(Cl)
        AñadeC(Cl,v)
        AñadeC(C,v)
        Inicializa(&Visitados)
        Dato.v ← v
        AñadeVertice(&Visitados, dato)
        Mientras(ColaVacía(Cl) ≠ 1) entonces
                Dato.v ← Frente(Cl)
                ElimC(&Cl)
                EncuentraPosGrafo(G, dato, &Ant, &Pos)
                Si (Pos ≠ Nulo) entonces
                        Ady ← Pos->el.Ady
                        Mientras (Ady ≠ Nulo)
                                Dato.v ← Ady->el.v
                                Si (PerteneceVertice(Visitados, dato) ≠ 1) entonces
                                        AñadeC(&Cl, dato.v)
                                        AñadeC(C,dato.v)
                                        AñadeVertice(&Visitados, dato)
                                FinSi
                                Ady ← Ady->sig
                        FinMientras
                FnSi
        FinMientras
Termina
```

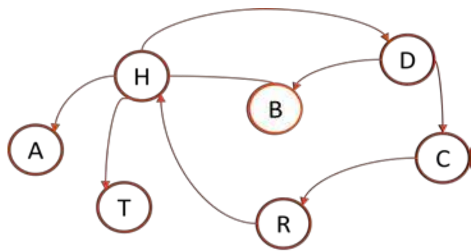
4.5.2 Recorrido en profundidad

De la misma manera que el recorrido en anchura, el recorrido en profundidad empieza por un vértice V del grafo G; V se marca como visitado. Se denomina en profundidad porque la dirección de "visitar" es hacia adelante mientras que sea posible, al contrario que el de anchura, que primero visita todos los vértices posibles en amplitud.

Los pasos a ejecutar son los mismos sólo siguiendo la dirección de profundidad y considerando además en lugar de una cola una pila.

Ejercicio N° 9

Realizar el recorrido en profundidad del mismo grafo que del recorrido en anchura.



Paso	Instrucción	Resultado
1	Vértice de partida	*D*
2	Meter a la pila y marcar como procesado	*Tope* D
3	Repetir los pasos 4 y5 hasta que la pila esté vacía	
4	Retirar el vértice frente (*D*) de la pila, visitar *D*	*Tope* D *Elemento Retirado* D
5	Meter en la pila todos los vértices adyacentes a **D** que no estén procesados y marcarlos como procesados	*Tope* B C
4 y 5	Para el vértice tope (*C*)	*Tope* B R *Elemento Retirado* C
4 y 5	Para el vértice tope (*R*)	*Tope* B H *Elemento Retirado* R
4 y 5	Para el vértice tope (*H*) Todos los vértices marcados	*Tope* B A T *Elemento Retirado* H

Paso	Instrucción	Resultado
4 y 5	Para el vértice tope (***T***) Todos los vértices marcados	Tope: B A — Elemento Retirado: T
4 y 5	Para el vértice tope (***A***) Todos los vértices marcados.	Tope: B — Elemento Retirado: A
4 y 5	Para el vértice tope (***B***) Todos los vértices marcados y pila vacía.	Tope: (vacía) — Elemento Retirado: B
6	Recorrido final	***D C R H T A B***

Actividad 61

4.5.2.1 Implementación del recorrido en Profundidad

```
Modulo RecorridoProfundidad( )
Inicio
    *P, *Pl:Pila
    Dato: ItemG
    *Ady: ListaAdy
    *Visitados, *Ant, *Pos: ListaG
    InicializaCola(C)
    InicializaPila(P)
    AñadeP(&P,v)
    Inicializa(&Visitados)
    Mientras(PilaVacía(P) ≠ 1) entonces
        Dato.v ← Tope(P)
        ElimP(&P)
        AñadeC(C,dato.v)
        AñadeVertice(&Visitados, dato)
        EncuentraPosGrafo(G, dato, &Ant, &Pos)
        Si (Pos ≠ Nulo) entonces
            Ady ← Pos->el.Ady
            Mientras (Ady ≠ Nulo)
                Dato.v ← Ady->el.v
                VaciaP(&Pl)
                Si (PerteneceVertice(Visitados, dato) ≠ 1) entonces
                    AñadeP(&pl, dato.v)
                FinSi
                Ady ← Ady->sig
            FinMientras
            Mientras(PilaVacia(Pl) ≠ 1)
                Dato.v ← Tope(Pl)
                ElimP(&Pl)
                AñadeP(&P,dato.v)
            FinMientras
        FnSi
    FinMientras
Termina
```

Actividad 62

Resumen

- **Un árbol binario de búsqueda es aquel en el que para cualquier nodo su valor es *superior* a los valores de los nodos de su subárbol izquierdo e *inferior* a los de su subárbol derecho.**
- **Para la inserción y búsqueda de elementos en un árbol binario sólo es necesario comparar el valor a insertar/buscar y considerar si es mayor insertarlo/buscarlo a la derecha y en otro caso a la izquierda, esto para cada subárbol encontrado.**
- **En la eliminación de un elemento se deben considerar 3 casos con base en el nodo a eliminar:**
    - **Si tiene 0 hijos: se elimina nada más.**
    - **Si tiene 1 hijo: se sube e hijo al padre del nodo a eliminar.**
    - **Si tiene 2 hijos: se sube el hijo más a la derecha del subárbol izquierdo.**

4.6 ALGORITMOS

Es importante mencionar que existe una gran cantidad de problemas de la vida real que son de difícil solución pero aplicando teoría de grafos se facilita esta solución. Este tipo de problemas en ocasiones requiere examinar todos los nodos o todas las aristas de un grafo pero en otras, sólo se necesitan visitar algunos de los nodos o bien algunas de las aristas.

Entre las principales aplicaciones de teoría de grafos está la del cálculo de caminos y la de la obtención del costo mínimo, éstas se implementan con los algoritmos de Dijkstra, Floyd, Warshall, Prim y Kruscal.

4.6.1 Algoritmos para la obtención del camino más corto

Cuando se trabaja con gráficas dirigidas etiquetadas es frecuente buscar el camino más corto entre dos vértices; es decir, el camino que permita llegar desde un vértice origen a un vértice destino recorriendo la menor distancia o con el menor costo. Los algoritmos más usados para este fin son: *Dijkstra, Floyd y Warshall* [3]. Cabe mencionar que aunque es un enfoque a grafos dirigidos, pueden también aplicarse a no dirigidos.

Los tres algoritmos utilizan una matriz de adyacencia etiquetada, donde:

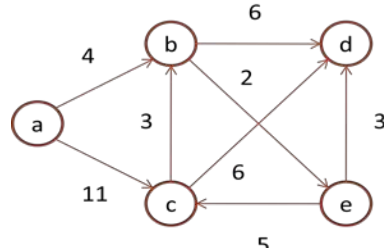
$M[i][j] = 0$, *si* $i = j$
$M[i][j] = \infty$, *si no existe un camino de* i *a* j, *donde* $i \neq j$
$M[i][j]$ = *costo de ir del vértice* i *al vértice* j

A partir de este punto, a la matriz de adyacencia etiquetada se llamará *matriz de distancias o matriz de costos [1, 3].*

4.6.1.1 Algoritmo de Dijkstra: Los caminos más cortos con un solo origen.

Este algoritmo encuentra el camino más corto de un vértice elegido a cualquier otro vértice del grafo, donde la longitud de un camino es la suma de los pesos de las aristas que lo forman. Las aristas deben tener un peso no negativo [3].

Por *ejemplo*, del siguiente grafo,



se puede ver que si se elige el vértice *a* como origen los caminos a los demás vértices son:

Origen	Vértice	Camino(s)	Longitud	
			Suma pesos	Total
a	*b*	***ab***	***4***	***4***
	c	***ac***	***11***	***11***
	d	*abd*	*4+6*	*10*
		acd	*11+6*	*17*
		acbd	*11+3+6*	*20*
		abed	***4+2+3***	***9***
		abecd	*4+2+5+6*	*17*
	e	***abe***	***4+2***	***6***
		acbe	*11+3+2*	*16*

De los cuales se eligen los que tengan la menor longitud que exista del vértice *a* a cada uno de los demás, estos caminos mínimos son los que se encuentran sombreados. Éste es el objetivo del algoritmo de Dijkstra.

Una aplicación de este algoritmo se presenta cuando se desea encontrar la ruta más corta entre dos ciudades; cada vértice representa una ciudad y las aristas representan la duración de los vuelos.

La implementación de este algoritmo, debe considerar: [1, 3]

a) Un arreglo formado por los vértices de los cuales ya se conoce la distancia mínima entre ellos y el origen. Este arreglo inicialmente está formado por el origen como único elemento.

b) Otro arreglo formado por la distancia del vértice origen a cada uno de los otros. Es decir, almacena la menor distancia entre el origen y el vértice *i*. este arreglo se forma en cada paso del algoritmo. Finalmente el arreglo contendrá la distancia mínima entre el origen y cada uno de los otros vértices del grafo.

c) Una matriz de distancias de *nxn* elementos, tal que almacena la distancia o costo entre el vértice *i* y el vértice *j*, si entre ambos existe una arista. En caso contrario el contenido es un valor muy grande ∞.

d) Un vector de predecesores, donde se almacenen los nodos inmediatos anteriores al vértice en cuestión *j* para saber el vértice anterior por medio del cual se llega al camino más corto.

Para esta implementación se propone:

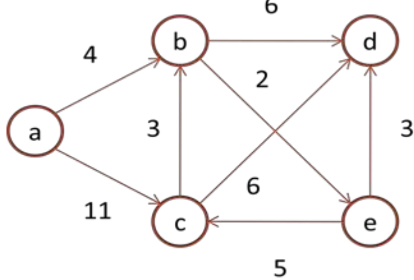
a) Arreglo de vértices: S[n]
b) Arreglo de distancias: D[n]
c) Matriz de distancias: M[n][n]
d) Arreglo de predecesores: P[n]
e) n: es el número de vértices del grafo.

Modulo Dijkstra ( )
Inicia
Agregar el vértice 1 a S
Para i←2, i<=n, i←i+1
Elegir un vértice v en V-S tal que D[v] sea el mínimo valor
Agregar v a S
Repetir para cada vértice w en V-S
D[w] ← mínimo(D[w], D[v] + M[v][w])
FinRepite
FinPara
Termina

El algoritmo anterior se implementa con el siguiente ejercicio.

Ejercicio N° 10

Del siguiente grafo obtener su matriz de distancias e implementar el algoritmo de Dijkstra.



	a	b	c	d	e
a	0	4	11	∞	∞
b	∞	0	∞	6	2
c	∞	3	0	6	∞
d	∞	∞	∞	0	∞
e	∞	∞	5	3	0

Vértices: V = {a,b,c,d,e}

Paso	*S*	*S - V Vértices*	*Camino(s)*	*Grafo*	*Distancia*	*Predecesor P[]*	*Vectores D y P*
0	*a*	*b*	*ab*		*4*	*a*	a b c d e **D:** | 0 | **4** | 11 | ∞ | ∞ | *Menor D[b]: 4 entonces se elige **b** para la siguiente iteración* a b c d e **P:** | | **a** | | | |
		c	*ac*		*11*	*a*	
		d	∞	-	-	-	
		e	∞	-	-	-	
1	*ab*	*c*	*ac*		*11*	*a*	a b c d **D:** | 0 | 4 | 11 | 10 | **6** | *Menor D[b]: 4 ya considerado, entonces* *Menor D[e]:6 se elige **e** para la siguiente iteración* a b c d e **P:** | | **a** | | | **b** |
		d	*abd*		*10*	*b*	
		e	*abe*		*6*	*b*	

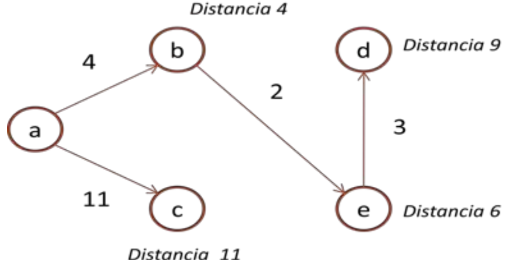
<table>
<tr><th>Paso</th><th>S</th><th>S - V Vértices</th><th>Camino(s)</th><th>Grafo</th><th>Distancia</th><th>Predecesor P[]</th><th>Vectores D y P</th></tr>
<tr><td rowspan="3">2</td><td rowspan="3">abe</td><td>c</td><td>ac</td><td>a 11 c</td><td>11</td><td>a</td><td rowspan="3">a b c d e
D: 0 | 4 | 11 | 9 | 6
Menor D[b]: 4 ya considerado,
Menor D[e]:6 ya considerado
Menor D[d]:9 se elige d para la siguiente iteración
a b c d e
P: | a | | e | b</td></tr>
<tr><td rowspan="2">d</td><td>abd</td><td>a 4 b 6 d</td><td>10</td><td>b</td></tr>
<tr><td>abde</td><td>a 4 b 2 e 3 d</td><td>9</td><td>e</td></tr>
<tr><td>3</td><td>abed</td><td>c</td><td>ac</td><td>a 11 c</td><td>11</td><td>a</td><td>a b c d e
D: 0 | 4 | 11 | 9 | 6
Menor D[c]:11 se elige d final
a b c d e
P: | a | a | e | b</td></tr>
<tr><td>4</td><td colspan="7">abedc</td></tr>
</table>

Con base en la tabla del ejercicio anterior, finalmente el camino más corto del vértice *a* a cada uno de los demás vértices es el obtenido en el vector de distancias considerando también el vector de predecesores.

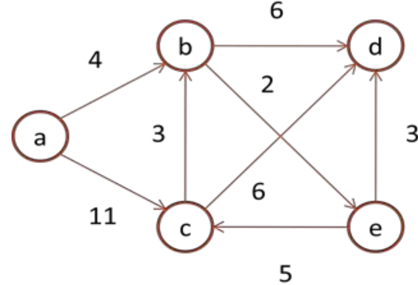
	a	*b*	*c*	*d*	*e*
D:	***0***	***4***	***11***	***9***	***6***

	a	*b*	*c*	*d*	*e*
P:	**-**	***a***	***a***	***e***	***b***

Vértices	Camino	Distancia
a → b	*ab*	*4*
a → c	*ac*	*11*
a → d	*abed*	*9*
a → e	*abe*	*6*



Una vez ejemplificado y entendido paso a paso el ejercicio anterior, una manera simplificada de la implementación del algoritmo se presenta a continuación considerando el mismo grafo.

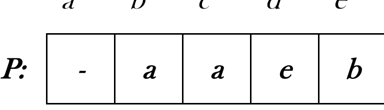


	a	**b**	**c**	**d**	**e**
a	0	4	11	∞	∞
b	∞	0	∞	6	2
c	∞	3	0	6	∞
d	∞	∞	∞	0	∞
e	∞	∞	5	3	0

Vértices: V = {a,b,c,d,e}

Paso	***S***	***D[b], P[b]***	***D[c], P[c]***	***D[d], P[d]***	***D[e], P[e]***
0	*a*	***4,a***	*11,a*	∞	∞
1	*ab*		*11,a*	*10,b*	***6,b***
2	*abe*		*11,a*	***9,e***	
3	*abed*		***11,a***		
4	*abedc*	***4,a***	***11,a***	***9,e***	***6,b***

	a	*b*	*c*	*d*	*e*
D:	***0***	***4***	***11***	***9***	***6***



	a	*b*	*c*	*d*	*e*
P:	**-**	***a***	***a***	***e***	***b***

Actividad 63

4.6.1.2 Algoritmo de Floyd: Los caminos más cortos entre todos los pares de vértices.

Este algoritmo encuentra el camino más corto entre todos los vértices del grafo [1, 3]. Esto mismo se puede realizar mediante el algoritmo de Dijkstra aplicándolo a cada uno de los vértices, sin embargo el algoritmo de Floyd es más directo [2].

Para la implementación de este algoritmo se deben tomar en cuenta las siguientes consideraciones:

- Sea *G=(V,A)* una grafo donde cada arco $u \rightarrow v$ tiene asociado un peso.

- La matriz de adyacencia *A* es la matriz de pesos, de tal forma que todo arco *(vi, vj)* tiene asociado un peso *cij*; si no existe arco cij = ∞. Cada elemento de la diagonal es 0.

- La matriz de distancias D (sirve como punto de partida para este algoritmo) de *nxn* elementos tal que cada elemento *Dij* contenga el costo mínimo de los caminos que van del vértice *i* al vértice *j*. El proceso es el mismo que del algoritmo de Dijkstra.

Para esta implementación se propone:

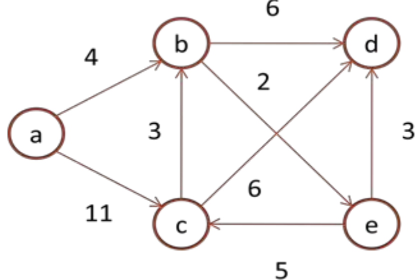
a) *Matriz de Adyacencia: A[n][n]*
b) *Matriz de distancias: D[n][n]*
c) *n: es el número de vértices del grafo.*

```
Modulo Floyd ( )
Inicia
    Para k←1, k<=n, k←k+1
        Para i←1, i<=n, i←i+1
            Para j←1, j<=n, j←j+1
                Si ( (D[i][k] + D[k][j]) < D[i][j] ) entonces
                    D[i][j] ← D[i][k] + D[k][j]
                FinS
            FinPara
        FinPara
    FinPara
Termina
```

El algoritmo anterior se implementa con el siguiente ejercicio.

Ejercicio N° 11

Del siguiente grafo obtener su matriz de distancias e implementar el algoritmo de Floyd para encontrar el camino más corto entre todos los vértices del grafo.



	a	b	c	d	e
a	0	4	11	∞	∞
b	∞	0	∞	6	2
c	∞	3	0	6	∞
d	∞	∞	∞	0	∞
e	∞	∞	5	3	0

Recordar que se parte de la matriz de adyacencia, y para cada iteración la matriz resultante antecesora.

<table>
<tr><th colspan="6">Vértice Intermedio a
(Renglón de a queda igual)</th></tr>
<tr><th>Cada vértice</th><th>Cada columna</th><th>Camino</th><th>Grafo</th><th>Distancia</th><th>Matriz de distancias</th></tr>
<tr><td rowspan="5">b</td><td>a</td><td>baa</td><td>No existe</td><td>∞</td><td rowspan="5"><table><tr><th></th><th>a</th><th>b</th><th>c</th><th>d</th><th>e</th></tr><tr><th>a</th><td>0</td><td>4</td><td>11</td><td>∞</td><td>∞</td></tr><tr><th>b</th><td>∞</td><td>0</td><td>∞</td><td>6</td><td>2</td></tr><tr><th>c</th><td>∞</td><td>3</td><td>0</td><td>6</td><td>∞</td></tr><tr><th>d</th><td>∞</td><td>∞</td><td>∞</td><td>0</td><td>∞</td></tr><tr><th>e</th><td>∞</td><td>∞</td><td>5</td><td>3</td><td>0</td></tr></table></td></tr>
<tr><td>b</td><td>bab</td><td>-</td><td>0</td></tr>
<tr><td>c</td><td>bac</td><td>No existe</td><td></td></tr>
<tr><td>d</td><td>bad = bd</td><td>b 6 d</td><td>6</td></tr>
<tr><td>e</td><td>bae = be</td><td>b 2 e</td><td>2</td></tr>
<tr><td rowspan="5">c</td><td>a</td><td>caa</td><td>No existe</td><td>∞</td><td rowspan="5"><table><tr><th></th><th>a</th><th>b</th><th>c</th><th>d</th><th>e</th></tr><tr><th>a</th><td>0</td><td>4</td><td>11</td><td>∞</td><td>∞</td></tr><tr><th>b</th><td>∞</td><td>0</td><td>∞</td><td>6</td><td>2</td></tr><tr><th>c</th><td>∞</td><td>3</td><td>0</td><td>6</td><td>∞</td></tr><tr><th>d</th><td></td><td></td><td></td><td></td><td></td></tr><tr><th>e</th><td></td><td></td><td></td><td></td><td></td></tr></table></td></tr>
<tr><td>b</td><td>cab</td><td>No existe, pero se considera valor de la matriz original</td><td>3</td></tr>
<tr><td>c</td><td>cac</td><td>-</td><td>0</td></tr>
<tr><td>d</td><td>cad = cd</td><td>d 6 c</td><td>6</td></tr>
<tr><td>e</td><td>cae</td><td>No existe</td><td>∞</td></tr>
</table>

<table>
<tr><th colspan="6">Vértice Intermedio a
(Renglón de a queda igual)</th></tr>
<tr><th>Cada vértice</th><th>Cada columna</th><th>Camino</th><th>Grafo</th><th>Distancia</th><th>Matriz de distancias</th></tr>
<tr><td rowspan="3">d</td><td>a</td><td>daa</td><td>No existe</td><td>∞</td><td rowspan="3"><table><tr><th></th><th>a</th><th>b</th><th>c</th><th>d</th><th>e</th></tr><tr><th>a</th><td>0</td><td>4</td><td>11</td><td>∞</td><td>∞</td></tr><tr><th>b</th><td>∞</td><td>0</td><td>∞</td><td>6</td><td>2</td></tr></table></td></tr>
<tr><td>b</td><td>dab</td><td>No existe</td><td>∞</td></tr>
<tr><td>c</td><td>dac</td><td>No existe</td><td>∞</td></tr>
</table>

Vértice Intermedio a (*Renglón de **a** queda igual*)					
Cada vértice	**Cada columna**	**Camino**	**Grafo**	**Distancia**	**Matriz de distancias**
	d	dad	-	0	**c** ∞ 3 0 6 ∞ **d** ∞ ∞ ∞ 0 ∞
	e	dae	No existe	∞	**e**
e	a	eaa	No existe	∞	
	b	eab	No existe	∞	**a b c d e**
	c	eac = ec		5	**a** 0 4 11 ∞ ∞ **b** ∞ 0 ∞ 6 2
	d	ead = ed		3	**c** ∞ 3 0 6 ∞ **d** ∞ ∞ ∞ 0 ∞ **e** ∞ ∞ 5 3 0
	e	eae	-	0	

La matriz anterior se refiere a la matriz de entrada o de inicio.

Vértice Intermedio b (*Renglón de **b** queda igual*)					
Cada vértice	**Cada columna**	**Camino**	**Grafo**	**Distancia**	**Matriz de distancias**
a	a	aba	-	0	
	b	abb = ab		4	**a b c d e**
	c	abc = ac		11	**a** 0 4 11 10 6 **b** ∞ 0 ∞ 6 2
	d	abd		10	**c** ∞ 3 0 6 ∞ **d** ∞ ∞ ∞ 0 ∞ **e** ∞ ∞ 5 3 0
	e	abe		6	
c	a	cba	No existe	∞	**a b c d e**

Vértice Intermedio b
(*Renglón de **b** queda igual*)

Cada vértice	Cada columna	Camino	Grafo	Distancia
	b	cbb =cb	b, 3, c	3
	c	cbc	-	0
	d	cbd = cd	d, 6, c	6
	e	cbe	b, 3, 2, c, e	5

Matriz de distancias

a	0	4	11	10	6
b	∞	0	∞	6	2
c	**∞**	**3**	**0**	**6**	**5**
d					
e					

Cada vértice	Cada columna	Camino	Grafo	Distancia
d	a	dba	No existe	∞
	b	dbb	No existe	∞
	c	dbc	No existe	∞
	d	dbd	-	0
	e	dbe	No existe	∞

	a	b	c	d	e
a	0	4	11	10	6
b	∞	0	∞	6	2
c	∞	3	0	6	5
d	**∞**	**∞**	**∞**	**0**	**∞**
e					

Cada vértice	Cada columna	Camino	Grafo	Distancia
e	a	eba	No existe	∞
	b	ebb	No existe	∞
	c	ebc = ec	c, e, 5	5
	d	ebd = ed	d, 3, e	3
	e	ebe	-	0

	a	b	c	d	e
a		4	11	10	6
b	∞	0	∞	6	2
c	∞	3	0	6	5
d	∞	∞	∞	0	∞
e	**∞**	**∞**	**5**	**3**	**0**

Comparando la matriz de inicio

	a	b	c	d	e
a	0	4	11	∞	∞
b	∞	0	∞	6	2
c	∞	3	0	6	∞
d	∞	∞	∞	0	∞
e	∞	∞	5	3	0

Con la obtenida en intermedio b,

	a	b	c	d	e
a	0	4	11	**10**	**6**
b	∞	0	∞	6	2
c	∞	3	0	6	**5**
d	∞	∞	∞	0	∞
e	∞	∞	5	3	0

los caminos encontrados son: ***abd, abe, cbe.***

Así para cada vértice intermedio de *c, d y e.*

Simplificando los pasos, las matrices siguientes serían*:*
Vértice intermedio c

	a	b	c	d	e
a	0	4	11	10	6
b	∞	0	∞	6	2
c	∞	3	0	6	5
d	∞	∞	∞	0	∞
e	∞	**8**	5	3	0

Cabe mencionar que para el valor de:

- D[a][b] se tenían dos posibles caminos:
  - *acb* con distancia 14, *ab* con 4, se elige el menor que es 4
- D[a][d] se tenían dos posibles caminos:
  - *acd* con distancia 17, *abd* con 10, se elige el menor que es 10
- D[e][b] se tenían dos posibles caminos:
  - *ecd* con distancia 11, *ed* con 3, se elige el menor que es 3

Los caminos encontrados son: ***ecb.***

Vértice intermedio d

	a	b	c	d	e
a	0	4	11	10	6
b	∞	0	∞	6	2
c	∞	3	0	6	5
d	∞	∞	∞	0	∞
e	∞	8	5	3	0

No se encontraron otros caminos.

Vértice intermedio e

	a	b	c	d	e
a	0	4	11	**9**	6
b	∞	0	**7**	**5**	2
c	∞	3	0	6	5
d	∞	∞	∞	0	∞
e	∞	8	5	3	0

- D[a][d] se tenían dos posibles caminos:
    - *abd* con distancia 10, *abed* con 9, se elige el menor que es 9

Los caminos encontrados son: ***aed, bec y bed.***

La matriz final de distancias entre vértices es

	a	b	c	d	e
a	0	4	11	**9**	6
b	∞	0	**7**	5	2
c	∞	3	0	6	5
d	∞	∞	∞	0	∞
e	∞	8	5	3	0

Actividad 64

4.6.1.3 Algoritmo de Warshall: Matriz de caminos.

Este algoritmo identifica la existencia de un camino de longitud igual a uno o mayor, entre cada uno de los vértices del grafo dirigido. Es decir, la solución encontrada por el algoritmo no presenta distancias entre los vértices, sólo muestra si hay o no camino entre ellos [3].

Este algoritmo se basa en un concepto llamado *cerradura transitiva* de la matriz de adyacencia para determinar la *matriz de caminos* [1-3].

Sea el grafo dirigido *G(V,A)* y su matriz de adyacencia ***M***, donde:

M[i][j] = 1, si hay un arco de i a j
M[i][j] = 0, si no hay un arco de i a j

La cerradura de *M* es *A* tal que:
A[i][j] = 1 si hay un camino de longitud mayor o igual que 1 de i a j
A[i][j] = 0 en otro caso.

El algoritmo permite establecer que existe un camino del vértice *i* al vértice *j* que no pasa por un número de vértices mayor que *k* si: [1-3]

- Ya existe un camino de *i* a *j* que no pasa por un número de vértices mayor que *k-1*
- Hay un camino de *i* a *k* que no pasa por un número de vértices mayor que *k-1*, y hay un camino de *k* a *j* que no pasa por un número de vértices mayor que *k-1*

Para esta implementación se propone:

a) *Matriz de Cerradura Transitiva: A[n][n]*
b) *Matriz de Adyacencia: M[n][n]*
c) *n: es el número de vértices del grafo.*

Modulo Warshall ()
Inicia

```
Para k←1, k<=n, k←k+1
    Para i←1, i<=n, i←i+1
        Para j←1, j<=n, j←j+1
            Si (A[i][j] = 0) entonces
                A[i][j] ← A[i][k] y A[k][j]
            FinS
        FinPara
    FinPara
FinPara
```

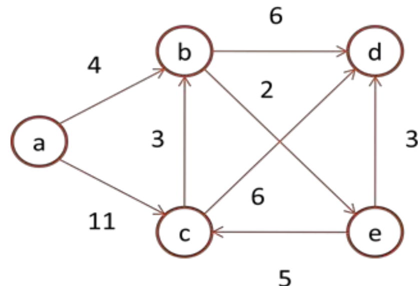
Termina

A es una matriz que representa la cerradura transitiva de *M*. *A[i][j]* será igual a 1 si hay un camino de longitud mayor o igual que 1 de *i* a *j*, o 0 en otro caso. Inicialmente *A* es igual a *M*.

El algoritmo anterior se implementa con el siguiente ejercicio.

Ejercicio N° 12

Se presenta la aplicación del algoritmo de Warshall para identificar si existe o no un camino entre todos los vértices de un grafo.



Matriz de adyacencia

	a	b	c	d	e
a	0	1	1	0	0
b	0	0	0	1	1
c	0	1	0	1	0
d	0	0	0	0	0
e	0	0	1	1	0

Inicialmente $A = M$ entonces

A:

	a	**b**	**c**	**d**	**e**
a	0	1	1	0	0
b	0	0	0	1	1
c	0	1	0	1	0
d	0	0	0	0	0
e	0	0	1	1	0

Intermedio b,

A:

	a	**b**	**c**	**d**	**e**
a	0	1	1	**1**	**1**
b	0	0	0	1	1
c	0	1	0	1	**1**
d	0	0	0	0	0
e	0	0	1	1	0

los caminos encontrados son: ***abd, abe, cbe.***

Intermedio c,

A:

	a	**b**	**c**	**d**	**e**
a	0	1	1	1	1
b	0	0	0	1	1
c	0	1	0	1	1
d	0	0	0	0	0
e	0	**1**	1	1	**1**

los caminos encontrados son: ***ecb, ece.***

Intermedio d, no hay caminos, el renglón son columnas de 0.

Intermedio e,

A:

	a	**b**	**c**	**d**	**e**
a	0	1	1	1	1
b	0	**1**	**1**	1	1
c	0	1	**1**	1	1
d	0	0	0	0	0
e	0	1	1	1	1

los caminos encontrados son: ***beb, bec, cec.***

✍ Actividad 65

4.6.2 Algoritmos para la obtención de costo mínimo. Problema del árbol de expansión (abarcador) de costo mínimo.

Este tipo de algoritmos se aplica principalmente sobre grafos no dirigidos. También llamado árbol abarcador.

Un árbol abarcador de un grafo *G* se define como un árbol libre que conecta todos los vértices de *V*. El costo del árbol abarcador resulta de la suma de las aristas incluidas en él. Un árbol abarcador de costo mínimo es el construido con las aristas de menor costo [1, 3], es decir, es un árbol que contiene a todos los vértices de una red y tal que la suma de los pesos de sus arcos es mínima.

Una aplicación típica de los árboles abarcadores de costo mínimo es el diseño de redes de comunicación [3].

Sea *G* un grafo no dirigido tal que todos sus arcos son positivos.
Un árbol en una red es un Subgrafo *G'* del grafo *G* que es conectado y sin ciclos. Los árboles tienen dos propiedades importantes: [1]

a) Todo árbol de *n* vértices contiene exactamente *n-1* arcos.
b) Si se añade un arco a un árbol de expansión entonces resulta un ciclo

Entre estos algoritmos se encuentran los algoritmos de Prim y Kruskal.

4.6.2.1 Algoritmo de Prim

Este algoritmo permite encontrar el árbol abarcador de costo mínimo de un grafo. Para ello utiliza dos conjuntos:

- *V:* conjunto de todos los vértices.
- *U:* conjunto auxiliar inicializado con el primer vértice.

Las consideraciones para la implementación de este algoritmo son:

- En cada iteración del algoritmo se busca la arista *(u,v)* que conecte *U* con el Subgrafo *V-U*.
- Se agrega el nodo *v* perteneciente a *V-U*.
- Este proceso se repite hasta que *U=V*

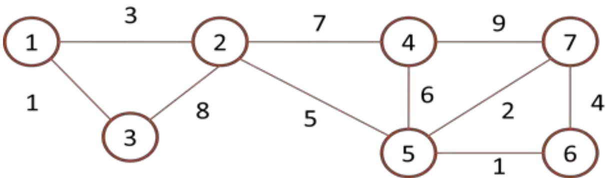
Este algoritmo sigue la metodología de hacer en cada iteración "lo mejor que se puede hacer", esta metodología se llama *"algoritmo voraz"* [2]

Modulo Prim ()
Inicia
Mientras (V ≠ U)
Elegir una arista (u,v) ϵ A(G) tal que su costo sea mínimo, u ϵ U y v ϵ V-U
Agregar la arista (u,v) a L, o se van marcando los nodos ya seleccionados
Agregar el nodo v a U
FinMientras
Termina

Donde:

- *V* es el conjunto de vértices: V={1,2,..,n}
- *U* es un subconjunto propio del conjunto *V*, siendo su valor inicial el del primer vértice.
- *L* es una lista de aristas que se va formando a medida que las aristas de menor costo se van seleccionando. Inicialmente *L* está vacía: *L=Ø,* o se van marcando los nodos ya seleccionados.
- *N* es el número de vértices del grafo.

Ejercicio N° 13



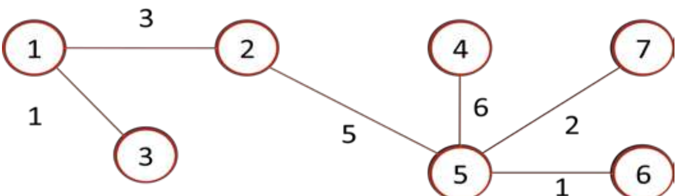
Se presenta la aplicación del algoritmo de Prim para determinar el árbol de expansión de costo mínimo del siguiente grafo.

Se parte del vértice **1.**

Caminos	Valor	Arista elegida	Grafo	Árbol de expansión
1-2	3	(1,3)		
1-3	1			
1-2	3	(1,2)		
1-3	8			
2-4	7	(2,5)		
2-5	5			
5-6	1	(5,6)		

Caminos	Valor	Arista elegida	Grafo	Árbol de expansión
5-7	2			
5-7	2	(5,7)		
6-7	4			
5-4	6	(5,6)		
7-4	9			
Termina porque todos los vértices ya están marcados.				

El árbol de expansión del costo mínimo es:



Actividad 66

4.6.2.2 Algoritmo de Kruskal

Kruskal propone otra estrategia para encontrar el árbol de expansión. La construcción del árbol se realiza al agregar repetidamente una arista al árbol de expansión y seleccionar siempre el de menor costo.

El proceso para este algoritmo es:

- Obtener una serie de particiones a partir del conjunto de vértices *V*. Las particiones tendrán tamaño uno.
- A partir de este paso se busca la arista de menor costo y si ésta une a dos vértices que pertenecen a particiones diferentes, dichas particiones se reemplazan por su unión.
- En caso contrario, la arista no forma parte del árbol de expansión ya que produciría un ciclo.
- Se continua eligiendo la arista *(u,v)* de menor costo y uniéndolas particiones a las cuales pertenecen *u* y *v* respectivamente, hasta que se tenga una sola partición formada por todos los vértices de la gráfica.

Modulo Kruskal( )
Inicia

Mientras (existan vértices en P que se encuentren en particiones distintas)
Seleccionar la arista (u,v) de menor costo
Si la arista conecta dos vértices que se encuentran en particiones diferentes entonces
Unir las particiones a las cuales pertenecen u y v
FinSi
FinMientras

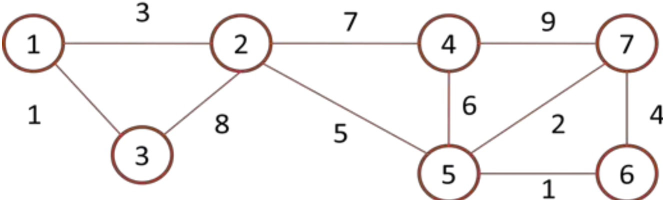
Termina

Donde:

- *V* es el conjunto de vértices: V={1,2,..,n}
- *P* es la serie de particiones obtenidas a partir de *V*. Inicialmente P = {{1}, {2},...,{n}}.
- *N* es el número de vértices del grafo.

Ejercicio N° 14

Se presenta la aplicación del algoritmo de Kruskal para determinar el árbol de expansión de costo mínimo del siguiente grafo.

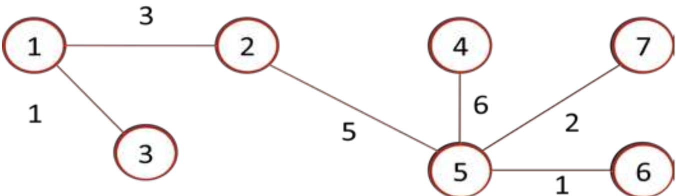


Se van eligiendo los caminos de menor costo en cada iteración.

Camino Menor	*Valor*	*Decisión*	*Grafo*
1-3	*1*	*Se marcan*	
5-6	*1*	*Se marca*	
5-7	*2*	*Se marca ya que no forma ciclos con ninguna arista de las ya marcadas.*	

Camino Menor	*Valor*	*Decisión*	*Grafo*
1-2	*3*	*Se marca ya que no forma ciclos con ninguna arista de las ya marcadas.*	
6-7	*4*	*Se desecha ya que formaría ciclos con las aristas (5,7) y (5,6)*	--
2-5	*5*	*Se marca ya que no forma ciclos con ninguna arista de las ya marcadas.*	
4-5	*6*	*Se marca ya que no forma ciclos con ninguna arista de las ya marcadas.*	
Termina ya que todos los vértices ya están marcados			

El árbol de expansión mínimo es:



Actividades 67 y 68

Resumen

- Entre las principales aplicaciones de teoría de grafos está la del cálculo de caminos y la de la obtención del costo mínimo, éstas se implementan con los algoritmos de Dijkstra, Floyd, Warshall, Prim y Kruscal.
- Para la obtención del camino más corto entre vértices están los algoritmos de Dijkstra, Floyd y Warshall.
- Para la obtención del costo mínimo o árbol abarcador están los algoritmos de Prim y Kruskal.

REFERENCIAS

1. Joyanes, L., et al., *Estructuras de datos en C*. 2005, Madrid: Mc Graw Hill, Serie Schaum.

2. Joyanes, L. and I. Zahonero, *Estructura de Datos. Algoritmos, Abstracción y Objetos*. 1998, Madrid: McGraw-Hill.

3. Cairó, O. and S. Guardati, *Estructuras de Datos*. 2a. Edición ed. 2002, México: Mc Graw Hill.

REFERENCIAS

1. Albarrán Silvia E. y Salgado Mireya. Apuntes de Estructuras de datos. UAEM, 2011.
2. Cairó, Osvaldo y Guardati Silvia. Estructuras de Datos. 3ª ed. McGraw-Hill. México, 2006.
3. Franch Gutiérrez, Xavier. Estructuras de datos. Especificación, diseño e implementación, Ediciones de la UPC, S.L., 2004.
4. Garrido, Antonio y Fernández Joaquín, Abstracción y Estructuras de Datos en C++, Delta Publicaciones, 2006.
5. Joyanes, Aguilar Luis; Zahonero, Martínez Ignacio. Estructura de Datos. Algoritmos y estructuras de datos: una perspectiva en C. McGraw-Hill, Madrid, 2004.
6. Luján Mora Sergio, Ferrández Rodríguez Antonio, Peral Cortés Jesús, Requena Jiménez Antonio. Ejercicios resueltos sobre Programación y estructuras de datos. Universidad de Alicante, 2014.
7. Narciso Martí Oliet, Yolanda Ortega Mallén, José Alberto Verdejo López. Estructuras de datos y métodos algorítmicos: ejercicios resueltos. Pearson Educación, 2004
8. Rodríguez Artalejo, González Caldero, Gómez Martin, Estructuras de datos. Un enfoque moderno, Editorial Complutense, 2011.

ACTIVIDADES

UNIDAD DE COMPETENCIA I
Reconocer y manejar las variables dinámicas.

1.1 Estructuras de datos
INSTRUCCIONES: Con base en el archivo Temas Básicos realiza las actividades 1 – 7.

Actividad 1: Identificar Tipos de Datos

Actividad 2: Diagramas de Flujo

Actividad 3: Pseudocódigos

Actividad 4: Estructuras Repetitivas

Actividad 5: Vectores y Matrices

Actividad 6: Registros

Actividad 7: Realizar un mapa mental (con freemind o novamind) de los temas anteriores.

1.2 Abstracción, estructuras de datos y representación.
Actividad 8: Realizar una investigación de cómo puede definirse un **TAD** en 3 lenguajes de programación.

1.3 Estructuras de datos
Actividad 9: Considerando el siguiente código en lenguaje C y realiza las instrucciones que se presentan a continuación.

```
1.  #include <stdio.h>
2.  #include <conio.h>
3.  void main(){
    4.  int *x, y;      /*se declara x de tipo apuntador entero y y de tipo entero*/

    5.  clrscr();
    6.  printf("Dame un número: ");
    7.  scanf("%d", &y);
    8.  x = &y;
    9.  printf("%p : %d \n",x,y);
    10. printf("%d : %p \n",*x,&y);
    11. getch();
12. }
```

Instrucciones

1. Escribe el programa en el editor de C.
2. Compila y corrige los errores de edición (si los hubiera).
3. Visualizar en el área del *Watch* las variables *x,*x, y* y *&y*
4. Rastrea línea a línea el programa (F7), considera el valor de *y* como 150 y detente en la línea 8, es decir, antes de ejecutar la instrucción *scanf("%d", &y);*
5. Cuáles son los valores de *x,*x, y* y *&y,* anótalos y captura la pantalla como imagen.
6. Ejecuta la línea 8 (*x = &y;)*, ahora cuáles son los valores de *x,*x, y* y *&y,* anótalos y captura la pantalla como imagen.
7. *Qué valor le asignó a x* la instrucción de la línea 8?
8. Ejecuta las instrucciones 9 y 10 y escribe los resultados que imprime en pantalla y captura la pantalla como imagen.
9. Cuál es la diferencia de las variables *x* y *y* en la instrucción número 4?
10. Qué número de instrucción imprime la localidad de memoria de *x* y el contenido de *y*?
11. Escribe dos instrucciones en el programa antes de la instrucción *getch(),* una que permita visualizar las direcciones de memoria de *x* y *y*, y la otra que permita visualizar los contenidos de *x* y *y*.
12. Corre el programa con las nuevas instrucciones y captura la pantalla final como imagen.
13. Entrega esta actividad con las imágenes de las pantallas capturadas al igual que el código del programa

Actividad 10

Instrucciones

Con base en las operaciones de desplazamiento de apuntadores y utilizando apuntadores, realiza un programa que permita determinar el tamaño (número de bytes) de los tipos de datos *entero(int), flotante(float), entero largo (long int) y doble (double)* en el lenguaje C (No utilizar la función *sizeof*, determinar el tamaño utilizando una operación de desplazamiento de apuntadores).

Actividad 11

```
#include <stdio>
#include <conio>

void main()
int *num, *ppio;

clrscr();
ppio = num
puts("Dame un número");
scanf("%",num);
while(*num <> -1){
        num++;
        scanf("Dame un número");
        scanf("%d",num)
}
num--;
clrscr();
while(num >= ppio)
        printf("%p : % \n",num,*num);
        num-;
}
num = ppio;
printf("\n\n\n);
while(num != -1){
        printf("%p                :
        %d\n",num,*num);
        num+;
}
getch();
}
```

Instrucciones

1. Este programa pide números hasta que teclea el usuario un -1 y los imprime en forma inversa y después de forma en que fueron introducidos.
2. Edita el programa en C.
3. Compila y corrige los errores de sintaxis propios del lenguaje y del uso de apuntadores que tiene el programa.
4. Corre el programa con los datos **21 32 53 47 55 61 17 88 90 110 -1** para que visualices su resultado.
5. Rastrea el programa (F7) visualizando en el área del *watch* la variable *num, *num* y *ppio,* escribe cuál es la diferencia de éstas.
6. Qué sucede con la variable *num* con las instrucciones *num--* y *num*++
7. Para qué sirvió la variable *ppio* en el programa.
8. Entrega tus respuestas asimismo una imagen de la pantalla del programa corrido.

Actividad 12

Archivo de cabecera ***<stdio.h>***
Archivo de cabecera ***<conio.h>***
Funciones sin valor de retorno ***Valor(Entero, Entero), Referencia(entero*, entero*)***

Principal
Inicio
    x, y : E
    limpiar pantalla
    x ← 5
    y ← 7
    Valor(x,y)
    Referencia*(&x,&y)*
    Lee una tecla
Termina

Módulo Valor*(a:E, b:E)*
Inicio
    a ← a+10
    b ← b+10
FinModulo
Módulo Referencia*(*a:E, *b:E)*
Inicio
    **a ← (*a)+10*
    **b ← (*b)+10*
FinModulo

Instrucciones

1. Codifica en lenguaje C.
2. Rastrea línea a línea el programa (F7)
3. Visualiza en el área del *Watch* las variables *x,y,a,b*
4. Escribe cuáles son los resultados de *x* y *y* después de haber ejecutado la función Valor, captura la imagen de la pantalla.
5. Escribe cuáles son los resultados de *x* y *y* después de haber ejecutado la función Referencia, captura la imagen de la pantalla..
6. Modifica el programa anterior a fin de que después de la llamada de cada función (Valor y Referencia) se puedan visualizar los resultados anteriores, auxíliate de las siguientes instrucciones en pseudocódigo.
   Escribir ("VALOR: x=", *x*, "y=", *y*)
   Escribir ("REFERENCIA: x= ",*x*," y= ",*y*)
7. Captura la imagen de la pantalla final del programa ya corrido.
8. Entrega el listado del código fuente asimismo el reporte los resultados y las imágenes solicitadas.

UNIDAD DE COMPETENCIA II
Aplicar las principales estructuras de datos lineales

2.1 Pilas: representación. Operaciones (inserción, eliminación, pila llena, pila vacía). Aplicaciones.

Actividad 13: Insertar y Quitar elementos de una pila

Actividad 14: Operaciones de una pila

Actividad 15

Instrucciones

1. Codifica las implementaciones de las operaciones de una pila en un lenguaje de programación.
2. Crea un programa principal que permita a través de un menú hacer el llamado a los módulos anteriores 1.Insertar 2.Quitar 3.Tope 4.Salir, considera los datos que debes pedir para poder ser enviados en los módulos que requieren parámetros, asimismo considera los que regresan valor. Se termina el programa cuando se teclee la opción Salir.
3. Ejecuta el programa y captura la pantalla final de la corrida de éste.
4. Implementa al programa anterior una función que permita desplegar los elementos de la pila desde del último al primer elemento, recuerda que el último que entra es el primero que sale e integra como opción 4 la de Consultar.
5. Ejecuta el programa y captura la pantalla final de la corrida de éste.
6. Entrega el listado del código fuente de tu programa junto con las pantallas capturadas, asimismo envía a msg.estructurasdedatos@gmail.com tu programa ejecutable (sin virus, en caso de que tenga virus queda anulada la calificación obtenida en esta Actividad).

Actividad 16

Instrucciones

1. Basándote en el programa (sólo hasta el inciso 2) de la actividad 17 realiza una animación simulando estas funciones.

Actividad 17

```
Pila: Registro
        datos[MAX]: S
        tope: E
FinRegistro

Principal
        p: Pila
        i,palmo: E
        pal[100]:S
        car: C

        Inicializar(&p)
        Escribir("Dame una palabra:")
        Leer(pal)
        i ← 1
        Repite
                car ← pal[i]
                Insertar(&p,car)
                i← i+1
        Hasta(pal[i] =Fin de palabra)
        Insertar(&p,'\0')
        p.tope←p.tope - 1
        palmo ← 1
        Para (i=1, palmo ≠ 0 y PilaVacia(&p) ≠ 1)
                Si (pal[i] = Quitar(&p)) entonces
                        i←i+1
                        palmo ← 1
                Otro
                        palmo ← 0
                FinSi
        FinPara
        Si (palmo = 1) entonces
                Escribir("La palabra ", pal, " SI es Palindrome")
        Otro
                Escribir("La palabra ", pal," NO es Palindrome")
        FinSi
Termina
```

Instrucciones
Este programa es el programa principal de la lectura de una palabra para almacenarla caracter por caracter en una pila y determinar si es palíndroma o no, auxíliate de éste (si así lo consideras) para las siguientes instrucciones:

1. Implementa (en cualquier lenguaje de programación) las operaciones de una pila para realizar un programa que lea una **frase** y determine si ésta es palíndroma o no. Considera que estás trabajando con frases entonces debes eliminar (no considerar) los espacios, esto es, cuando insertes caracter por caracter en la pila no debes insertar los espacios. Tampoco consideres mayúsculas ni minúsculas ni acentos.
2. Ejecuta el programa con los ejemplos:
    a. **animo romina**
    b. **la casa es grande**
    c. **ella te dara detalle**

    captura la pantalla final de cada uno de los ejemplos anteriores.
3. Envía el listado del código fuente, las pantallas capturadas y el programa ejecutable a msg.estructurasdedatos@gmail.com (sin virus, en caso de que tenga virus queda anulada la calificación obtenida en esta Actividad).

2.2 Colas: Representación. Operaciones (inserción, eliminación, cola llena, cola vacía). Cola circular. Aplicaciones.

Actividad 18: Insertar y Quitar elementos de una cola

Actividad 19: Operaciones de una cola

Actividad 20
Instrucciones

1. Codifica las implementaciones anteriores en un lenguaje de programación.
2. Crea un programa principal que permita a través de un menú hacer el llamado a los módulos anteriores 1.Insertar 2.Quitar 3.Frente 4.Salir, considera los datos que debes pedir para poder ser enviados en los módulos que requieren parámetros, asimismo considera los que regresan valor. Se termina el programa cuando se teclee la opción Salir.
3. Ejecuta el programa y captura la pantalla final de la corrida de éste.
4. Implementa al programa anterior una función que permita desplegar los elementos de la cola desde del primer al último elemento, recuerda que el primero que entra es el primero que sale e integra como opción 4 la de Consultar.

2.3 Cola circular

Actividad 21-22: Operaciones de una cola circular

Actividad 23
Instrucciones

1. Codifica las implementaciones de las operaciones de una cola circular en un lenguaje de programación.

2. Crea un programa principal que permita a través de un menú hacer el llamado a los módulos anteriores 1.Insertar 2.Quitar 3.Salir, considera los datos que debes pedir para poder ser enviados en los módulos que requieren parámetros, asimismo considera los que regresan valor. Se termina el programa cuando se teclee la opción Salir.
3. Para cada función Insertar y Quitar, integra una función que permita imprimir los elementos de la cola con el fin de apreciar los movimientos que se realizan en ella con cada una de estas funciones, que no se borren los cambios en la pantalla.
4. Ejecuta el programa y captura la pantalla final de la corrida de éste.
5. Realiza un reporte de las diferencias que encontraste al realizar una cola simple y una circular.

2.4 Listas: Representación. Operaciones (inserción, eliminación, recorrido, búsqueda). Listas simplemente y doblemente ligadas. Lista circular, lista doble, lista doble circular. Aplicaciones.

Actividad 24: Esquema de una lista simplemente enlazada

Actividad 25: Operaciones de una lista simplemente enlazada

Actividad 26

Instrucciones

1. Codifica las implementaciones de las operaciones de una lista simplemente enlazada en un lenguaje de programación.
2. Realiza un programa que a través de listas simplemente enlazadas permita:
    1) **Insertar** (ordenadamente, auxíliate de *InsertarPrim, InsertarFin, Insertar2*)
    2) **Eliminar** (pedir el nodo a eliminar, auxíliate de *EliminarPrim, EliminarFin, Eliminar2*)
    3) **Consultar** (desplegar cada uno de los elementos de la lista)
    4) **Desplegar** (la memoria de la computadora simulada como en el ejercicio Nº 5 de esta sección, sólo imprimir:

	info	*sig*
0011	**7**	**0014**
0012	**14**	**001D**
0013	**…**	

	Info	*sig*
0018	**…**	
0019		
001A	**10**	**001C**

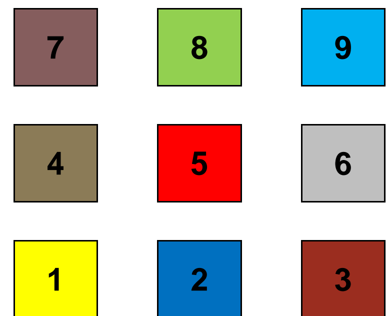
Ésta se desplegará en el momento en que el usuario lo desee.

5) Salir

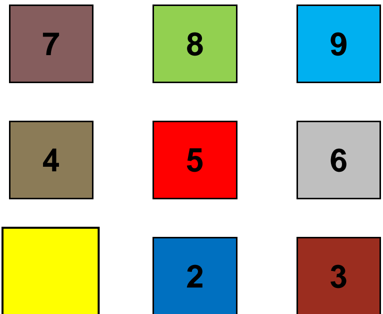
Actividad 27

Instrucciones

1. Realiza un programa que a través de listas enlazadas permita:
    a. Jugar memoria secuencial con la computadora y un jugador:
        i. Primer turno: A través de un número aleatorio (1-9) la computadora despliega el primer número
        ii. Turno dos: El usuario debe repetir ese número y otro más (las teclas a utilizar son las del bloque numérico)
        iii. Turno tres: la computadora teclea los dos primeros números y uno más
        iv. En los turnos siguientes computadora o usuario van dando la secuencia de números e incrementan un número.
        v. El juego termina hasta se pierda o bien con la tecla ESC
    b. Decir quien ganó
    c. Imprimir la secuencia correcta de números
2. Trata de implementarlo con gráficos, por ejemplo:



Esto simularía el teclado numérico, si la computadora genera o el usuario da el número 1 entonces en pantalla tendría que verse más grande para identificar que éste es el presionado y después volver a tamaño normal, de la misma manera que cuando se presente la corrida y espere a agregar un número más. Cuando se muestre la secuencia de números (colores) correcta tendrá que irse "encendiendo" cada color



3. Envía a la dirección de trabajo, el código fuente de tu programa asimismo como una secuencia de pantallas (mínimo 6) de una corrida del programa.

Actividad 28

Instrucciones

1. Realiza los algoritmos de implementación de las operaciones de *Pila Vacía, Visualizar* y *Cima* de una pila utilizando estructuras dinámicas. Sólo en pseudocódigo.

Actividad 29: Expresiones

Actividad 30: Expresiones Postfija y Prefija

Actividad 31: Algoritmo expresión posfija

Actividad 32

Instrucciones

1. Realiza un programa en cualquier lenguaje de programación (excepto lenguaje C) que permita la evaluación de una expresión, es decir, este programa debe leer la expresión a evaluar, desplegar la expresión transformada a postfija y finalmente desplegar el resultado de la evaluación, todo esto a través de del menú:
    a. Expresión
    b. Postfija
    c. Evaluación
    d. Salir
2. Enviar el código fuente y pantallas capturadas de la ejecución del programa.

Actividad 33: Esquema de una lista doblemente enlazada

Actividad 34: Operaciones de una lista doblemente enlazada

Actividad 35

Instrucciones

1. Implementa una función que a partir de una “tabla” de datos permita insertar y crear una lista doblemente enlazada dependiendo de los datos dados por el usuario, por ejemplo:
    a. Al usuario se le pedirán 2 datos: *DatoIzq, Valor*
    b. Donde *DatoIzq* es el nodo anterior al que se va a ligar, *Valor* es el nuevo nodo.
    c. Para el inciso anterior:
        i. verificar si la lista está vacía (si es el primer dato sólo pedirá el *valor* del nuevo nodo)
        ii. verificar que exista *DatoIzq* (sólo así se podrá insertar el nuevo nodo)
2. Implementa dos funciones que permitan desplegar una lista doblemente enlazada a través de:
    a. Desplegar cabeza -> final
    b. Desplegar final -> cabeza
3. Con las 3 funciones anteriores crear un menú:
    a. Insertar
    b. Desplegar de cabeza
    c. Desplegar de final
    d. Salir
4. Enviar código fuente del programa y las pantallas capturadas (mínimo 3) de una corrida del programa.

Actividad 36-37: Operaciones de una lista circular simplemente enlazada

Actividad 38

Instrucciones

1. Codifica las implementaciones de las operaciones de una lista circular simplemente enlazada en un lenguaje de programación.
2. Con la implementación de las operaciones, realiza un programa que permita simular una ruleta de casino a través del siguiente menú.
    a. **Insertar** (tiene que pedir los datos, toda inserción es al final)
    b. **Eliminar** (pedir el dato a eliminar)
    c. **Desplegar** (despliega los datos en forma horizontal)
    d. **Ruleta** (pide el valor a “atinar”, verificar que el valor exista)
        i. En la pantalla se simulará la ruleta “prendiendo y apagando” el dato en el que va corriendo de manera rápida, éste realiza desplegando la lista de manera horizontal de inicio a fin
        ii. Este recorrido de los datos se realizará hasta que el usuario teclee ESC
        iii. La “ruleta” mostrará el dato en que se quedó detenida y si el dato es igual aparecerá un mensaje de “GANASTE” en caso contrario “PERDISTE”
    e. Salir
3. Entregar código fuente y pantallas de captura de cada una de las opciones del menú (excepto Salir).

UNIDAD DE COMPETENCIA III
Aplicar la estructura de datos árbol

3.1 Recursividad Directa
Actividad 39

Instrucciones

1. Con base en los conceptos de recursión, realizar el siguiente programa (en cualquier lenguaje de programación) que calcule *n* números *Fibonacci* con base en la siguiente especificación:

$$Fibonacci(n) = \begin{cases} n, & n = 0 \; o \; n = 1 \\ Fibonacci(n-1) + Fibonacci(n-2), & n > 1 \end{cases}$$

2. Como ejemplo los números *Fibonacci* de **4** son:
   0,1,1,2,3. Esto se realiza sumando a partir del 0 el siguiente 1 que da 1, 1+1 es 2, 1+2 es 3, así sucesivamente.
3. Entregar código fuente y pantallas de captura (por lo menos 4) de la corrida del programa.

3.2 Árboles: usos, características, representación y construcción
Actividad 40: Terminología de un árbol

3.3 Árboles binarios (representación. recorrido en preorden, inorden, postorden)
Actividad 41: Tipología de un árbol

Actividad 42: Recorridos de un árbol

Actividad 43: Árbol de expresión

Actividad 44: Árbol Binario de Búsqueda

Actividad 45: Búsqueda en un Árbol Binario

Actividad 46: Inserción en un Árbol Binario

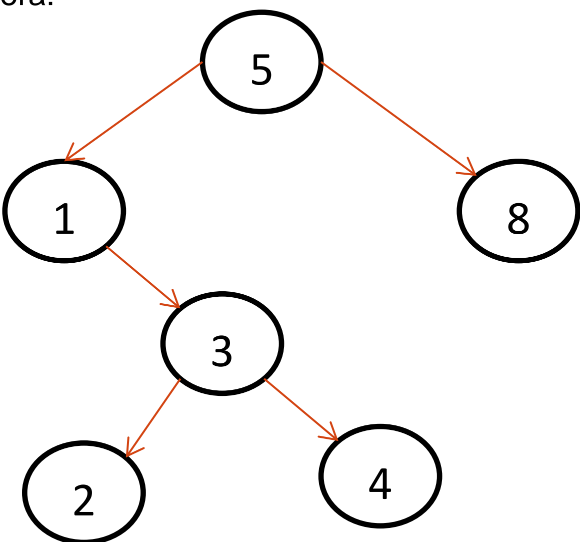
Actividad 47: Eliminación en un Árbol Binario

Actividad 48
Instrucciones

```
Nodo: Registro
        *I: Nodo
        Num: E
        *D: Nodo
FinRegistro

*N: Nodo
```

1. Con base en los algoritmos de implementación de las operaciones de un árbol binario y el registro definido, realizar un programa que permita realizar las siguientes funciones:
    a. Insertar
    b. Buscar
    c. Visualizar
    d. Recorridos
    e. Eliminar
    f. Estadísticas
    g. Salir
2. Descripción de funciones:
    a. Insertar: Sólo pide el número a insertar.
    b. Buscar: Pide el elemento a buscar y sólo dice si está o no en el árbol.
    c. Visualizar: implementar una función que permita desplegar los elementos del árbol de la siguiente manera:
    Si el árbol fuera:



    Lo que desplegaría sería:
        Nivel 1: 5
        Nivel 2: 1, 8
        Nivel 3: 3
        Nivel 4: 2, 4
    d. Recorridos: Desplegar el de cada uno de los recorridos en forma lineal
        i. PreOrden:
        ii. InOrden:
        iii. PostOrden:

   e. Estadísticas: Desplegar los datos correspondientes a la tabla del ejercicio 19 de esta unidad (página 100), cuidar que sean visibles todos los datos ya sea por página o datos.

3. Enviar código fuente y por lo menos 5 pantallas de captura de las funciones de Visualizar, Estadísticas, Recorridos y Buscar.

UNIDAD DE COMPETENCIA IV
Aplicar la estructura de datos grafo

4.1 Grafos: Características y Clasificación
Actividad 49: Características de un Grafo

Actividad 50: Clasificación de un Grafo

4.2 Grafos Dirigidos
Actividad 51: Matriz de adyacencia de un Grafo Dirigido

Actividad 52: Matriz de adyacencia de un Grafo Dirigido con Factor de Peso

Actividad 53: Lista de adyacencia de un Grafo Dirigido

4.3 Grafos No Dirigidos
Actividad 54: Matriz de adyacencia de un Grafo No Dirigido

Actividad 55: Matriz de adyacencia de un Grafo No Dirigido con Factor de Peso

Actividad 56: Lista de adyacencia de un Grafo No Dirigido

4.4 TAD Grafo (Estático Y Dinámico)
Actividad 57-58: Operaciones de un Grafo

Actividad 59

Instrucciones

Con base en las operaciones de un grafo en su representación estática:

1. Elaborar un programa que permita a través de un menú implementar las operaciones de un grafo, cuya información a almacenar son las ciudades de un estado con sus rutas y sus distancias entre éstas.
   a. Crear grafo
      i. Insertar ciudades (vértices)
      ii. Insertar rutas y distancias (arcos)
   b. Eliminar
      i. Eliminar una Ciudad
      ii. Eliminar una ruta
   c. Desplegar
      i. Ciudades
      ii. Rutas
      iii. Estado (Matriz de adyacencia)
   d. Salir
2. Entregar programa fuente. Si es posible graficar el grafo (1 punto extra en parcial)

4.5 Recorridos De Un Grafo

Actividad 60: Recorrido en Anchura de un Grafo

Actividad 61: Recorrido en Profundidad de un Grafo

Actividad 62

Instrucciones

1. Integrar a las opciones del menú las siguientes:
    a. Recorridos (desplegará el recorrido deseado)
        i. Recorrido en Profundidad
        ii. Recorrido en Anchura

4.6 Algoritmos

Actividad 63: Algoritmo de Dijkstra

Actividad 64: Algoritmo de Floyd

Actividad 65: Algoritmo de Warshall

Actividad 66: Algoritmo de Prim

Actividad 67: Algoritmo de Kruskal

Actividad 68

Instrucciones

Con base en la Actividad Nº 62:

1. Integra las siguientes opciones al menú
    a. Camino más corto
        i. Dijkstra
        ii. Floyd
        iii. Warshall
    b. Árbol de expansión de costo mínimo
        i. Prim
        ii. Kruskal
2. Para cada inciso anterior desplegar los resultados finales y si es posible graficar el grafo (1 punto extra en parcial)