Estructura de datos

CESAR AUGUSTO LUNA LOPEZ

Red Tercer Milenio

ESTRUCTURA DE DATOS

ESTRUCTURA DE DATOS

CESAR AUGUSTO LUNA LOPEZ

RED TERCER MILENIO

AVISO LEGAL

Datos para catalogación bibliográfica

César Augusto Luna López

Estructura de datos

ISBN 978-607-733-129-2

Primera edición: 2012

Eduardo Durán Valdivieso

DIRECTORIO

ÍNDICE

Introducción	4
Mapa conceptual	6
Unidad 1. Arreglos	7
Mapa conceptual	8
Introducción	9
1.1. Conceptos	10
1.2. Arreglos unidimensionales	11
1.3. Arreglos bidimensionales	16
1.4. Arreglos de tres o más dimensiones	19
Autoevaluación	23
Unidad 2. Pilas y colas	24
Mapa conceptual	25
Introducción	26
2.1. Definiciones y representaciones	27
2.2. Notaciones infijas, prefijas, postfijas en expresiones	29
2.3. Inserción y remoción de datos en una pila (LIFO)	30
2.4. Inserción y remoción de datos en una cola simple y circular	33
2.5 Problemas	37
Autoevaluación	42
Unidad 3. Algoritmos de ordenamiento y búsqueda	44
Mapa conceptual	45
Introducción	46
3.1. Método de burbuja	47
3.2. Método Shell	49
3.3. Método de quicksort	50
3.4. Búsqueda secuencial	51
3.5. Búsqueda binaria	52
Autoevaluación	55

Unidad 4. Listas 56
Mapa conceptual 57
Introducción 58
4.1. Representación en memoria 59
4.2. Listas enlazadas 59
4.3. Listas doblemente enlazadas 64
4.4. Operaciones con listas 66
4.5. Problemas 69
Autoevaluación 73

Unidad 5. Árboles 74
Mapa conceptual 75
Introducción 76
5.1. Terminología 77
5.2. Árboles binarios y representaciones gráficas 78
5.3. Recorrido de un árbol 81
5.4. Árboles enhebrados 83
5.5. Árboles de búsqueda 85
5.6. Problemas 86
Autoevaluación 107

Unidad 6. Grafos 109
Mapa conceptual 110
Introducción 111
6.1. Terminología 112
6.2. Características generales 113
6.3. Representación de un grafo 114
Autoevaluación 117

Bibliografía 118
Glosario 119

INTRODUCCIÓN

En la actualidad, la eficiencia de un programa informático va de la mano con las técnicas de programación que se emplean en su desarrollo, partiendo desde la elaboración de diagramas de flujo de datos, hasta la escritura de los códigos para el desarrollo del software. Lo anterior busca el acceso a los datos de la información de una manera ordenada mediante instrucciones válidas, empleando una secuencia lógica.

La estructura de datos se refiere a un conjunto de técnicas que aumentan considerablemente la productividad del programa, reduciendo en elevado grado, el tiempo requerido para escribir, verificar, depurar y mantener los programas. El término estructura de datos hace referencia a un conjunto de datos que, por medio de un nombre, identifican un espacio en memoria, teniendo ciertas características como la organización y estructuración, permitiendo realizar operaciones definidas en ellas. Las estructuras de datos pueden ser de dos tipos:

- Estructuras de datos estáticas (las que tienen un tamaño definido).
- Estructuras de datos dinámicas (en las cuales su tamaño puede ser cambiado en tiempo de ejecución).

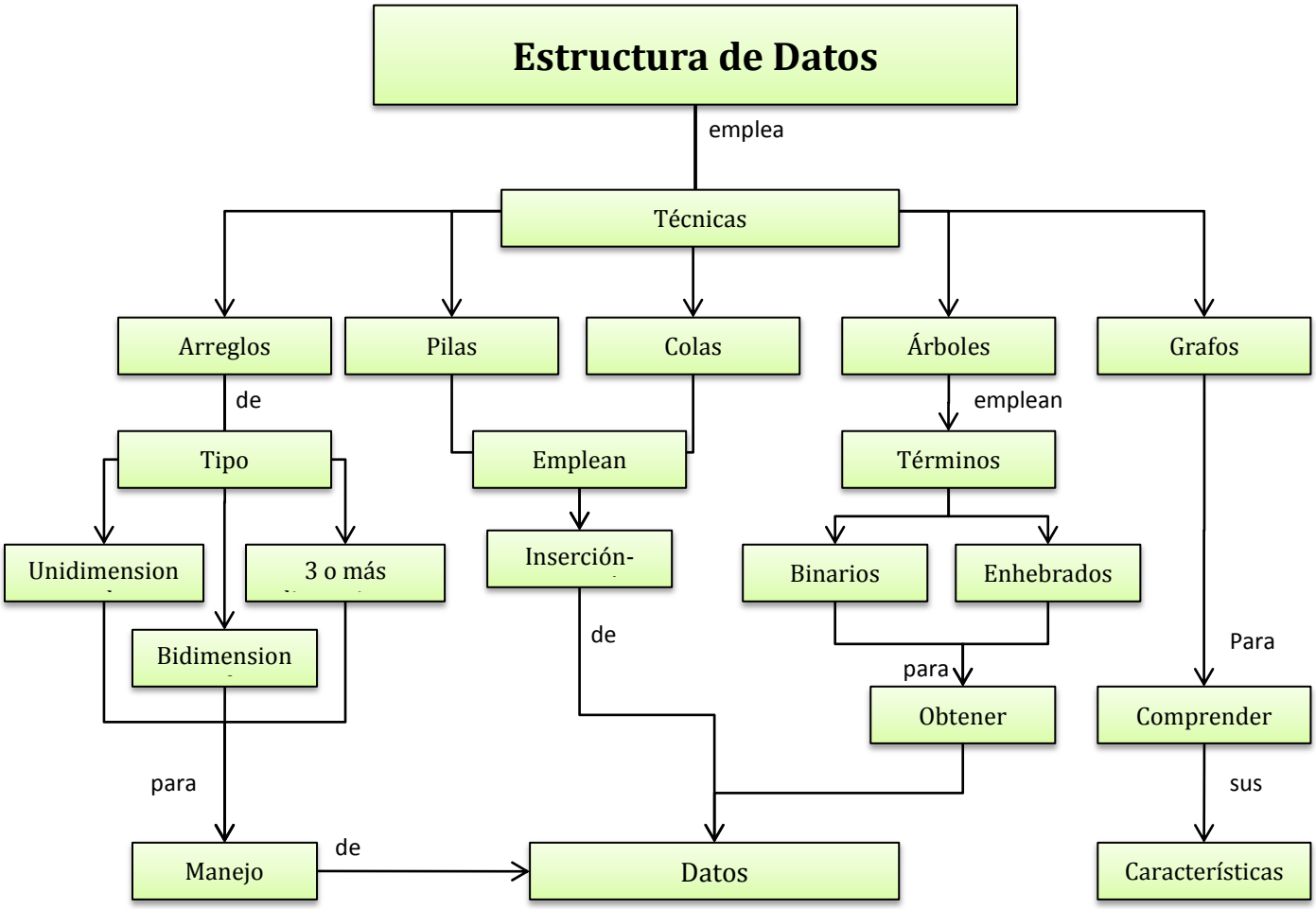
La presente obra comienza su estudio, con las estructuras de datos estáticas, analizando el concepto y fundamento de los arreglos, sus métodos para el manejo de datos, sus variantes que pueden dar origen a estructuras de arreglos de una o más dimensiones.

En la unidad dos analizaremos el concepto de las pilas y colas, los métodos en que se manipulan los datos para insertarlos o eliminarlos, así como sus reglas de uso.

En la unidad tres combinaremos lo aprendidos en las anteriores unidades para lograr realizar prácticas más complejas, donde involucren conceptos y métodos que permitan ordenar datos, empleando arreglos, o bien para realizar búsquedas de información a través de algoritmos computacionales. Las últimas unidades abordarán los temas de árboles y

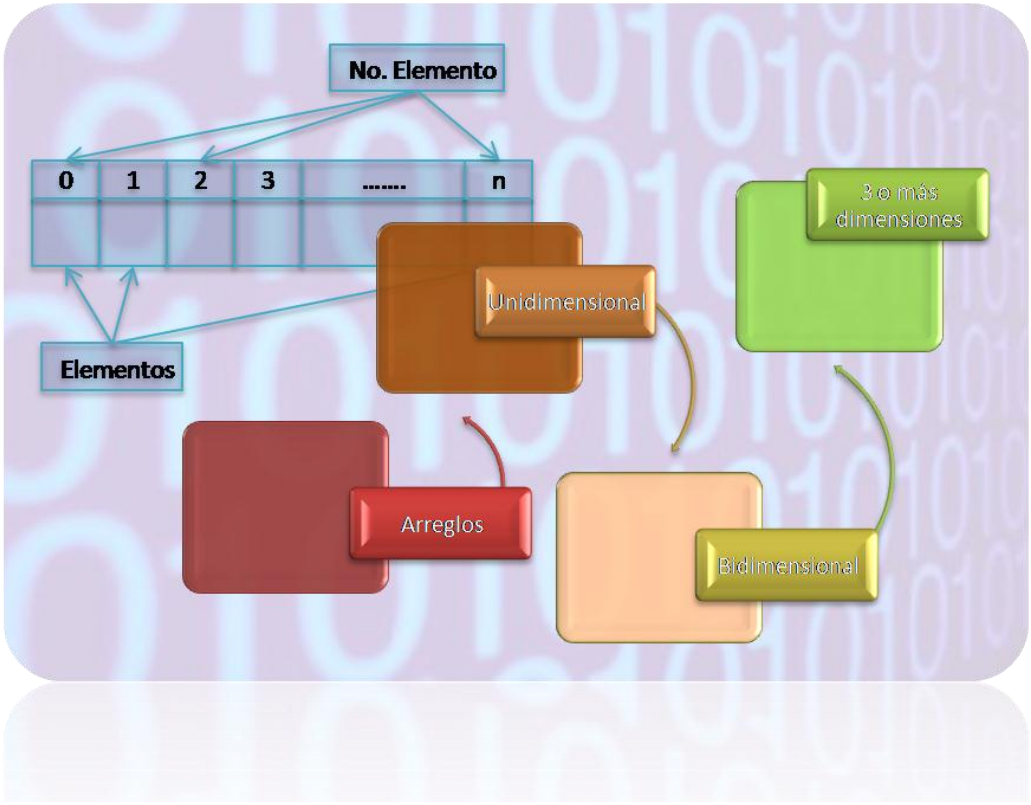
grafos, que son estructuras dinámicas que nos permitirán trabajar con almacenamientos secundarios. Asimismo, se introducirá sobre el uso de los grafos, su empleo y creación dentro de un lenguaje de programación.

MAPA CONCEPTUAL



UNIDAD 1

ARREGLOS



OBJETIVO

Analizar el uso e implementación de los arreglos para la resolución de problemas de datos estructurados, mejorando el tiempo de desarrollo y eficacia de los sistemas.

TEMARIO

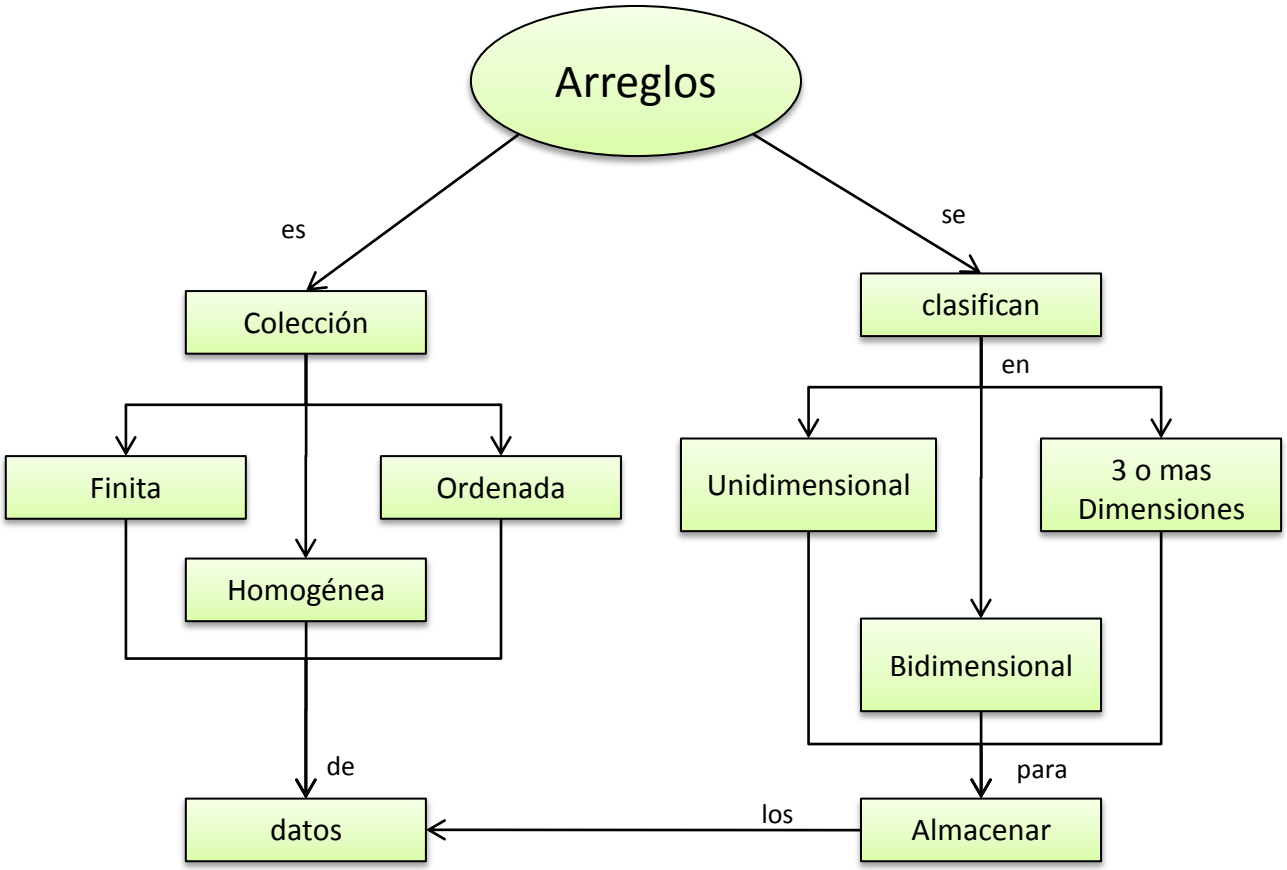
1.1. CONCEPTOS

1.2. ARREGLOS UNIDIMENSIONALES

1.3. ARREGLOS BIDIMENSIONALES

1.4. ARREGLOS DE TRES O MÁS DIMENSIONES

MAPA CONCEPTUAL



INTRODUCCIÓN

En esta unidad conoceremos uno de los tipos de datos estructurados más empleados dentro de la programación, los arreglos. La utilidad de estos es trascendental al momento de codificar los programas, debido a que dentro de un programa tendemos a emplear una gran cantidad de variables, esto puede ser un poco ortodoxo debido a que si necesitamos 10, 20 o 100 variables para un mismo tipo de datos, la manipulación de estos se vuelve más complicada.

¿Entonces como resolvemos el manejar tantas variables? Aquí es donde se emplean los arreglos, para formar una estructura más definida, más fácil de procesar e implementar en nuestros códigos.

Partiremos de la conceptualización de los arreglos, como dependiendo de la necesidad pueden emplearse de una, dos o más dimensiones. Como escribir y leer sus datos, así como ejemplos de la codificación en el lenguaje Java.

1.1. CONCEPTOS

Un arreglo es una estructura de datos lineal, “un arreglo puede definirse como un grupo o una colección finita, homogénea y ordenada de elementos.”[1] Donde finita significa que debe tener un límite de elementos, homogénea que los elementos deben de ser del mismo tipo de datos, y ordenada porque se puede conocer cuál es el primer elemento, los subsiguientes o el último.

En la siguiente gráfica puede observarse cómo se representa un arreglo y se identifican las partes que lo integran.

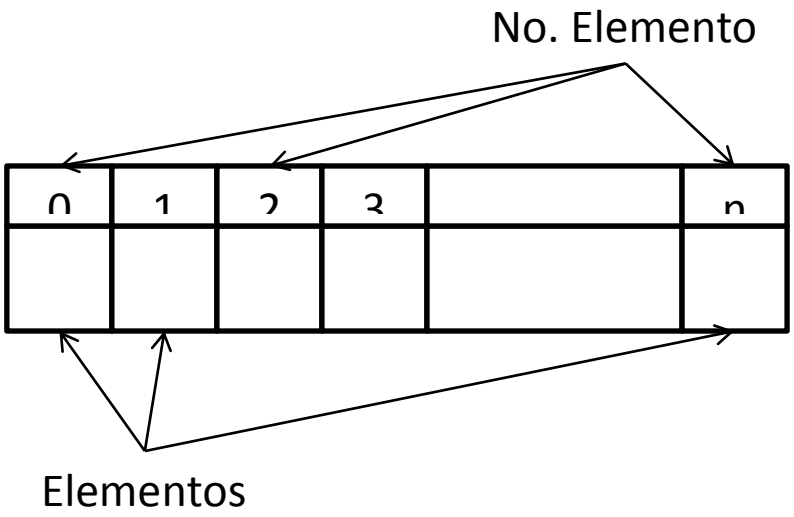


Fig.1.1. Representación de un arreglo

Dentro del uso de los arreglos, se puede hacer referencia a un número de elemento para introducir, actualizar o extraer valores, ese Número de elemento se conoce como índice y especifica la posición del elemento en el arreglo.

Para referirse a una posición del arreglo, es necesario conocer el *nombre* del arreglo y su *índice*, por ejemplo, tomando la figura 1.2., tenemos un arreglo llamado “Datos” y necesitamos extraer el valor “77”, es necesario especificar: Datos [3]

[1] Osvaldo Cairó /Silvia Guardati , Estructuras de datos, pág. 4

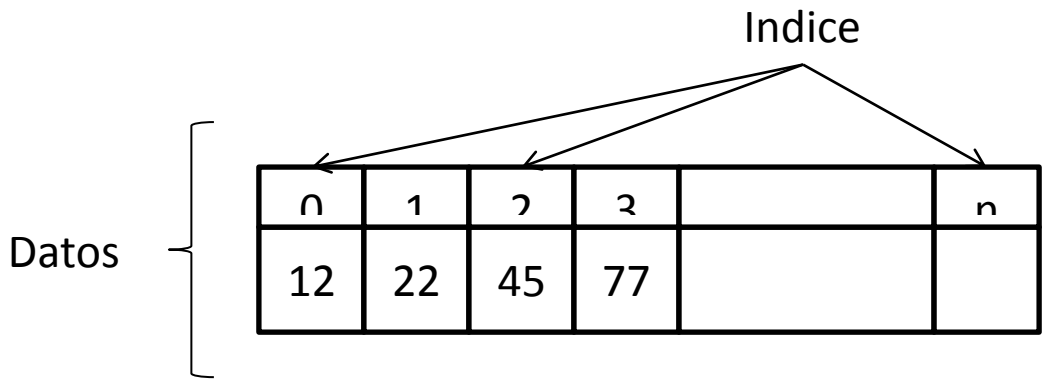


Fig. 1.2. Arreglo Datos

Según su estructura los arreglos se clasifican en:

- Arreglos unidimensionales o de una dimensión.
- Arreglos bidimensionales o de dos dimensiones.
- Arreglos tridimensionales o de tres o más dimensiones.

1.2. ARREGLOS UNIDIMENSIONALES

El concepto de arreglo empleado en el tema anterior, se aplica también para referirse a los arreglos de dimensión, es decir, es un tipo de datos estructurado compuesto de un número de elementos finitos, homogéneos y ordenados.

Los elementos del arreglo se almacenan en posiciones contiguas de memoria, a cada una de las cuales se puede acceder directamente mediante su índice. La forma de acceder a un arreglo de una dimensión es directa, pues se puede especificar el índice del elemento sin necesidad de conocer el contenido de los elementos contiguos.

Esto se puede ilustrar mejor con el siguiente ejemplo: supóngase que desea capturar las edades de un grupo de 100 personas, para conocer cuántas están sobre el promedio de edad.

Este problema se podría resolver empleando 100 variables simples para almacenar las edades; luego, se calcula el promedio, y por último se comparan las 100 variables para determinar cuántas personas están sobre el promedio. Este método consumirá muchos recursos del sistema (principalmente por la creación de tantas variables), y representa la complejidad de utilizar en la programación sólo datos simples.

Definición de un arreglo. La forma correcta de resolver el anterior problema, sería empleando un arreglo. ¿Pero cómo se definen?, esto depende principalmente del lenguaje de programación que utilicemos, para cuestiones de estudio, la mayoría de los autores sobre este tema emplean una sintaxis generalizada de la siguiente forma:

Nombre_Arreglo = ARREGLO [Limite_inferior . . Limite_Superior] de TIPO

Donde:

Arreglo. Es la palabra reservada para crear los arreglos

Nombre_Arreglo. Representa el Nombre que emplearemos para referirnos al arreglo

Limite_Inferior y Limite_Superior. Representan el número Índice para indicar el inicio y fin del arreglo, es decir, un rango, por ejemplo del [1 .. 3] o [5 .. 20].

Tipo. Es el tipo de datos que almacenará ese arreglo, hay que recordar que los arreglos son homogéneos y pueden ser cualquier tipo ordinal (cadena, entero, booleano, real, etc.)

Como ya se ha mencionado, manejaremos la sintaxis empleado en el lenguaje JAVA, donde los arreglos se definen de la siguiente forma:

Tipo Nombre_Arreglo [ ] = new Tipo [Tamaño_Arreglo];

Note como en este lenguaje sólo se emplea Tamaño_Arreglo, para definir el tamaño y no el rango, esto es debido a que JAVA emplea como primer índice el 0. Así que por deducción indicando el Tamaño_arreglo, el lenguaje emplea el rango 0 al Tamaño_Arreglo-1.

Retomando el ejemplo de las edades, para almacenar los valores basta con definir un arreglo de la siguiente forma:

int Edad [ ] = new Edad [100];

La representación gráfica del anterior arreglo, se presenta en la siguiente ilustración:

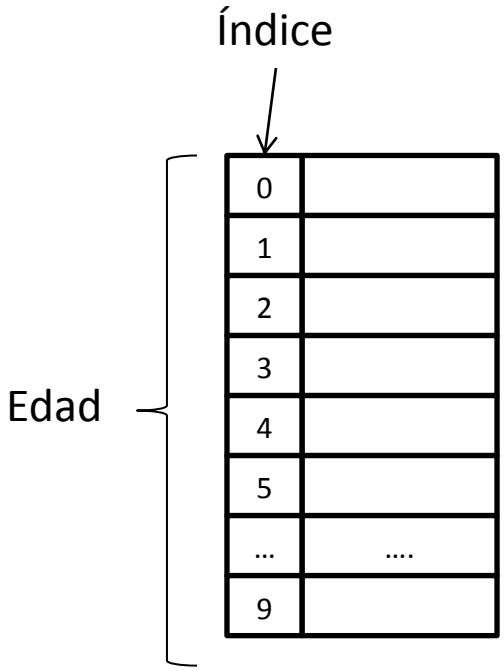


Fig. 1.3. Arreglo de 100 posiciones

En el proceso de escritura en un arreglo, se debe escribir un valor para cada uno de sus componentes de la siguiente manera:

Nombre_ Arreglo [Indice] = Valor

Empleando el lenguaje JAVA quedaría:

Edad [0] = 33;

Almacenando en el arreglo Edad en la posición 0 el valor de 33. Para asignar los valores es conveniente que se integre un ciclo para que la lectura sea más automatizada.

```
Repetir con I desde 1 hasta N
Leer E
Escribir Arreglo [ I ] = E
```

Para actualizar el contenido de un componente del arreglo, sólo basta con reasignarle el valor, como una variable simple, el anterior valor se elimina para dar paso al nuevo valor.

Otra forma de crear arreglos, es empleando la asignación directa de la forma:

Tipo Nombre_Arreglo [ ] = {Valor1, Valor2, Valor3…Valor_n}

Ejemplos:

String Cadenas [ ] ={“Juan”, ”Pedro”, “Tomas”, “Raul”};

Esto creará automáticamente un arreglo con la cantidad de registros especificados, es decir, creará un arreglo llamado Cadenas con 4 componentes, sustituyendo do instrucciones String Cadenas[ ] = new String [4] y la asignación de los datos.

Dentro de las estructuras de los arreglos unidimensionales, podemos llevar a cabo las siguientes operaciones:

- Lectura.
- Escritura.
- Actualización.
- Ordenación.
- Búsqueda.

Lectura. Esta operación se refiere al hecho de leer los datos de un registro contenido en un arreglo, permitiendo asignar el valor a una variable.

X ← Arreglo [i] donde i representa la posición del arreglo

Escritura. Esta operación se refiere al hecho de asignarle valor al registro contenido en un arreglo.

Arreglo [i] ← valor

Actualización. En este proceso, las operaciones consideradas como actualización, son los procesos de eliminar, insertar y modificar datos, tomando en cuenta si los datos están ordenados o no.

Ordenación. Este proceso consiste en reordenar los datos del arreglo tomando un criterio, por ejemplo, ordenar los datos numéricos de mayor a

menor, ordenar las cadenas alfabéticamente, etc. Los métodos de ordenamiento serán abordados con detalle más adelante.

Continuando con el ejemplo de las edades, se presenta el código para la solución en pseudocódigo y Java:

```
Inicio
        Definir  Max← 100
        definir arreglo “Arreglo_Edad”
        desde i=0 hasta i < Max
                leer Edad;
                Arreglo_edad[ i ] ←edad
                Suma ←Suma + Arreglo_edad [ i ]
        Fin_desde
        Promedio←Suma/Max
        Des i=0 hasta i < Max
                Tempo← Arreglo_Edad[i]
                Si Tempo > Promedio entonces
                        Arriba_Prom=Arriba_Prom + 1
                Fin_Si
        Fin_desde
        Imprimir Arriba_Prom
Fin_Inicio
```

```
1    import javax.swing.JOptionPane;
2
3   public class ArregloUni {
4       public static void main(String[] args) {
5           int Max=100, Suma=0, Promedio=0,Tempo=0, Arriba_Prom=0;
6          String Edad;
7           int Arreglo_Edad[]=new int[Max];              // Declaraccion del arreglo
8                for (int i=0; i<Max;i++){
9                Edad= JOptionPane.showInputDialog("Edad:");
10              Arreglo_Edad [ i ] = Integer.parseInt (Edad);
11                   Suma += Arreglo_Edad [ i ];
12               }
```

```
13              Promedio=Suma/Max;
14
15          for (int i=0; i<Max;i++){
16                  Tempo=Arreglo_Edad[i];
17                  if (Tempo>Promedio){
18                      Arriba_Prom++;
19                  }
20          }
21
22      JOptionPane.showMessageDialog(null, "El número de edades arriba
        del promedio es "+Arriba_Prom);
23
24    }
25  }
```

Donde se puede observar la declaración del arreglo en la línea 7

int Arreglo_Edad [ ] =new int [Max];

Además, la asignación de los valores en la línea10, donde se convierte la cadena a entero para asignarla al arreglo.

Arreglo_Edad [i] = Integer.parseInt (Edad);

Y por último, la lectura de los valores del arreglo en la línea 16

Tempo=Arreglo_Edad[i];

ACTIVIDAD DE APRENDIZAJE

1.- Desarrolla un programa que permita la captura de las ventas de un Comisionista de forma mensual, mostrando al final la venta Promedio, la venta más alta y la más baja.

1.3. ARREGLOS BIDIMENSIONALES

“Un arreglo bidimensional es un conjunto de datos homogéneo, finito y ordenado, donde se hace referencia a cada elemento por medio de dos

índices. El primero se utiliza generalmente para indicar renglón, y el segundo para indicar columnas. También puede definirse como un arreglo de arreglos."[2]

Un arreglo bidimensional (también conocido como tabla o matriz), es un arreglo con dos índices, al igual que los vectores que deben ser ordinales.

Es común destacar que el empleo de los arreglos de dos dimensiones se emplea para la construcción de tablas, en la cual están representadas por filas y columnas. Como se mencionó, para localizar o almacenar un valor en el arreglo se deben especificar dos posiciones (dos subíndices), uno para la fila y otro para la columna.

Arreglo [Fila, Columna]

La definición de un arreglo bidimensional quedaría con la siguiente sintaxis:

Tipo Nombre_Arreglo [ ] [ ] = new Tipo [Tam_Arreglo_Fila]
[Tam_Arreglo_Columna];

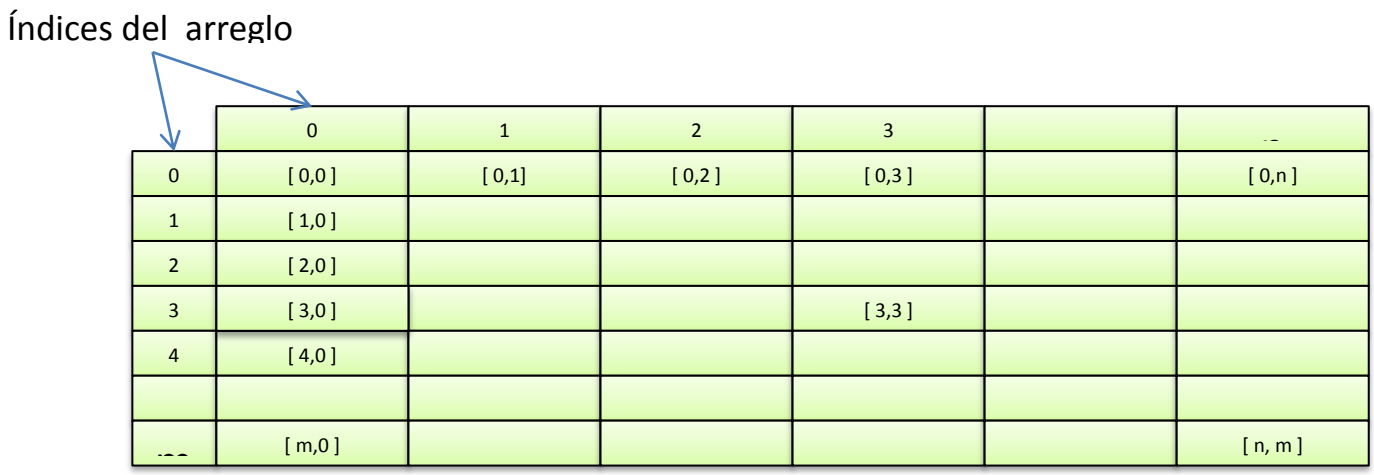


Fig. 1.4. Representación gráfica de un arreglo bidimensional

Donde Tam_Arreglo_Fila se define el número de renglones que tendrá el arreglo y con Tam_Arreglo_Columna se declara la cantidad de columnas que contendrá el arreglo

Las operaciones empleadas en un arreglo como la lectura, escritura, actualización, ordenamiento y búsqueda, seguirán el mismo método de los arreglos unidimensionales, con la diferencia de los dos índices.

[2] Osvaldo Cairó /Silvia Guardati, *Estructuras de datos*, p. 19.

Por ejemplo, necesitamos una matriz donde se almacenen las calificaciones de 30 personas, conteniendo cinco materias y su matrícula. La estructura del arreglo quedaría de la siguiente forma:

```
int Registros [ ] [ ] = new int [ 30 ][ 6 ];
```

Otra variante sería la siguiente:

```
String Datos[]=
{“Matricula”,”;Materia1”,”Materia2”,”Materia3”,”Materia4”,”Materia5”};
int Registros [ ] [ ] = new int [ 30 ][ Datos.length ];
```

El ejemplo completo sería el siguiente:

```
1   import javax.swing.JOptionPane;
2
3   public class Bidi {
4
5       public static void main(String[] args) {
6           int NoAlumno=2;
7           String DatoSimple;
8               String Datos[ ] = { "Matricula","Materia 1","Materia 2","Materia
            3",
                "Materia 4","Materia 5" };
9           int Registros [ ] [ ] = new int [ 30 ][ Datos.length ];
10
11          int Arreglo_Edad[]=new int[NoAlumno];
12              for (int i=0; i<NoAlumno;i++){
13                  for (int j=0; j<Datos.length; j++) {
14                      DatoSimple=JOptionPane.showInputDialog("Ingresa la "
                            + Datos[ j ] + " del alumno"+ (i+1));
```

```
15                          Registros[i][j]=Integer.parseInt(DatoSimple);
16                      }
17                  }
18
19              JOptionPane.showMessageDialog(null, "Datos capturados
                    completamente...");
20
21          }
22
23      }
```

Note cómo las asignaciones, y de un arreglo, son las mismas empleadas en un arreglo de una dimensión, lo que se debe tener presente es que ahora se maneja un arreglo con dos dimensiones, igual a una tabla, y para hacer referencia a una posición, tenemos que valernos de dos índices (Fila, columna).

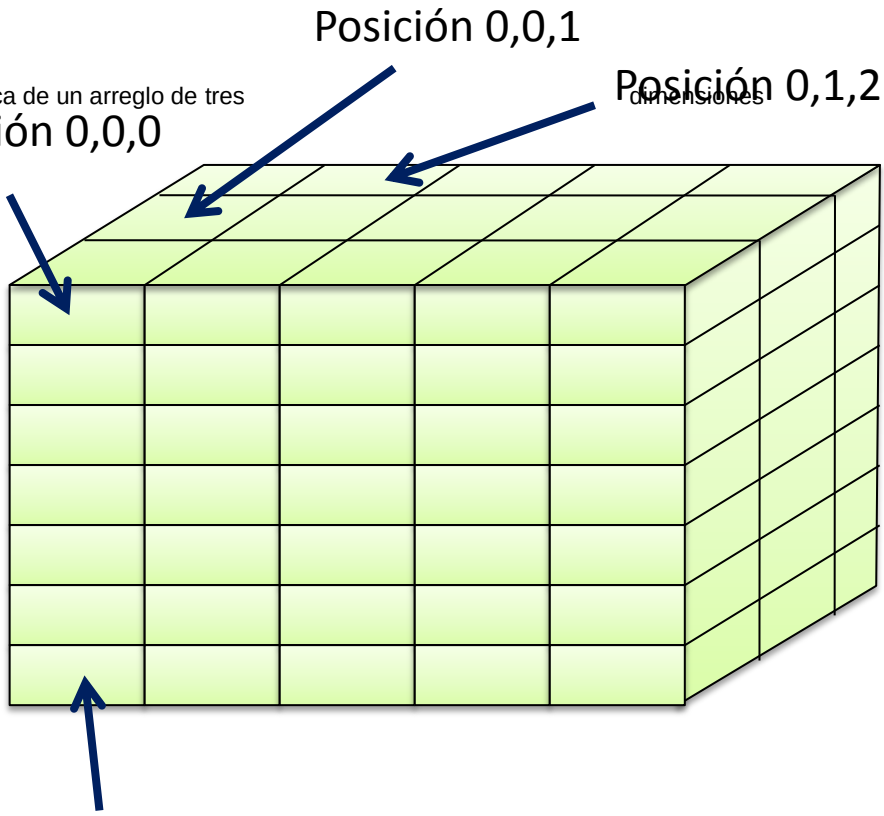
ACTIVIDAD DE APRENDIZAJE

1. Desarrolla un programa que permita la captura de las ventas de 30 Comisionistas de forma mensual, mostrando al final la venta Promedio, la venta más alta y más baja.

1.4. ARREGLOS DE TRES O MÁS DIMENSIONES

Esta estructura de datos está compuesta por *n* dimensiones, especificada en índices, y para hacer referencia a cada componente del arreglo es necesario utilizar *n* índices por cada dimensión empleada.

Fig. 1.5. Representación grafica de un arreglo de tres dimensiones



Para conocer el número total de elementos en un tipo de arreglos multidimensionales se emplea la siguiente fórmula:

No. TOTAL DE ELEMENTOS = D1 * D2* D3 * ...* Dn

donde:

D1, D2, D3…Dn= No. total de dimensiones

El anterior gráfico representa un arreglo de tres dimensiones con un total de 105 elementos, esto es porque tiene siete filas, cinco columnas y tres profundidades.

La sintaxis para la creación de un arreglo de tres dimensiones quedaría de la siguiente forma:

```
Tipo Nombre_Arreglo [ ] [ ] [ ]= new  Tipo [Fila] [ Columna ] [ Profundidad ];
```

Para comprender más claro los arreglos de tres dimensiones o más, realizaremos el siguiente problema:

En una empresa desean registrar las ventas de dos empleados que venden cinco productos, además de controlarlo por las tres semanas que estuvieron trabajando. La solución sería manejar un arreglo tridimensional de la siguiente forma:

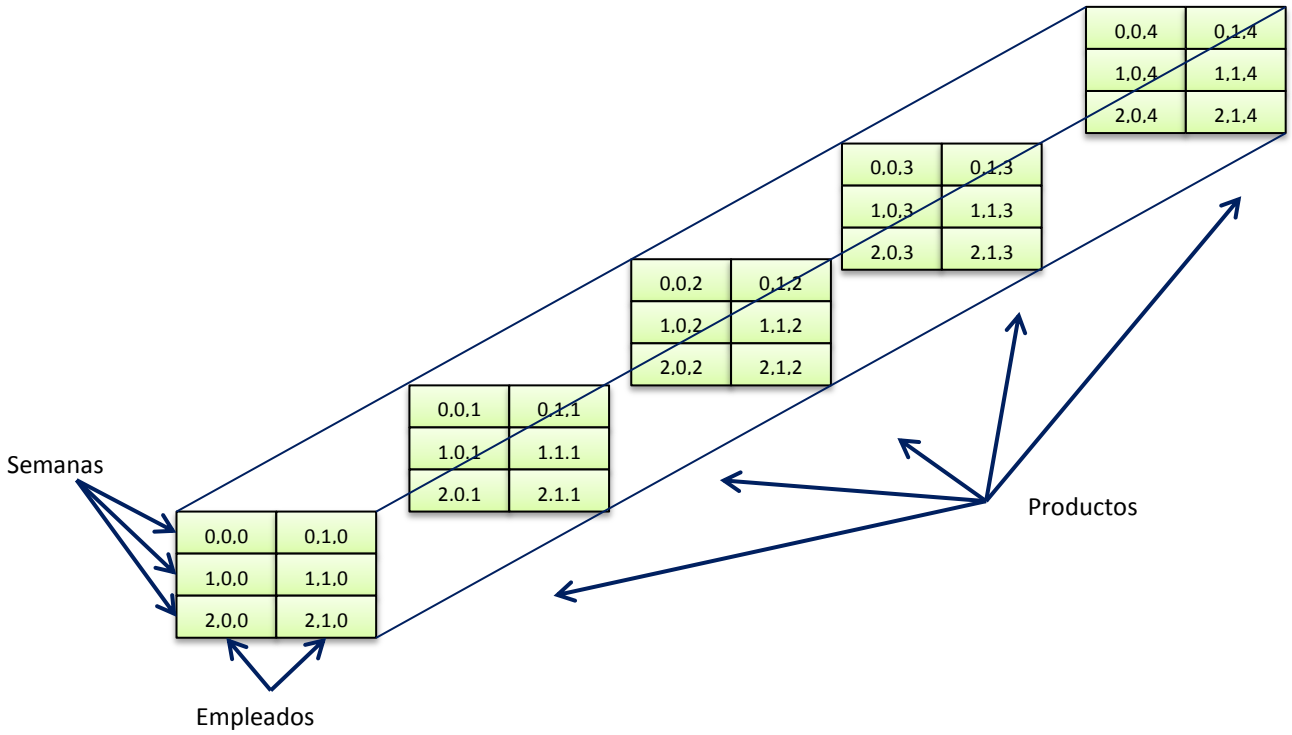


Fig. 1.6.- Arreglo de 3 filas (Semanas), 2 columnas (Empleados) y 5 profundidades (Productos)

El código quedaría expresado de la forma:

```
1       import javax.swing.JOptionPane;
2
3       public class tridimensional {
4               public static void main(String[] args) {
5                   String Valor;
6                   int Arreglo_Datos[][][]=new int[3][2][5];
7
8                   for (int i=0;i<3; i++){
9                      for (int j=0; j<2; j++){
10                        for (int k=0; k<5; k++){
11                              Valor=JOptionPane.showInputDialog("Semana:"+
(i+1)+" del
                                    vendedor: "+(j+1)+" del producto: "+(k+1));
12                              Arreglo_Datos[i][j][k]=Integer.parseInt(Valor);
13                        }
14                     }
```

```
15              }
16          }
17      }
```

ACTIVIDAD DE APRENDIZAJE

1.- Desarrolla un programa que permita la captura de las ventas de 20 Comisionista de forma mensual y de cinco productos diferentes, mostrando al final la venta Promedio, la venta más alta y más baja de cada vendedor.

AUTOEVALUACIÓN

1.- Puede definirse como un grupo o una colección finita, homogénea y ordenada de elementos. ( ) int x [ ] [ ]= int [3][4];

2.- En empleado para definir la posición de un dato en el arreglo ( ) int x [ ] = int [5];

3.- A los arreglos de dos dimensiones se le llaman ( ) Ordenado

4.- A los arreglos de una dimensión se le llaman ( ) Arreglo

5.- Ejemplo de un arreglo bidimensional ( ) Homogéneo

6.- Ejemplo de un arreglo unidimensional ( ) Finita

7.- Ejemplo de un arreglo tridimensional ( ) Índice

8.- Define que un arreglo tiene un límite de elementos ( ) int x [ ] [ ] [ ] = int [5][2][3];

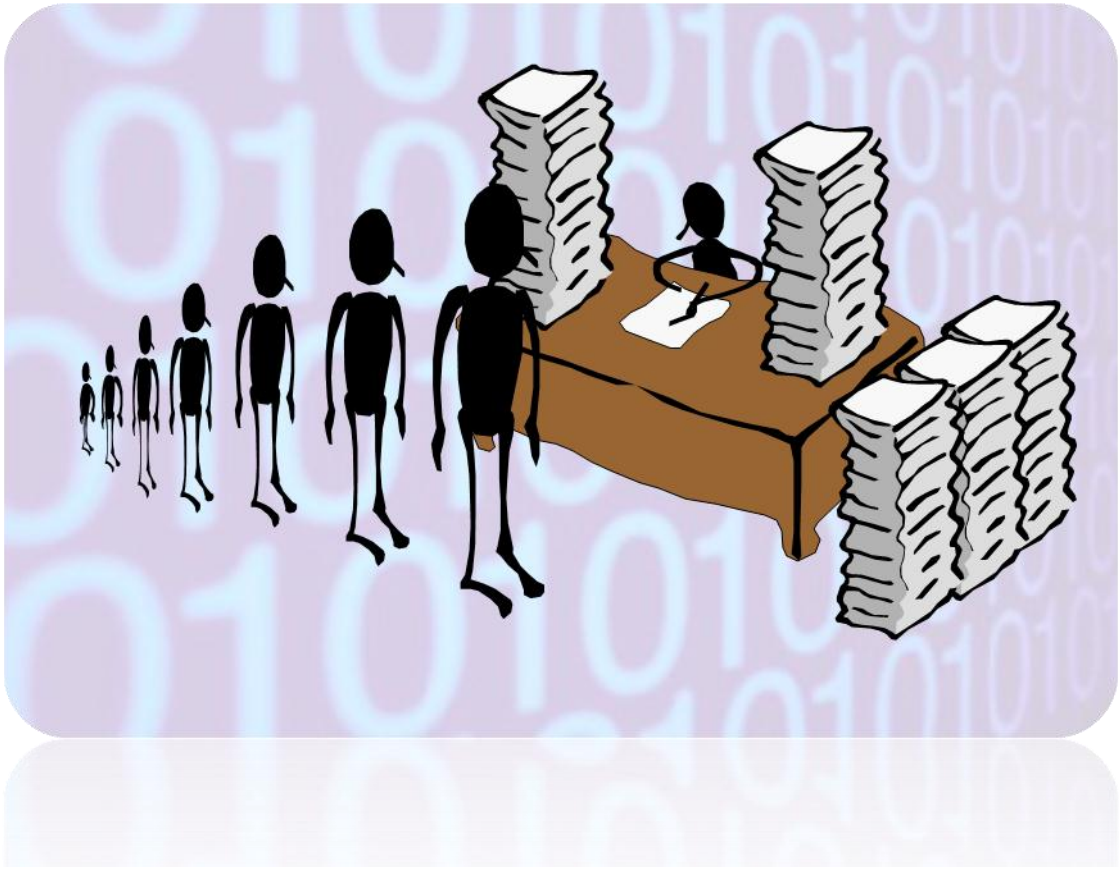
9.- Significa que un arreglo debe ser del mismo tipo ( ) Bidimensionales

10.- Define que cada elemento tiene su posición ( ) Unidimensionales

Respuesta: 6, 5, 10, 1, 9, 8, 2, 7, 3, 4.

UNIDAD 2

PILAS Y COLAS



OBJETIVO

Comprender los conceptos de *Pilas* y *Colas*, analizando las formas de manejar el acceso a datos, partiendo de sus fundamentos básicos.

TEMARIO

2.1. DEFINICIONES Y REPRESENTACIONES

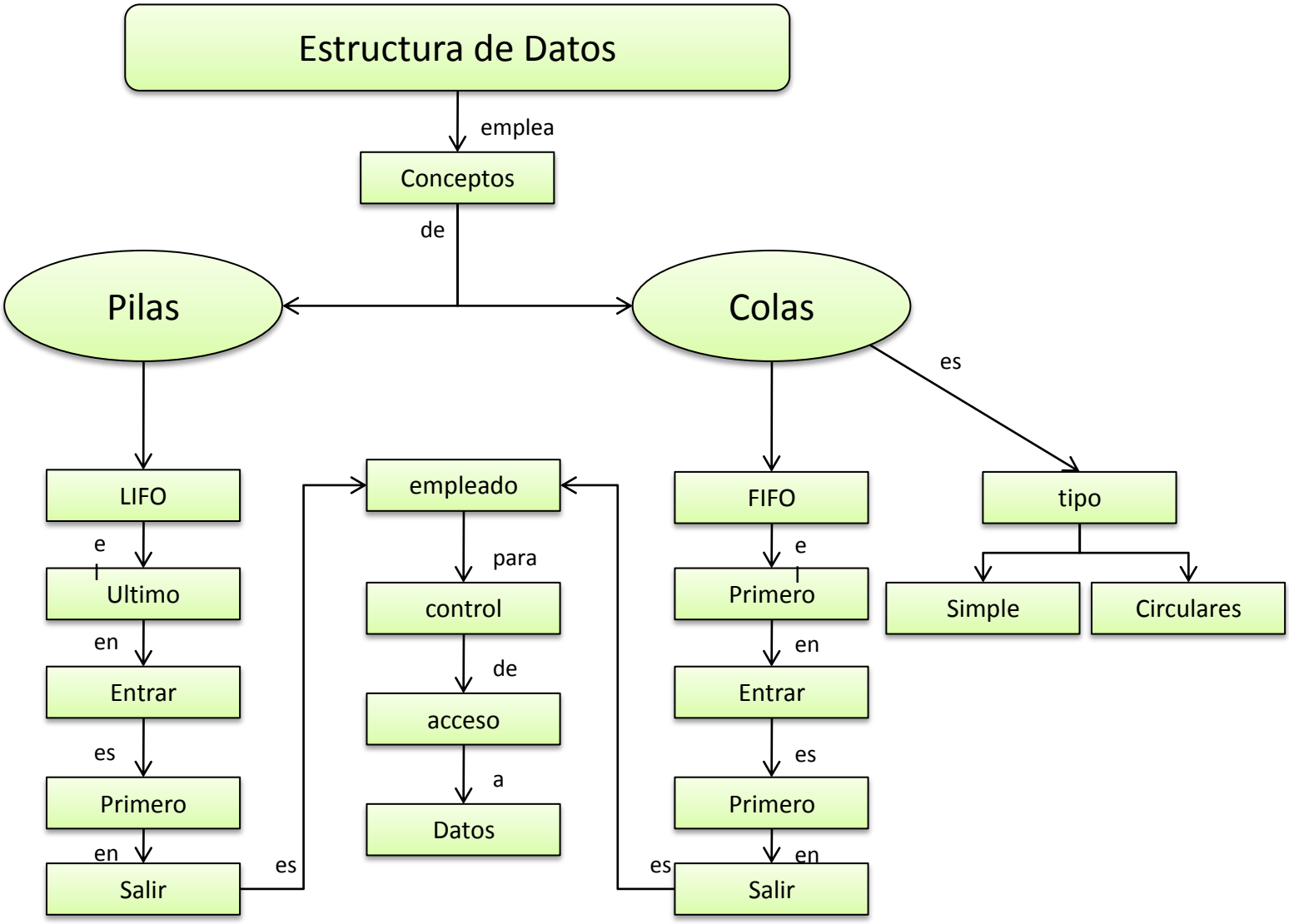
2.2. NOTACIONES INFIJAS, PREFIJAS, POSTFIJAS EN EXPRESIONES

2.3. INSERCIÓN Y REMOCIÓN DE DATOS EN UNA PILA (LIFO)

2.4. INSERCIÓN Y REMOCIÓN DE DATOS EN UNA COLA SIMPLE Y CIRCULAR

2.5. PROBLEMAS

MAPA CONCEPTUAL



INTRODUCCIÓN

Una de las características de los arreglos vistos en la unidad anterior, es que los datos pueden ser insertado o actualizados en cualquier posición y en cualquier momento. Sin embargo, dependiendo de las necesidades, en ocasiones necesitamos que cumplan con ciertas restricciones para controlar el acceso a los datos o bien para realizar determinados procesos en los cuales será necesario tener un orden.

En esta Unidad se comienza a explorar otro concepto fundamental en la *estructura de datos*, las Pilas y las Colas, a continuación se abordarán los conceptos, la implementación en los lenguajes de programación y su utilidad.

El manejo de las Pilas y las Colas incluye muchas restricciones a la forma de acceder a los datos, cada una de modo diferente, las cuales también se detallarán en el transcurso de la presente Unidad.

2.1. DEFINICIONES Y REPRESENTACIONES

Uno de los conceptos que más se emplean en las estructuras de datos lineales, en la elaboración de programas, son las *pilas*. Éstas son aplicadas en cuanto a las restricciones sobre el acceso a los datos del arreglo, ya sea para insertar o eliminar elementos, actualizando el contenido de los registros.

Iniciemos por definir el término fundamental de una pila como "una lista de elementos en la cual se puede insertar y eliminar elementos sólo por uno de los dos extremos."[3]

Para tener una comprensión más clara de lo que es una pila, imagine el acomodo de latas de un producto X en un centro comercial. O bien el apilamiento de libros en una biblioteca.

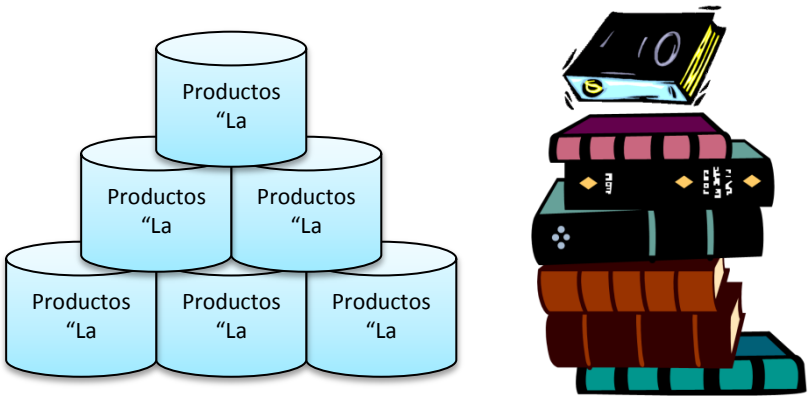


Fig. 2.1. Productos en un supermercado y libros apilados en la biblioteca

Según las imágenes anteriores, podemos deducir el razonamiento de cómo extraer un elemento de la pila, los componentes de una pila serán empleados en orden inverso al que se colocaron. Es decir, el último en entrar debe ser el primero en salir.

Las pilas son estructuras utilizadas muy a menudo como herramientas de programación de tipo LIFO (Last in-First out), ya que permiten el acceso solo a un elemento a la vez: el último elemento insertado. La mayoría de los procesadores utilizan una arquitectura basada en pilas.

La pila se considera un grupo ordenado de elementos, teniendo en cuenta que el orden de los mismos depende del tiempo que lleven "dentro" de la estructura. Las pilas son empleadas en el desarrollo de sistemas informáticos y software en general. Por ejemplo, el sistema de soporte en

[3] Osvaldo Cairó/Silvia Guardati, *Estructuras de datos*, p. 75. *Cfr*. http://www.cesarportela.com.ar/facultad/08-grafos2005.pdf

tiempo de compilación y ejecución de Pascal, utiliza una pila para llevar la cuenta de los parámetros de procedimientos y funciones, variables locales, globales y dinámicas. Este tipo de estructuras también son utilizadas para traducir expresiones aritméticas o cuando se quiere recordar una secuencia de acciones u objetos en el orden inverso del ocurrido.

Ahora conoceremos el concepto de Cola y se definirá como "una estructura de almacenamiento donde los datos van a ser insertados por un extremo y serán extraídos por otro".[4] El concepto anterior se puede simplificar como FIFO (first-in, first-out), esto es, el primer elemento en entrar debe ser el primero en salir.

Los ejemplos sobre este mecanismo de acceso, son visibles en las filas de un banco, los clientes llegan (entran) se colocan en la fila y esperan su turno para salir.

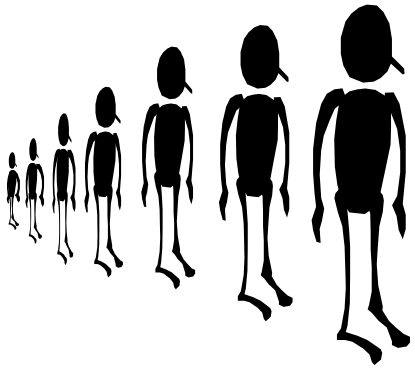


Fig. 2.2.- Ejemplo grafico de las colas en un banco.

Varios hechos de la vida que podemos aplicar en el concepto de pilas son, por ejemplo, cuando las personas hacen fila para esperar el uso de un teléfono público, para subir al transporte escolar o un autobús, para realizar los pagos en un supermercado, etc.

ACTIVIDAD DE APRENDIZAJE

1.-Realizar una investigación y entregar un resumen donde se comprenda el concepto pilas y colas.

[4] Goodrich/Tamassia, *Estructura de datos y algoritmos en Java*, p. 215.

2.2. NOTACIONES INFIJAS, PREFIJAS, POSTFIJAS EN EXPRESIONES

La implementación de las pilas en una aplicación para la resolución de problemas, es común; esto se debe a que dentro de las estructuras de datos, el uso de las pilas permite mejorar la forma de analizar y resolver problemas matemáticos.

Para tomar problemas reales y aclarar el uso de las pilas, se empleará el concepto de las notaciones, las cuales se pueden clasificar en:

Notaciones infijas: Son llamadas así a la anotación de fórmulas matemáticas que emplean operadores, por ejemplo 2+8, es decir, es la forma de escribir nuestras operaciones de uso común.

Notaciones prefijas: Estas permiten expresar nuestras operaciones matemáticas, colocando los operadores al inicio de la expresión, continuando con el ejemplo anterior quedaría + 28.

Notación posfija: Permite escribir las operaciones colocando los operadores al final de la expresión, por ejemplo: 28+.

Véase la siguiente expresión algorítmica para la conversión de una expresión infija en postfija (RPN - Reverse Polish Notation).

1. Aumentar la pila
2. Inicializar el conjunto de operaciones
3. Mientras no exista error y no es fin de la expresión infija realizar:
    - Si el carácter es igual a
        1. PARENTESIS IZQUIERDO. Colocarlo en la pila
        2. PARENTESIS DERECHO. Extraer y desplegar los valores hasta encontrar paréntesis izquierdo. Pero NO desplegarlo.
        3. UN OPERADOR.
            - Si la pila está vacía o el carácter tiene más alta prioridad que el elemento del tope de la pila insertar el carácter en la pila.
            - En caso contrario extraer y desplegar el elemento del tope de la pila y repetir la comparación con el nuevo tope.
        4. OPERANDO. Desplegarlo.
4. Extraer y mostrar los elementos de la pila hasta que se agote.

Ejemplo de conversión de expresiones fijas a postfija

Expresión en formato fija	Expresión en formato RPN
8+2	2 8 +
7	7
5+2/6	2 6 / 5 +
8+ 5/6 - 2/x	85 6 / + 2 x / -

Algoritmo para evaluar una expresión RPN

1. Incrementar la pila
2. Repetir
    - Tomar un caracter.
    - Si el caracter es un operando colocarlo en la pila.
    - Si el caracter es un operador entonces tomar los dos valores del tope de la pila, aplicar el operador y colocar el resultado en el nuevo tope de la pila. (Se produce un error en caso de no tener los 2 valores)
3. Hasta el fin de la expresión RPN.

ACTIVIDAD DE APRENDIZAJE

1.- Realizar una investigación y entregar un resumen, donde se comprenda el concepto de las notaciones.

2.3. INSERCIÓN Y REMOCIÓN DE DATOS EN UNA PILA (LIFO)

Las pilas no son estructuras de datos fundamentales, esto es, no se encuentran definidas como tales en los lenguajes de programación. Las pilas pueden representarse mediante lo siguiente:

- Arreglos.
- Listas enlazadas.

Como en la primera Unidad se estudió sobre los arreglos, sea más conveniente su implementación para la creación de las pilas y colas. Por lo tanto, se debe definir el tamaño máximo de la pila, además de un apuntador al

último elemento insertado en la pila, el cual se denomina SP. La representación gráfica de una pila es la siguiente, denotando como TOPE al T-enésimo elemento de la pila que se inserta:

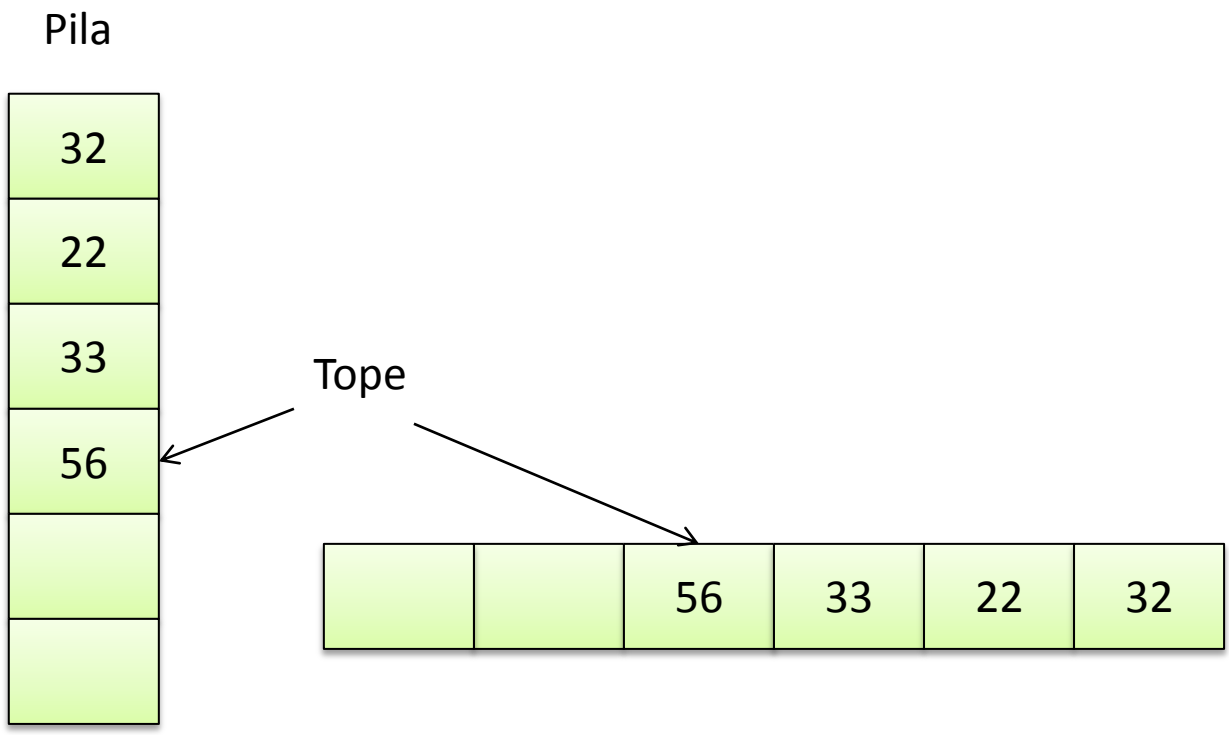


Fig. 2.2.- Representación de una pila

Para implementar el concepto de pilas, se usará como base a los arreglos, éstos estarán limitados a un número máximo de elementos, lo que no permitirá agregar otros registros.

Existen cuatro operaciones básicas que son válidas para el tipo de datos pila, las cuales son las siguientes:

- Lleno (Pila).
- Vacio (Pila).
- Agregar (Elementos, Pila).
- Eliminar (Pila).

Los algoritmos principales serán Agregar y Eliminar, debido a que Crear y Lleno sólo devolverán valores boleanos, para saber si una pila está creada o si tiene datos.

Pseudocódigo 1.- Vacía (Pila, Tope, Res1)

// Este algoritmo verifica si la pila tiene datos, asignando a Res1 el valor de verdadero

```
Si Tope=0 entonces
        Res1 ← Verdadero
Si_no
        Res1 ←Falso
Fin_Si
```

Pseudocódigo 2.- Lleno (Pila, Tope, Max, Res2)

// Este algoritmo verifica si la pila tiene espacio, asignando a Res2 el valor de
// verdadero, definiendo como Max al tamaño del arreglo

```
Si Tope=Max  entonces
        Res2 ← Verdadero
Si_no
        Res2←Falso
Fin_Si
```

Pseudocódigo 3.- Agregar

// Este algoritmo añade un elemento a la pila

```
Llamar Lleno (Pila, Tope, Max, Res)
Si Resp2=Verdadero entonces
        Mostrar “Error pila llena, desbordamiento de datos”
Si_no
        Tope← Tope + 1
        Pila[Tope] ←Dato
Fin_Si
```

Pseudocódigo 4.- Eliminar

// Este algoritmo añade un elemento a la pila

```
Llamar Vacio(Pila, Tope,  Res)
Si Resp1=Verdadero entonces
        Mostrar “Error pila vacía”
Si_no
```

Tope← Tope - 1

Dato ← Pila[Tope]

Fin_Si

ACTIVIDAD DE APRENDIZAJE

1. Desarrolla un programa que permita controlar los autos que entran en un estacionamiento, empleando el número de placa, a través de pilas.

2.4. INSERCIÓN Y REMOCIÓN DE DATOS EN UNA COLA SIMPLE Y CIRCULAR

Podemos representar el uso de las colas de dos formas:

- Como arreglos.
- Como listas ordenadas.

Al igual que las pilas, trataremos las colas como arreglos, en donde debemos definir el tamaño de la cola y dos apuntadores, uno para accesar el primer elemento de la lista y otro que guarde el último. En lo sucesivo, al apuntador del primer elemento se le denominará Inicial, al de el último elemento Final y Max, para definir el número máximo de elementos en la cola.

La cola lineal es un tipo de almacenamiento creado por el usuario que trabaja bajo la técnica FIFO (primero en entrar primero en salir). Las colas lineales se representan gráficamente de la siguiente manera:

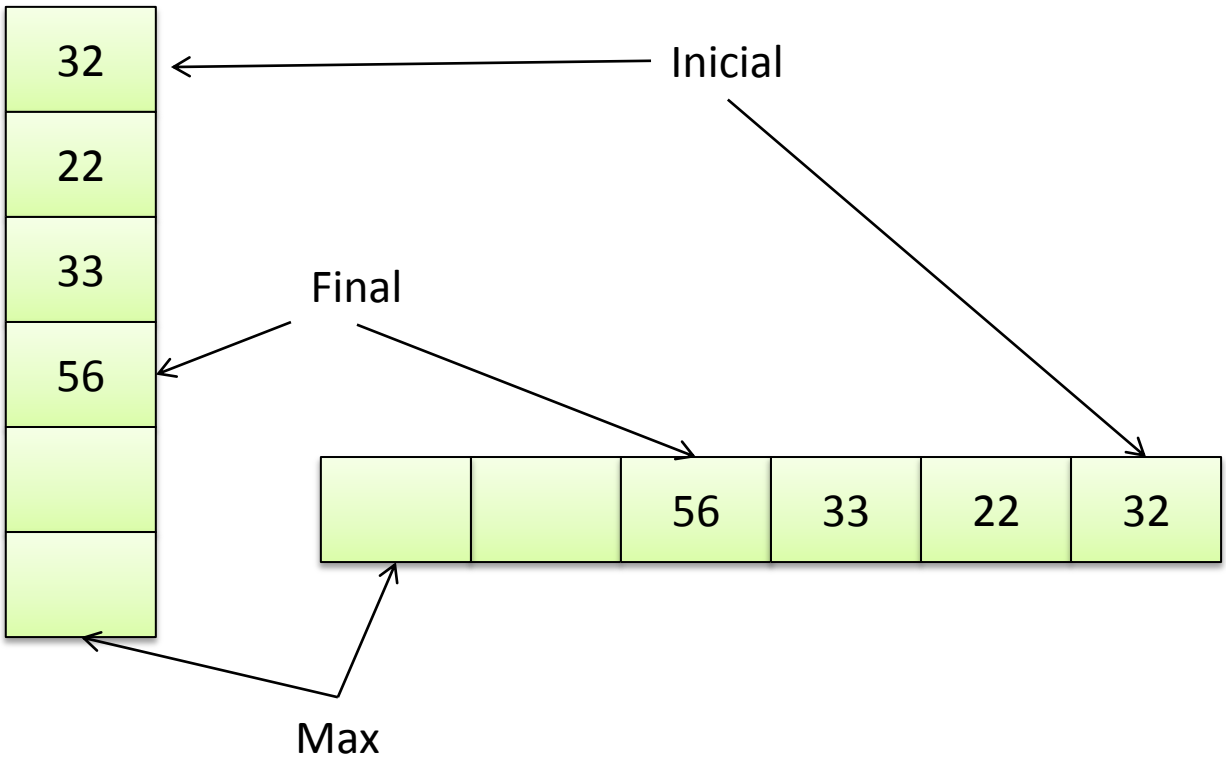


Fig. 2.3. Representación gráfica de una Cola

Las colas también se utilizan en muchas maneras en los sistemas operativos para planificar el uso de los distintos recursos de la computadora. Uno de estos recursos es la propia CPU (Unidad Central de Procesamiento).

Si está trabajando en un sistema multiusuario, cuando le dice a la computadora que ejecute un programa concreto, el sistema operativo añade su petición a su “cola de trabajo”.

Cuando su petición llega al frente de la cola, el programa solicitado pasa a ejecutarse. Igualmente, las colas se utilizan para asignar tiempo a los distintos usuarios de los dispositivos de entrada/salida (E/S), impresoras, discos, cintas y demás. El sistema operativo mantiene colas para peticiones de imprimir, leer o escribir en cada uno de estos dispositivos.

Las operaciones que se pueden implementar en una cola son las siguientes:

- Insertar.
- Eliminar.

Hay que recordar siempre que la inserción se realiza al Final de la cola, mientras que en Inicial llevará el control de las eliminaciones.

```
Algoritmos 1. InsertarDato
Si Final < Max entonces
        Final ← Final + 1
        Cola [ Final ] ←Dato
        Si Final=1 entonces
                Inicial ← 1
        Si_no
                Mostrar “Error”
        Fin_si
Fin_si
```

```
Algoritmo 2.- EliminarDato
Si Inicial ≠ 0 entonces
        Datos ←Cola [ Inicial ]
        Si Inicial= Final entonces
                Frente←0
                Final ← 0
        Si_no
                Frente ← Frente – 1
        Fin_si
Fin_Si
```

Colas circulares

Las colas lineales tienen un grave problema, como las extracciones sólo pueden realizarse por un extremo, puede llegar un momento en que el apuntador Inicial sea igual al máximo número de elementos en la cola, siendo que al frente de la misma existan lugares vacíos, y al insertar un nuevo elemento nos mandará un error.

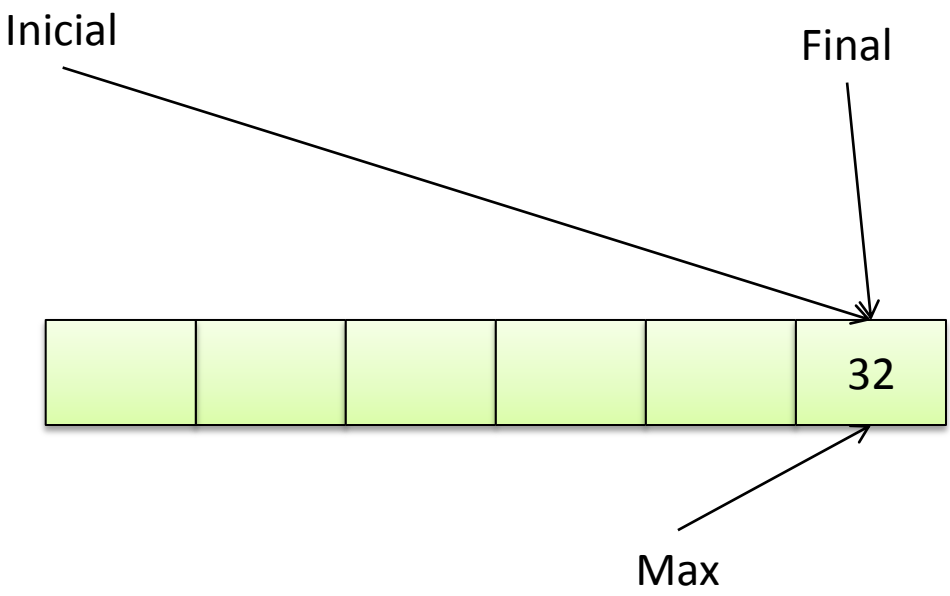


Fig. 2.4.- Problemas con las Colas al llegar al elemento final

Para solucionar el problema de desperdicio de memoria, se implementaron las colas circulares, en las cuales existe un apuntador desde el último elemento al primero de la cola. Así, al momento de no haber más posiciones al final de la cola, se reasignan a los elementos vacios de la cola.

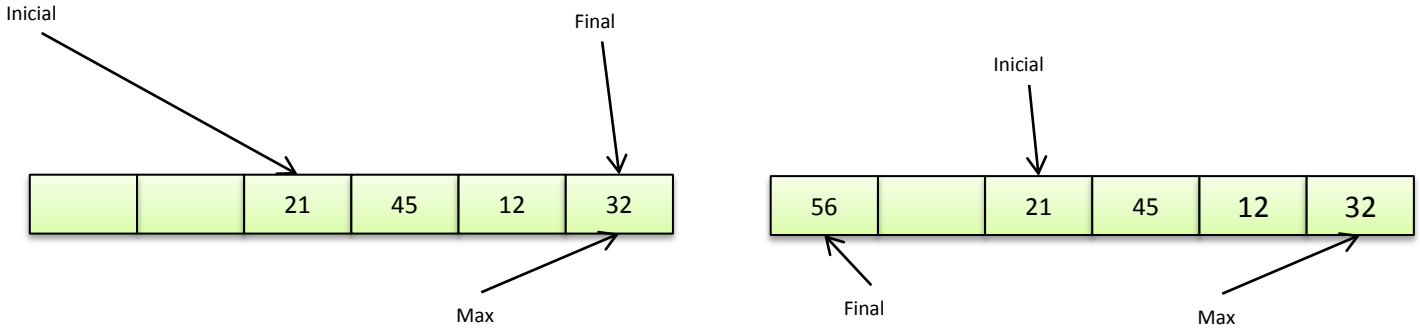


Fig. 2.5. Cola circular

Algoritmo para InsertarColaCircular

Si ((Final = Max) y (Frente = 1)) o ((Final + 1) = Frente) entonces

    Mostrar “Desbordamiento, cola llena”

Si_no

    Si Final=Max

        Final ←1

    Si_no

```
            Final ← Final +1
        Fin_Si
        ColaCir[Final]←Datos
        Si Frente=0 entonces
            Frente←1
        Fin_Si
Fin_Si

Algoritmo para EliminarColaCircular
Si Frente=0
        Mostrar “Error cola vacia”
Si_no
        Dato=ColaCir[Frente]
        Si Frente=Final entonces
            Frente ←0
            Fina←0
        Si_no
            Si Frente=Max entonces
                Frente ←1
            Si_no
                Frente←Frente +1
            Fin_si
        Fin_si
Fin_Si
```

ACTIVIDAD DE APRENDIZAJE

1. Desarrolla un programa que permita controlar los autos que entran en un estacionamiento, empleando el número de placa, a través de colas circulares.

2.5 PROBLEMAS

A continuación, se realizarán dos programas que permitirán observar la codificación en el lenguaje de programación Java, sobre el manejo de las Pilas y Colas, los métodos de acceso y el cumplimiento de sus algoritmos.

Problema 1. Empleando las estructuras de datos, analizar el siguiente programa que permite la captura y eliminación de códigos de libros, en un arreglo de 10 elementos controlando los accesos través de pilas.

```
import javax.swing.JOptionPane;

public class Libros {
   int Max=10, Tope=0;
   int Pila[]= new int[Max];
   String Dato;
   boolean Res1=false, Res2=false;
   public static void main(String arg[]){
      Libros Acceso=new Libros();
   }

   public Libros(){
      int op=0;
     while(op!=4){
      op=Integer.parseInt(JOptionPane.showInputDialog(
            "Opciones Para La Pila" +
            "\n [1] Agregar Dato" +
            "\n [2] Eliminar Dato" +
            "\n [3] Mostrar Pila" +
            "\n [4] Terminar"
            ));
      switch(op){
         case 1:{AgregarPila(); break;}
         case 2:{BorrarPila(); break;}
         case 3:{MostrarPila(); break;}
      }

     }
   }
```

```
public void AgregarPila(){
          PilaLleno();
          if (Res1==true){
             JOptionPane.showMessageDialog(null, "Pila  Llena");
          } else{
             Dato=JOptionPane.showInputDialog("Codigo:");
             Pila[Tope]=Integer.parseInt(Dato);
             Tope++;
          }
}

public void PilaLleno(){
        if (Tope==Max) Res1=true;
        else   Res1=false;
}

public void MostrarPila(){
    String Cadena="";
     for (int i=0;i<Max;i++){
         Cadena+="Posicion "+ i +" ---> "+Pila[i]+"\n";
     }
     JOptionPane.showMessageDialog(null, Cadena);
}

public void BorrarPila(){
    PilaVacia();
    if (Res2==true){
       JOptionPane.showMessageDialog(null, "Pila  Vacia");
    } else{
       Tope-=1;
       JOptionPane.showMessageDialog(null, "Dato eliminado "+Pila[Tope]);
       Pila[Tope]=0;

    }
```

```
    }

    public void PilaVacia(){
            if (Tope==0) Res2=true;
            else Res2=false;
    }

}
```

Problema 2. Empleando las estructuras de datos, analizar un programa que permita la captura de una lista de clientes mediante su código, para que sean atendidos por un personal, dicha lista será manejada en un arreglo de 20 elementos, permitiendo ser controlado por el criterio de las Colas (Primero en llegar, primero en salir).

```
import javax.swing.JOptionPane;

public class Clientes {
    int Max=20, Frente=-1, Final=0;
    int Cola[]= new int[Max];
    String Dato;
    boolean Res1=false, Res2=false;

    public static void main(String arg[]){
            Clientes Acceso=new Clientes();
    }

    public Clientes(){
    int op=0;
      while(op!=4){
        op=Integer.parseInt(JOptionPane.showInputDialog(
                "Opciones el registro de clientes" +
                "\n [1] Agregar Dato" +
```

```
            "\n [2] Eliminar Dato" +
            "\n [3] Mostrar Elementos" +
            "\n [4] Terminar"
            ));
      switch(op){
         case 1:{AgregarCola(); break;}
         case 2:{BorrarCola(); break;}
         case 3:{MostrarCola(); break;}
      }

     }
   }

   public void MostrarCola(){
       String Cadena="";
        for (int i=0;i<Max;i++){
            Cadena+="Posicion "+ i +" ---> "+Cola[i]+"\n";
        }
        JOptionPane.showMessageDialog(null, Cadena);
   }

   public void AgregarCola(){
          if (Final<Max){
              Dato=JOptionPane.showInputDialog("Codigo:");
              Cola[Final]=Integer.parseInt(Dato);
              Final++;
              if (Final==1) Frente=0;
          } else
                JOptionPane.showMessageDialog(null,"No hay elementos
disponibles en Cola..");

   }

   public void BorrarCola(){
```

```
        if(Frente!=-1){
            JOptionPane.showMessageDialog(null, "Dato eliminado "+Cola[Frente]);
            Cola[Frente]=0;
            if (Frente==(Final-1)){
                System.out.print(Frente+"-"+Final);
                    Frente=-1;
                    Final=0;
            } else {
                Frente++;
            }
        }
        else {
            JOptionPane.showMessageDialog(null,"Datos Vacios");
        }
    }

}
```

AUTOEVALUACIÓN

1.- Es una lista de elementos en la cual se puede insertar y eliminar elementos sólo por uno de los extremos. ( ) Tope

2.- Estos elementos son eliminados en orden inverso al que se insertaron. Es decir, el último en entrar es el primero en salir. ( ) Agregar

3.- Es una estructura de almacenamiento donde los datos van a ser insertados por un extremo y serán extraídos por otro ( ) Pilas

4.- Es un método empleado en las filas de un banco, los clientes llegan se colocan en la fila y ( ) Arreglos

esperan su turno para salir.

5.- Es una manera de representar las pilas ( ) Colas

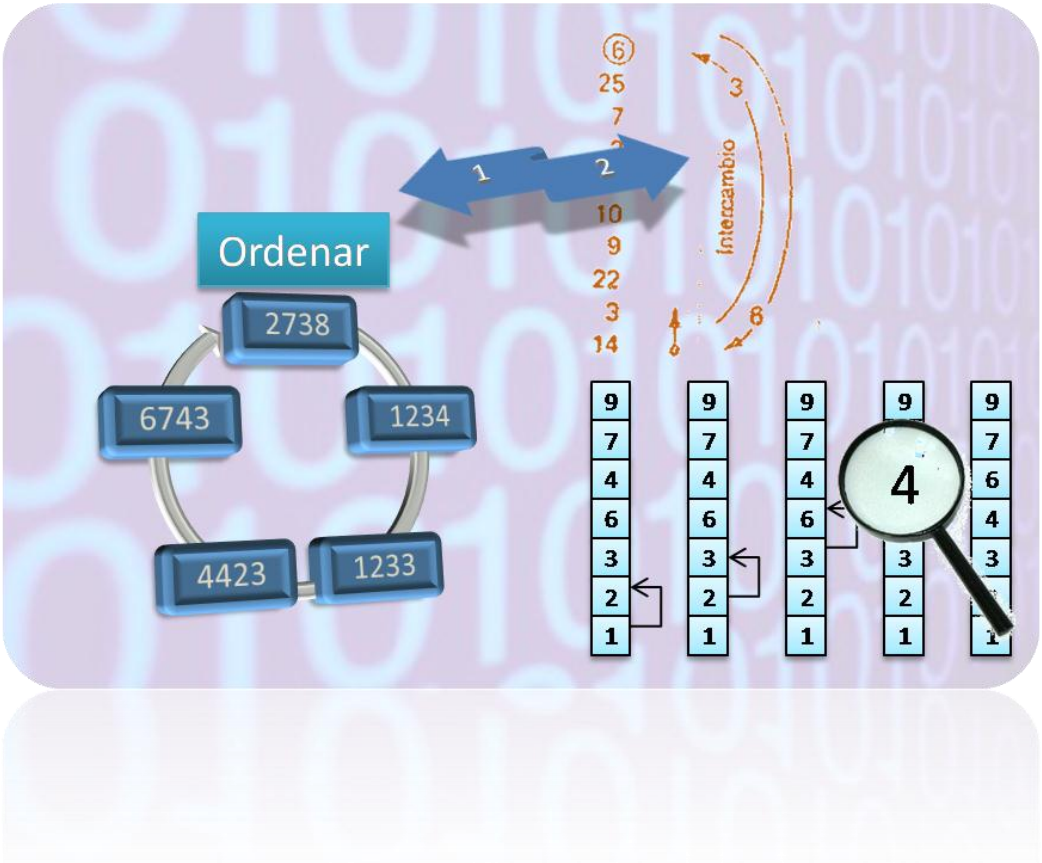
6.- Es el T-enésimo elemento de la pila que se inserta ( ) Colas

7.- Es una operación básica de las pilas ( ) Pilas

Respuestas: 6, 7, 2, 5, 3, 4, 1

UNIDAD 3

ALGORITMOS DE ORDENAMIENTO Y BÚSQUEDA



OBJETIVO

Identificar y analizar los métodos más comunes para el ordenamiento y las búsquedas de la información dentro de estructuras de datos.

TEMARIO

3.1. MÉTODO DE BURBUJA

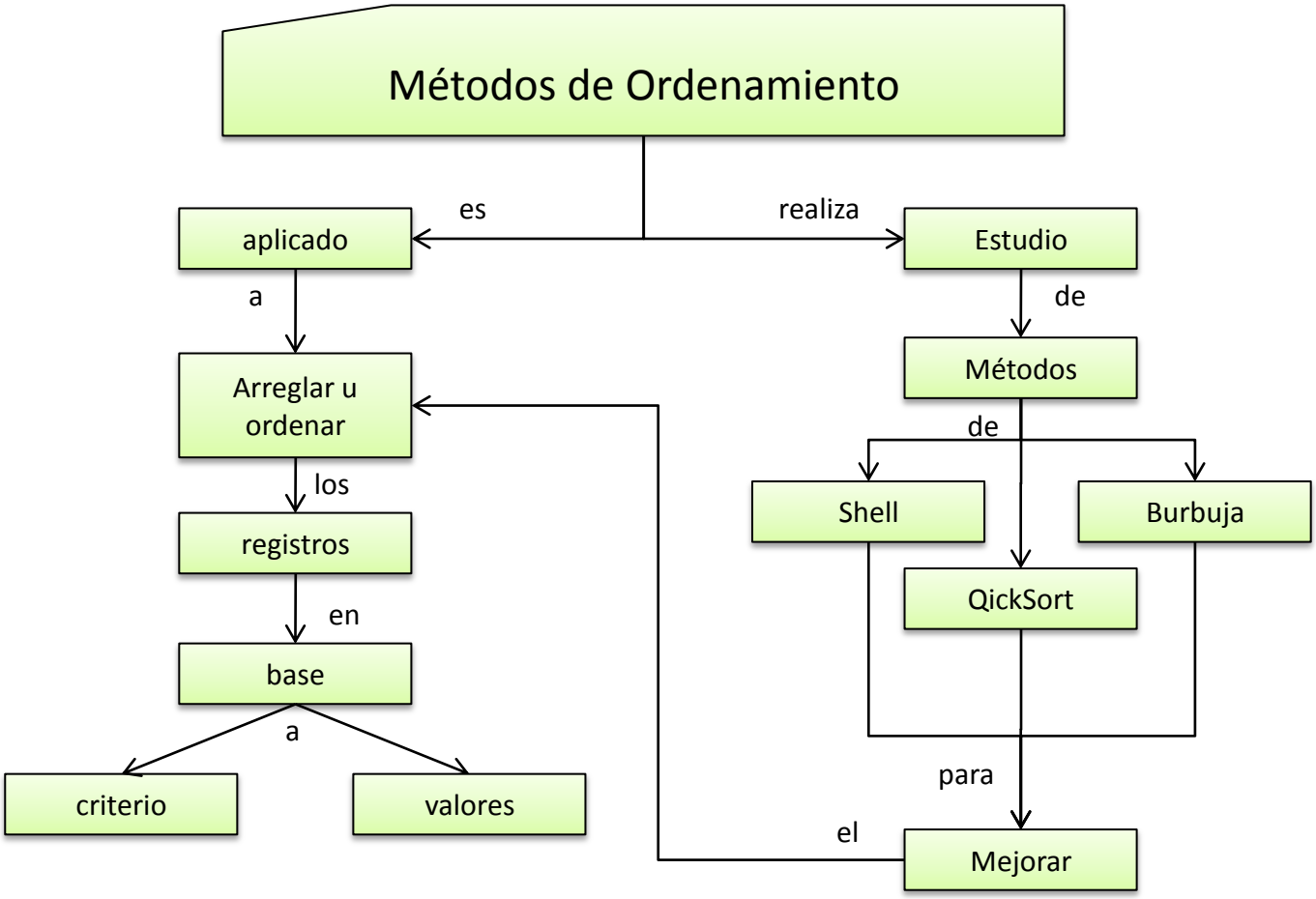
3.2. MÉTODO SHELL

3.3. MÉTODO DE QUICKSORT

3.4. BÚSQUEDA SECUENCIAL

3.5. BÚSQUEDA BINARIA

MAPA CONCEPTUAL



INTRODUCCIÓN

En la siguiente Unidad emplearemos un pequeño descanso sobre nuevos conocimientos en estructura de datos, en su lugar, profundizaremos sobre mecanismos que nos permitirán mejorar la calidad de la información contenida en los arreglos y nos referimos a los métodos de ordenamiento y búsqueda.

La búsqueda y ordenamiento son procesos estrechamente relacionados, la búsqueda es el proceso de localizar un registro con valor particular, mientras que el ordenamiento es el proceso de arreglar los registros de tal manera que queden ordenados en base a un criterio o valor.

En el procesamiento de datos podemos encontrar dos tipos de ordenamientos, los de arreglo y los de archivo. Los primeros se refieren a los datos que se encuentran almacenados en la memoria de la computadora como pueden ser los arreglos. Los de archivo son aquellos que se encuentran almacenados en un medio de almacenamiento como los discos duros.

Ahora nos enfocaremos en la implementación y prueba de los métodos de ordenamiento por arreglos, o como se conoce *métodos de ordenamiento interno*.

3.1. MÉTODO DE BURBUJA

Este método consiste en acomodar los vectores moviendo el mayor hasta la última casilla, comenzando desde la casilla cero, esto se logra comparando valores de llaves y al intercambiarlos si no están en una posición relativa correcta.

Este algoritmo es muy deficiente ya que al ir comparando las casillas para buscar el siguiente más grande, éste vuelve a comparar las ya ordenadas. A pesar de ser el algoritmo de ordenamiento más deficiente que hay, éste es el más usado en todos los lenguajes de programación.

Este método logra la idea básica de la burbuja, que cada valor flote a su posición adecuada mediante comparaciones en pares. Cada paso hace que el valor suba a su posición final, como una burbuja. Ilustremos el concepto con los siguientes valores:

4
7
3
2
9
1
6

Como se ha dicho, la burbuja sube, por lo que cada valor se compara con el que se encuentra arriba de ella, y se intercambia, si la de arriba es más pequeña, después de una pasada, habrá cambios en el ordenamiento como se ilustra a continuación:

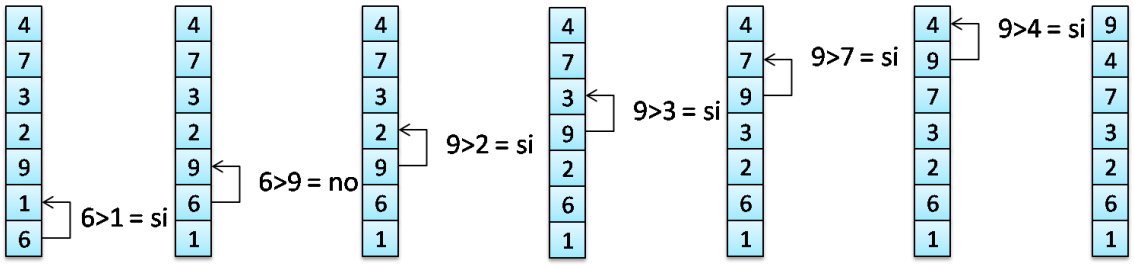


Fig. 3.1 Método de Burbuja Primer ordenamiento

Después del primer ordenamiento se obtiene el número más alto y reinician la comparación desde el último valor hasta completar un ciclo de *n* veces.

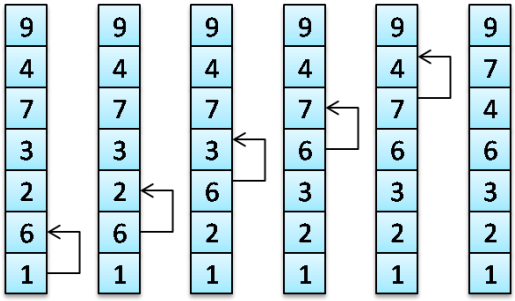


Fig. 3.2 Método de Burbuja segundo ordenamiento

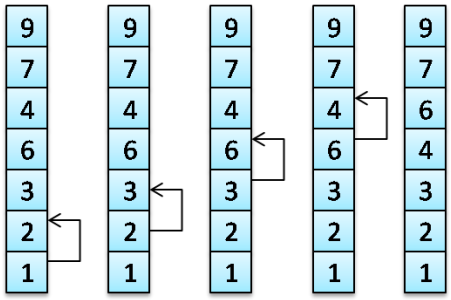


Fig. 3.3 Método de Burbuja tercer ordenamiento

Algoritmo:

```
Repetir con I desde 2 hasta N
        Repetir con J desde N hasta I
                Si  A [ J – 1 ] > A [ J ] entonces
                        Tempo ←A [ J-1 ]
                        A [ J-1 ] ← A [ I ]
                        A [ I ] ← Tempo
                Fin_Si
        Fin_Repetir
Fin_Repetir
```

ACTIVIDAD DE APRENDIZAJE

1. Desarrolla un programa que permita ingresar y ordenar la siguiente numeración 33, 45, 67, 8, 12, 32, 56, 7, 9, 10, 22, 45, 98, 9. De forma ascendente y descendente, usando el método de burbuja.

3.2. MÉTODO SHELL

Este método también se conoce con el nombre de inserción con incrementos decrecientes. El método de *shell* es una versión mejorada del método de inserción directa, recibe ese nombre en honor a su autor Donald L. Shell quien lo propuso en 1959.

En el método de ordenación Shell propone que las comparaciones entre elementos se efectúen con saltos de mayor tamaño pero con incrementos decrecientes, así los elementos quedarán ordenados en el arreglo.

Ejemplo: Se desean ordenar las siguientes claves del arreglo : 15, 67, 08, 16, 44, 27, 12, 35, 56, 21, 13, 28, 60, 36, 07, 10

Primera Pasada.- Los elementos se dividen en 8 grupos:
15, 67, 08, 16, 44, 27, 12, 35 | 56, 21, 13, 28, 60, 36, 07, 10

La ordenación produce:
15, 21, 08, 16, 44, 27, 07, 10, 56, 67, 13, 28, 60, 36, 12, 35

Segunda Pasada .- Se dividen los elementos en 4 grupos:
15, 21, 08, 16 | 44, 27, 07, 10 | 56, 67, 13, 28 | 60, 36, 12, 35

La ordenación produce:
15, 21, 07, 10, 44, 27, 08, 16, 56, 36, 12, 28, 60, 67, 13, 35

Tercera Pasada .- Se divide los elementos 2 grupos
15, 21 | 07, 10 | 44, 27 | 08, 16 | 56, 36 | 12, 28 | 60, 67 | 13, 35

La ordenación produce:
07, 10, 08, 16, 12, 21, 13, 27, 15, 28, 44, 35, 56, 36, 60, 67

Cuarta Pasada.- Divida los elementos en un solo grupo.

La ordenación produce:

07, 08, 10, 12, 13, 15, 16, 21, 27, 28, 35, 36, 44, 56, 60, 67

ACTIVIDAD DE APRENDIZAJE

1. Realizar y entregar una investigación sobre el método Shell con ejemplos
2. Desarrolla un programa que permita ingresar y ordenar la siguiente numeración 33, 45, 67, 8, 12, 32, 56, 7, 9, 10, 22, 45, 98, 9. De forma ascendente y descendente, usando el método de Shell.

3.3. MÉTODO DE QUICKSORT

El método Quickasort emplea la técnica de “divide y vencerás”, el cual se refiere a tomar el arreglo y empezar a dividirlos en secciones más pequeñas, para así poder empezar a ordenar esos arreglos.

El procedimiento consisten en tomar un valor que fungirá como punto de pivote, luego se ordenan los elementos, los menores se mueven a la izquierda y los mayores a la derecha, siempre partiendo del pivote. Continuando se realiza el mismo procedimiento a las otras partes de las divisiones del arreglo.

La mayoría de los programadores realizan los siguientes pasos: primero se toma como punto de pivote al primer elemento del arreglo. Luego, de izquierda a derecha, se busca el elemento mayor, así como de derecha a izquierda el elemento menor, una vez encontrados éstos, se intercambian de posición. Estos pasos se realizan una y otra vez hasta que los dos procesos se encuentren y no haya más elementos.

A continuación aplicaremos el método de ordenamiento QuickSort para ordenar el siguiente arreglo: {21,40,4,9,10,35}.

Como se explicó, primero se toma como punto de pivote al primer elemento, en este caso el 21. En segundo lugar se realizan las búsquedas (Izquierda-Derecha) se encuentra el valor 40, la otra búsqueda encuentra el valor 10 (Derecha-Izquierda). Se realiza el intercambio quedando de la siguiente forma:

{21,10,4,9,40,35}

El ciclo de las búsquedas se reinicia encontrando el valor 40 como mayor, y el valor 9 como el elemento menor, en este caso ya se cruzaron las búsquedas, así que se procede a cruzar el menor con el pivote, quedando:

{9,10,4,21,40,35}

Partiendo del punto del pivote, dividiremos el arreglo en otros dos más pequeños {9,10,4} y el {40,35}, ahora el método de ordenamiento QuickSort se aplica a cada uno de los arreglos.

ACTIVIDAD DE APRENDIZAJE

1. Realizar y entregar una investigación sobre el método Quicksort con ejemplos.
2. Desarrolla un programa que permita ingresar y ordenar la siguiente numeración 33, 45, 67, 8, 12, 32, 56, 7, 9, 10, 22, 45, 98, 9. De forma ascendente y descendente, usando el método de Quicksort.

3.4. BÚSQUEDA SECUENCIAL

Una de la operación más común empleada con los arreglos es la búsqueda de elementos, y la técnica más elemental empleada para realizar este trabajo es la búsqueda secuencial. La búsqueda secuencial es una técnica en la cual se toma un valor clave (Elemento a buscar) y se empieza a comparar con todos los demás elementos, registro por registro, el resultado de la comparación es el índice o posición del elemento y en caso de no encontrar valor, el resultado puede definirse como nulo.

Del anterior término, se deriva su nombre de *Búsqueda secuencial,* pues se comparan secuencialmente todos los elementos, desde el inicio hasta el fin de arreglo uno por uno, hasta que el elemento del arreglo se encuentre o hasta que se llegue al final del arreglo. La existencia se puede asegurar desde el momento en que el elemento es localizado, pero no se puede asegurar la no existencia hasta no haber analizado todos los elementos del arreglo.

La búsqueda secuencial funciona de forma lineal y es muy útil en arreglos de pocos elementos, o bien para arreglos que no estar ordenados. Su

efectividad y velocidad se incrementa si los elementos de un arreglo ya se encuentran ordenados, esto es porque reduce el área de búsqueda eliminando los registros que están sobre el elemento clave.

Algoritmo:

```
Iniciar
I←1
Bandera←Falso
Repetir mientras (I<=N) y (Bandera=Falso)
        Si V [ I ] =X entonces Bandera=Verdadedo
                Si_no I=I+1
        Fin_si
Fin_Repetir
Si Bandera=Verdadero entonces
        Mostrar “Elemento en la Posición I”
Si_no
        Mostrar “Elemento no encontrado”
Fin_si
Fin_Inicio
```

ACTIVIDAD DE APRENDIZAJE

1. Realizar y entregar una investigación sobre el término búsqueda y búsquedas secuenciales.
2. Desarrolla un programa que permita ingresar siguiente 30 números. Luego debe permitir realizar la búsqueda de uno de los números, cualesquiera, empleando la búsqueda secuencial.

3.5. BÚSQUEDA BINARIA

Se utiliza cuando el vector en el que queremos determinar la existencia o no de un elemento está ordenado, o puede estarlo, este algoritmo reduce el tiempo de búsqueda considerablemente, ya que disminuye exponencialmente con el número de iteraciones.

Este algoritmo está altamente recomendado para buscar en arreglos enormes: En uno de 50 000 000 elementos, tarda 26 iteraciones en ejecutarse 1, suponiendo que la búsqueda falla, sino, siempre tarda menos en buscarlo.

Para implementar este algoritmo, se compara el elemento a buscar con un elemento cualquiera del arreglo (normalmente el elemento central), si el valor de éste es mayor que el del elemento buscado, se repite el procedimiento en la parte del arreglo que va desde el inicio de éste hasta el elemento tomado, en caso contrario se toma la parte del arreglo que va desde el elemento tomado hasta el final. De esta manera obtenemos intervalos cada vez más pequeños, hasta que se obtenga un intervalo indivisible, con el elemento buscado como elemento central. Si el elemento no se encuentra dentro de este último, entonces se deduce que el elemento buscado no se encuentra en el arreglo.

Si la tabla de números está ordenada, por ejemplo, en orden creciente, es posible utilizar para la búsqueda un algoritmo más eficiente que se basa en un concepto muy utilizado en la programación: dividir para vencer.

Si está ordenada la tabla y miramos el número situado en la mitad para ver si es mayor o menor que el número buscado (o con suerte igual), sabremos si la búsqueda ha de proceder en la subtabla con la mitad de tamaño que está antes o después de la mitad. Si se repite recursivamente el algoritmo al final o bien encontraremos el número sobre una tabla de un sólo elemento, o estaremos seguros de que no se encuentra allí.

```
Algoritmo: (N es el número de componente)
Iniciar
Izq←1
Der ←N
Bandera←Falso
Repetir mientras (Izq <= Der) y (Bandera=falso)
        Cen←Parte_Entera((Izq+Der)/2)
        Si X=V [Cen] entonces
                Bandera←Verdadero
        Si_no
                Si X>v[Cen] entonces
```

```
                Izq←Cen+1
            Si_no
                Der ← Cen-1
            Fin_si
        Fin_si
Fin_si
Si (Bandera=Verdadero) entonces
            Mostrar “Elemento en la posición Cen”
Si_No
            Mostrar ”Elemento no está en registro”
Fin_si
Fin_Inicio
```

ACTIVIDAD DE APRENDIZAJE

1. Realizar y entregar una investigación sobre el términos búsqueda binaria.
2. Desarrolla un programa que permita ingresar 30 números. Luego debe permitir realizar la búsqueda de uno de los números, cualesquiera, empleando la búsqueda binaria.

AUTOEVALUACIÓN

1. Es el proceso de localizar un registro con valor particular ( ) Shell

2. Es el proceso de arreglar los registros de tal manera que queden ordenados en base a un criterio o valor. ( ) QuickSort

3. Son los tipos de ordenamientos ( ) Método de Burbuja

4.- Este método consiste en acomodar los vectores moviendo el mayor hasta la última casilla comenzando desde la casilla cero, esto se logra comparando valores de llaves e intercambiarlos si no están en una posición relativa correcta ( ) Búsqueda secuencial

5.- Este método también se conoce con el nombre de inserción con incrementos decrecientes ( ) Arreglo y Archivos

6.- Este método se basa en la táctica "divide y vencerás" ( ) Búsqueda secuencial

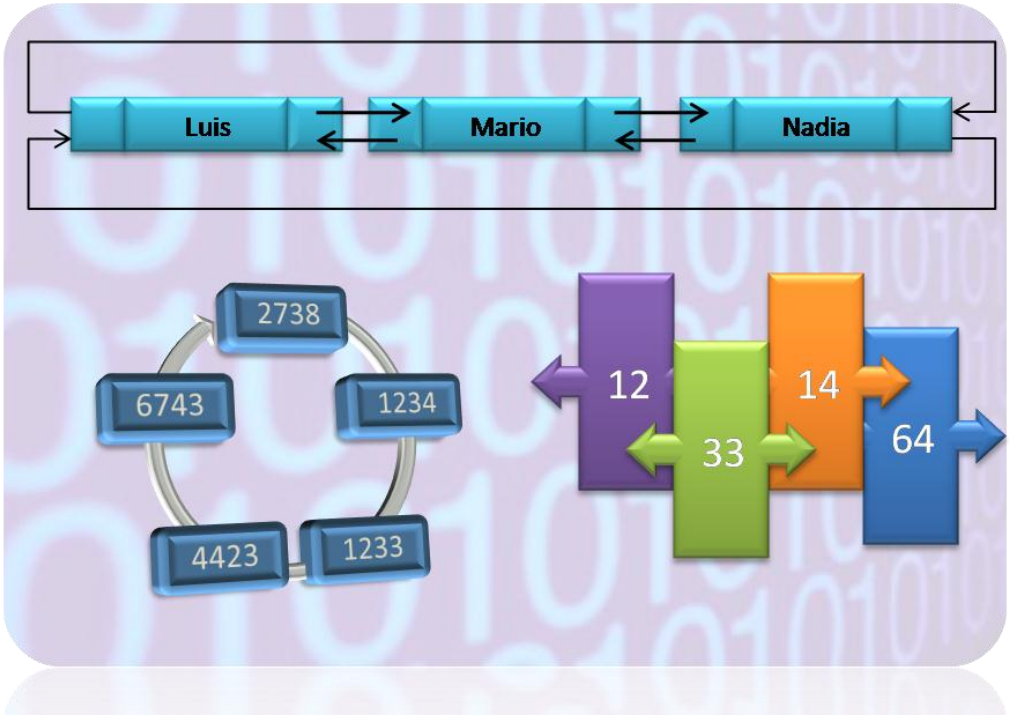
7.- es la técnica más simple para buscar un elemento en un arreglo. Consiste en recorrer el arreglo elemento a elemento e ir comparando con el valor buscado (clave) ( ) Ordenamiento

8.- Se utiliza cuando el contenido del vector no se encuentra o no puede ser ordenado. ( ) Búsqueda

Respuesta: 5, 6, 4, 7, 3, 8, 2, 1

UNIDAD 4

LISTAS



OBJETIVO

Definir y analizar el uso de listas simples, sus variantes y operaciones básicas para el acceso a datos.

TEMARIO

4.1. REPRESENTACIÓN EN MEMORIA

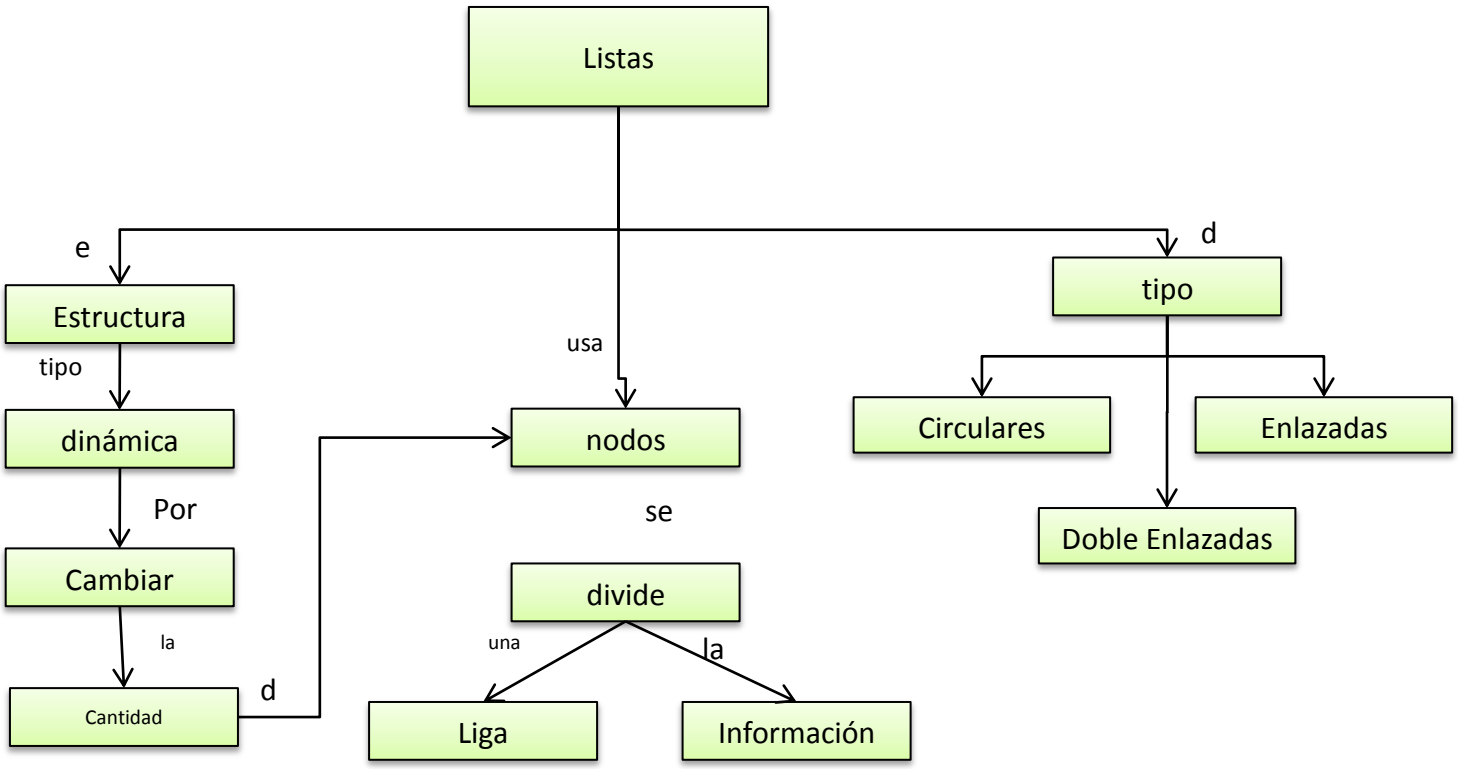
4.2. LISTAS ENLAZADAS

4.3. LISTAS DOBLEMENTE ENLAZADAS

4.4. OPERACIONES CON LISTAS DOBLEMENTE ENLAZADA

4.5. PROBLEMAS

MAPA CONCEPTUAL



INTRODUCCIÓN

Hasta este punto se han visto las estructuras de datos presentadas por Arreglos, Pilas y Colas, éstas son denominadas estructuras estáticas, porque durante la compilación se les asigna un espacio de memoria, y éste permanece inalterable durante la ejecución del programa.[5]

En la siguiente unidad se muestra la estructura de datos “Lista”. La cual es un tipo de estructura lineal y dinámica de datos. Lineal debido a que a cada elemento le puede seguir sólo otro elemento, y dinámica porque se puede manejar la memoria de manera flexible, sin necesidad de reservar espacio con anticipación.[6]

Una de las principales ventajas de manejar un tipo dinámico es que se pueden obtener posiciones de memoria a medida que se va trabajando con el programa, y éstas se liberan cuando ya no se requiere, de ese modo se crea una estructura más dinámica que cambia dependiendo de la necesidad, según se agreguen o eliminen elementos.

Lo anterior solucionaría en gran manera el manejo de los espacios en memoria, necesarios para la solución de problemas y así optimizar el uso de los recursos del sistema, es importante destacar que las estructuras dinámicas no pueden reemplazar a los arreglos en todas sus aplicaciones. Hay casos numerosos que podrían ser solucionados, de modo fácil, aplicando arreglos, en tanto que si se utilizaran estructuras dinámicas, como las listas, la solución de tales problemas se complicaría.

[5] *Cfr*. http://sistemas.itlp.edu.mx/tutoriales/estructuradedatos/t33.html
[6] *Cfr*. http://sistemas.itlp.edu.mx/tutoriales/estructuradedatos/t33.html

4.1. REPRESENTACIÓN EN MEMORIA

La *Lista Lineal* es una estructura dinámica, donde el número de nodos en una lista puede variar a medida que los elementos son insertados y removidos, el orden entre estos se establece por medio de un tipo de datos denominado punteros, apuntadores, direcciones o referencias a otros nodos, es por esto que la naturaleza dinámica de una lista contrasta con un arreglo que permanece en forma constante.

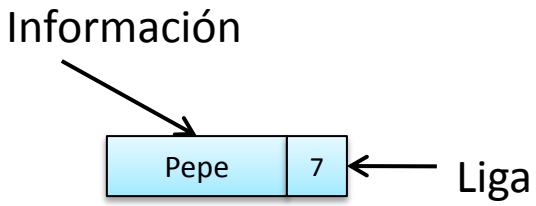


Fig. 4.1. Representación gráfica de un nodo.

Los nodos, en forma general, constan de dos partes: el campo información y el campo liga. El primero contendrá los datos a almacenar en la lista; el segundo será un puntero empleado para enlazar hacia el otro nodo de una lista.

Las operaciones más importantes que se realizan en las estructuras de datos Lista son las siguientes:

- Búsqueda
- Inserción
- Eliminación
- Recorrido

4.2. LISTAS ENLAZADAS

Una lista enlazada se puede definir como una colección de nodos o elementos. “El orden entre estos se establece por medio de punteros; esto es, direcciones

o referencias a otros nodos. Un tipo especial de lista simplemente ligada es la lista vacía."[7]

"El apuntador al inicio de la lista es importante porque permite posicionarnos en el primer nodo de la misma y tener acceso al resto de los elementos. Si, por alguna razón, este apuntador se extraviara, entonces perderemos toda la información almacenada en la lista. Por otra parte, si la lista simplemente ligada estuviera vacía, entonces el apuntador tendrá el valor NULO."[8]

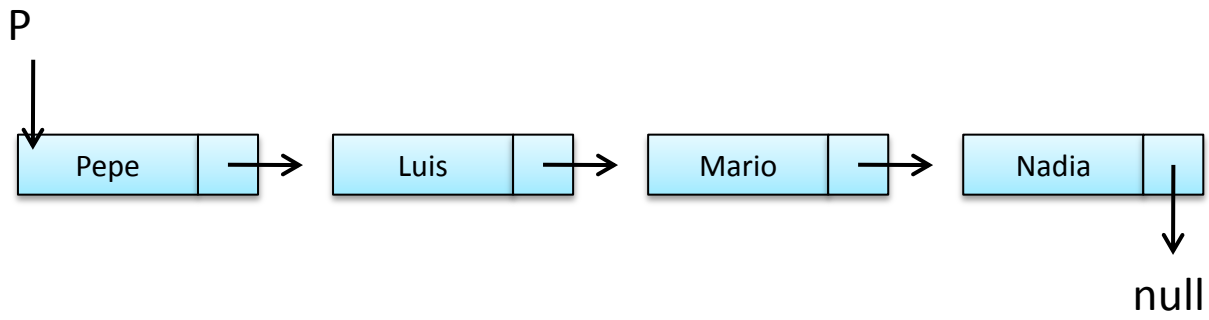


Fig. 4.2. Representación gráfica de una lista enlazada.

Dentro de las listas se pueden mencionar a las listas con cabecera, las cuales emplean a su primer nodo como contenedor de un tipo de valor (*, - , +, etc.). Un ejemplo de lista con nodo de cabecera es el siguiente:

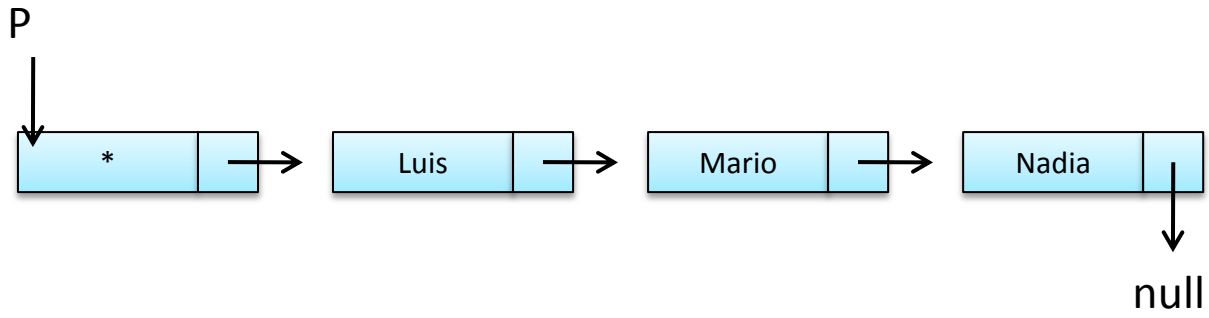


Fig. 4.2. Lista enlazada con nodo de cabecera.

Cuando se emplea este tipo de listas con nodos de cabecera, se ocupa un apuntador, normalmente llamado con el mismo nombre, el cual se utiliza para hacer referencia al inicio o cabeza de la lista. Cuando una lista no contiene un nodo cabecera, se emplea la palabra TOP, que indicará cuál es el

[7] *Cfr*. http://nancynohemi.webuda.com/index.php?option=com_content&view=article&id=40&Itemid=41
[8] Cfr. http://nancynohemi.webuda.com/index.php?option=com_content&view=article&id=40&Itemid=41

primer elemento de la lista, los términos TOP(dato) y TOP(liga) se usan para referirse al dato almacenado y a la liga del nodo siguiente.

Algoritmo de creación

```
top<--NIL
repite
        new(p)
        leer(p(dato))
         si top=NIL entonces
       top<--p
         en caso contrario
       q(liga)<--p
          p(liga)<--NIL
          q<--p
          mensaje('otro nodo?')
           leer(respuesta)
hasta respuesta=no
```

Algoritmo para Recorrido

```
p<--top
mientras p<>NIL haz
       escribe(p(dato))
       p<--p(liga:)
```

Algoritmo para insertar al final

```
p<--top
mientras p(liga)<>NIL haz
       p<--p(liga)
new(q)
p(liga)<--q
q(liga)<--NIL
```

```
Algoritmo para insertar antes/después de 'X' información
p<--top
mensaje(antes/despues)
lee(respuesta)
si antes entonces
        mientras p<>NIL haz
                si p(dato)='x' entonces
                        new(q)
                        leer(q(dato))
                        q(liga)<--p
                        si p=top entonces
                                top<--q
                        en caso contrario
                                r(liga)<--q
                        p<--nil
                en caso contrario
                        r<--p
                        p<--p(link)
si despues entonces
        p<--top
        mientras p<>NIL haz
                si p(dato)='x' entonces
                        new(q)
                        leer(q(dato))
                        q(liga)<--p(liga)
                        p(liga)<--q
                        p<--NIL
                en caso contrario
                        p<--p(liga)
        p<--top
        mientras p(liga)<>NIL haz
                p<--p(liga)
        new(q)
```

```
        p(liga)<--q
        q(liga)<--NIL
```

Algoritmo para borrar un nodo

```
p<--top
leer(valor_a_borrar)
mientras p<>NIL haz
        si p(dato)=valor_a_borrar entonces
                si p=top entonces
                        si p(liga)=NIL entonces
                                top<--NIL
                        en caso contrario
                                top(liga)<--top(liga)
                en caso contrario
                        q(liga)<--p(liga)
                dispose(p)
                p<--NIL
        en caso contrario
                q<--p
                p<--p(liga)
```

Algoritmo de creación de una lista con nodo de cabecera

```
new(cab)
cab(dato)<--'*'
cab(liga)<--NIL
q<--cab
repite
         new(p)
         leer(p(dato))
         p(liga)<--NIL
         q<--p
         mensaje(otro nodo?)
          leer(respuesta)
hasta respuesta=no
```

Algoritmo de extracción en una lista con nodo de cabecera

```
leer(valor_a_borrar)
p<--cab
q<--cab(liga)
mientras q<>NIL haz
        si q(dato)=valor_a_borrar entonces
                p<--q(liga)
                dispose(q)
                q<--NIL
        en caso contrario
                p<--q
                q<--q(liga)
```

ACTIVIDAD DE APRENDIZAJE

1.- Desarrolla un programa que permita ingresar 10 nombres, estructurándolo en una lista enlazada.

4.3. LISTAS DOBLEMENTE ENLAZADAS

Se puede referir a una lista doble o doblemente ligada, a una colección de nodos que emplean además de su dato, dos elementos llamados punteros, los cuales se utilizan para especificar cuál es el elemento anterior y sucesor. Estos punteros se denominan Li (anterior) y Ld (sucesor). Tales punteros permiten moverse dentro de las listas un registro adelante o un registro atrás, según tomen las direcciones de uno u otro puntero.

La estructura de un nodo en una lista doble es la siguiente:

Fig. 4.3. Liga doblemente enlazada.

Podemos mencionar a dos tipos de listas del tipo doblemente ligadas, las cuales se mencionan a continuación:

- Listas dobles lineales. En este tipo de lista el puntero Li del primer elemento apunta a NULL y el último elemento indica en su puntero Ld a NULL.
- Listas dobles circulares. Este otro tipo de lista tiene la particularidad que en su primer elemento el puntero, Li apunta al último elemento de la lista. Y el último elemento indica, en su puntero Ld, a el primer elemento de la lista.

Para lograr un fácil acceso a la información de la lista, se recomienda ocupar dos apuntadores, P y F, que apunten al principio y al final de ésta, respectivamente.[9]

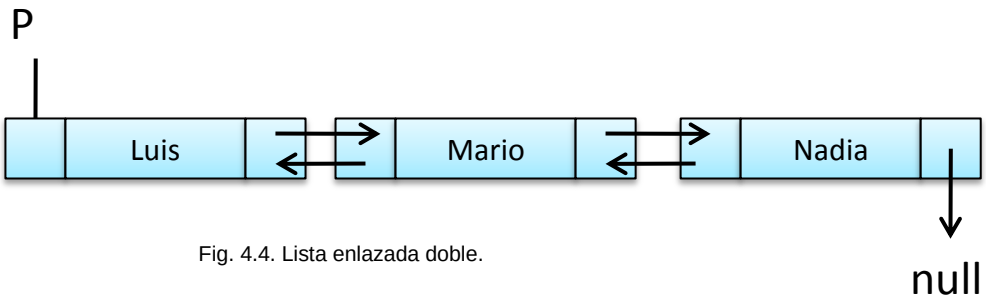


Fig. 4.4. Lista enlazada doble.

En la ilustración siguiente, se ejemplifica una lista doblemente ligada circular, la cual apunta a sus respectivos elementos.

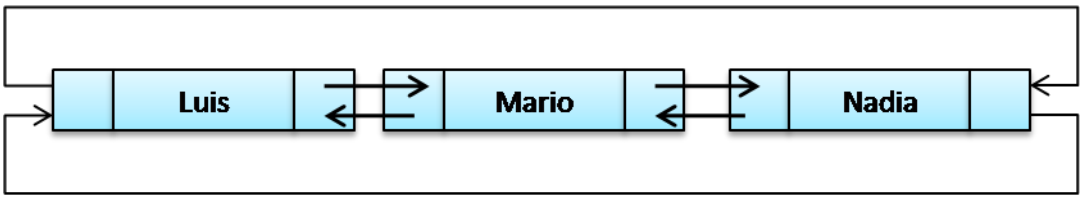


Fig. 4.4.- Lista circular enlazada doble.

ACTIVIDAD DE APRENDIZAJE

1. Realizar y entregar una investigación sobre la creación y uso de las listas doblemente enlazadas.

[9] *Cfr*. http://nancynohemi.webuda.com/index.php?option=com_content&view=article&id=40&Itemid=41

2. Desarrolla un programa que permita ingresar 10 nombres, estructurándolo en una lista doblemente enlazada.

4.4. OPERACIONES CON LISTAS DOBLEMENTE ENLAZADA

Como se mencionó en el tema anterior, las listas doblemente enlazadas nos permitirán acceder a la información de una forma rápida y efectiva, esto se logra mediante el uso correcto de operaciones o procedimientos, es decir, emplear primitivas (algoritmos básicos) para:

- Crear listas.
- Buscar valor
- Insertar valor
- Borrar valor

A continuación se escriben los algoritmos fundamentales para realizas las anteriores operaciones.

Algoritmo para la creación de listas

```
repite
     new(p)
     lee(p(dato))
      si top=nil entonces
       top<--p
       q<--p
     en caso contrario
       q(liga)<--p
       q<--p
    p(liga)<--top
    mensaje (otro nodo ?)
    lee(respuesta)
hasta respuesta=no
```

Algoritmo para recorrer la lista

```
p<--top
repite
      escribe(p(dato))
       p<--p(liga)
hasta p=top
```

Algoritmo para insertar antes de 'X' información

```
new(p)
lee(p(dato))
si top=nil entonces
       top<--p
       p(liga)<--top
en caso contrario
       mensaje(antes de ?)
       lee(x)
       q<--top
        r<--top(liga)
        repite
        si q(dato)=x entonces
                p(liga)<--q
                r(liga)<--p
                si p(liga)=top entonces
                        top<--p
        q<--q(liga)
        r<--r(liga)
      hasta q=top
```

Algoritmo para insertar después de 'X' información

```
new(p)
lee(p(dato))
mensaje(después de ?)
lee(x)
q<--top
r<--top(liga)
repite
     si q(dato)=x entonces
       q(liga)<--p
       p(liga)<--r
     q<--q(liga)
     r<--r(liga)
hasta q=top
```

Algoritmo para borrar

```
mensaje(valor a borrar )
lee(valor_a_borrar)
q<--top
r<--top
p<--top
mientras q(liga)<>top haz
       q<--q(liga)
repite
     si p(dato)=valor_a_borrar entonces
       si p=top entonces
              si top(liga)=top entonces
                     top<--NIL
              en caso contrario
                     top<--top(liga)
                     q(liga)<--top
       en caso contrario
              r(liga)<--p(liga)
       dispose(p)
```

```
      p<--top
    en caso contrario
      r<--p
      p<--p(liga)
hasta p=top
```

ACTIVIDADES DE APRENDIZAJE

1. Realiza y entrega una investigación sobre las operaciones que soportan las listas.
2. Desarrolla un programa que permita ingresar 10 nombre estructurándolo en una lista enlazada circular.

4.5. PROBLEMAS

A continuación recopilaremos ejemplos simples sobre el uso y aplicación de las listas enlazadas. Primero comenzaremos con un programa que permita ingresar tres nombres ya definidos en una lista, para luego mostrarlos en modo consola.

```
import java.util.*;

public class ListarLista {
  public ListarLista() {
      super();
    }

  public static void main(java.lang.String[] args) {
    // Definimos una ArrayList
    List<String> list = new ArrayList<String>();

    list.add("Luisa");
    list.add("Maira");
    list.add("Andes");
```

```
    Iterator iter = list.iterator();
    while (iter.hasNext())
      System.out.println(iter.next());

  }
}
```

A continuación se presenta el ejemplo de una lista enlazada doble con tres nodos, los cuales muestra, en pantalla, antes y después de una eliminación.

```
class DobleEnlace {
  static class Node {
     String Dato;
     Node Siguiente;
     Node Previo;
  }

  public static void main (String [] args) {

     Node topForward = new Node ();
     topForward.Dato = "A";

     Node temp = new Node ();
     temp.Dato = "B";

     Node topBackward = new Node ();
     topBackward.Dato = "C";

     topForward.Siguiente = temp;
     temp.Siguiente = topBackward;
     topBackward.Siguiente = null;

     topBackward.Previo = temp;
```

```
temp.Previo = topForward;
topForward.Previo = null;

System.out.print ("Lista de Datos: ");

temp = topForward;
while (temp != null){
   System.out.print (temp.Dato);
   temp = temp.Siguiente;
}

System.out.println ();

System.out.print ("Lista inversa: ");

temp = topBackward;
while (temp != null){
   System.out.print (temp.Dato);
   temp = temp.Previo;
}

System.out.println ();



temp = topForward.Siguiente;

// Eliminacion del nodo B

temp.Previo.Siguiente = temp.Siguiente;
temp.Siguiente.Previo = temp.Previo;

System.out.print ("Datos despues de la eliminacion de B: ");
```

```
        temp = topForward;
        while (temp != null){
          System.out.print (temp.Dato);
          temp = temp.Siguiente;
        }

        System.out.println ();

        System.out.print ("Lista simple hacia atras: ");

        temp = topBackward;
        while (temp != null){
          System.out.print (temp.Dato);
          temp = temp.Previo;
        }
        System.out.println ();
    }
}
```

AUTOEVALUACIÓN

1.- Es una ventaja de manejarlas ya que se pueden obtener posiciones de memoria a medida que se va trabajando con el programa y estas se libera cuando ya no se requiere ( ) Apuntador

2.- Son más simples de emplear que las Estructuras Dinámicas ( ) Información y liga

3.- Es una estructura dinámica, donde el número de nodos en una lista puede variar a medida que los elementos son insertados y removidos ( ) Estructura de datos Dinámicas

4.- El orden entre Listas se establece por medio de un tipo de datos denominado ( ) Listas simple enlazada

5.- Son las partes de un nodo ( ) Null

6.- Constituye una colección de elementos llamados nodos ( ) Listas

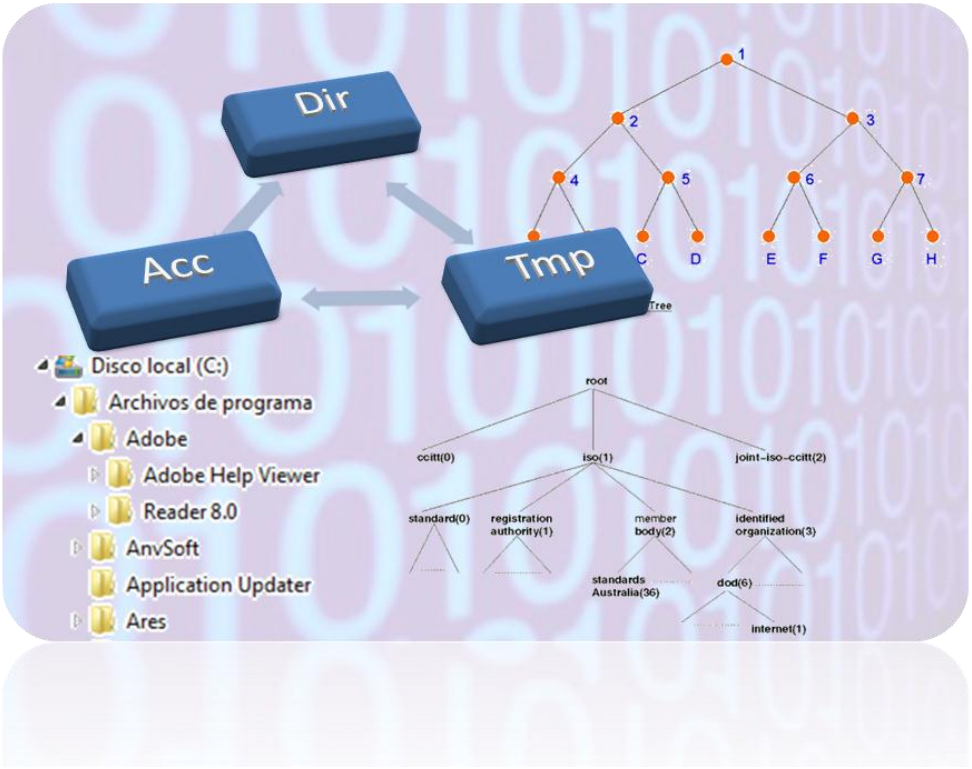
7.- Se da ese valor cuando la lista se encuentra vacía ( ) Puntero

8.- Este elemento es importante al inicio de la lista ya que permite posicionarnos en el primer nodo de la misma y tener acceso al resto de los elementos ( ) Arreglos

Respuesta: 8, 5, 1, 6, 7, 3, 4, 2

UNIDAD 5

ÁRBOLES



OBJETIVO

Comprender la importancia que tienen las estructuras de árboles para la manipulación de procesos complejos de ordenamiento, búsquedas secuenciales y binarias, así como su representación gráfica.

TEMARIO

5.1. TERMINOLOGÍA

5.2. ÁRBOLES BINARIOS Y REPRESENTACIONES GRÁFICAS

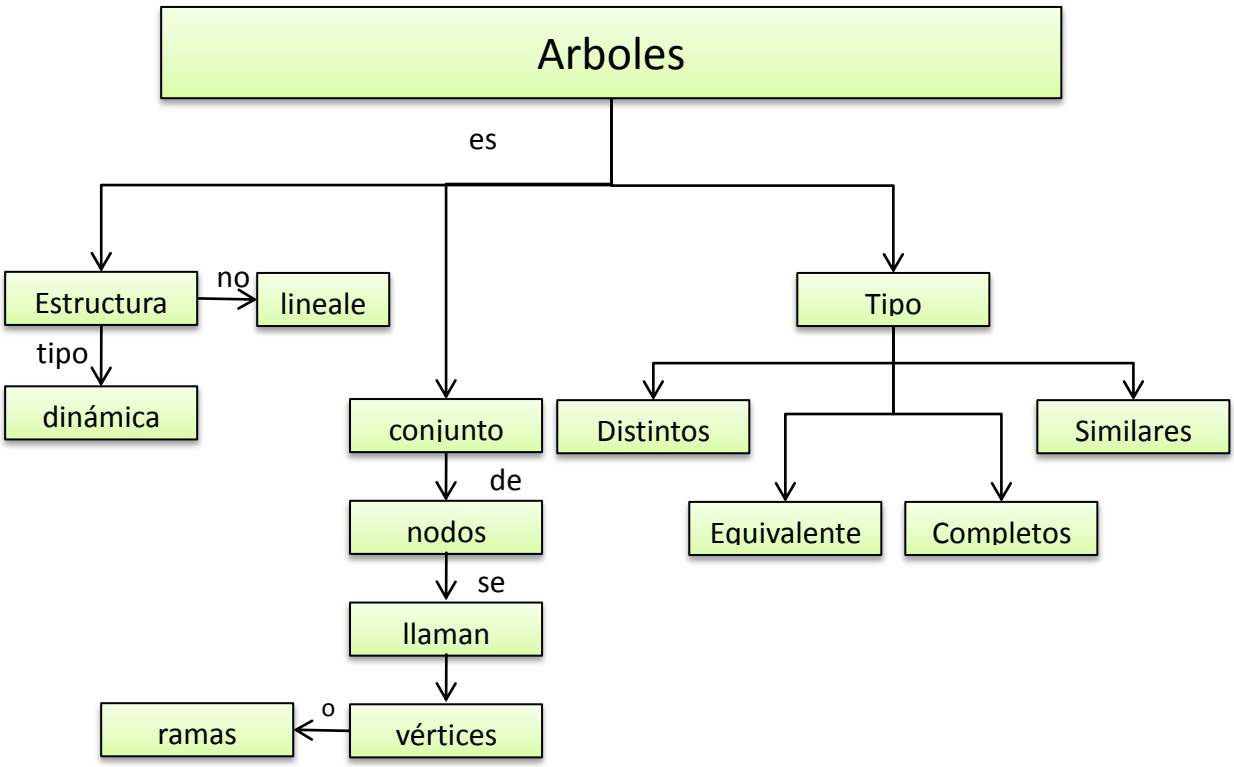
5.3. RECORRIDO DE UN ÁRBOL

5.4. ÁRBOLES ENHEBRADOS

5.5. ÁRBOLES DE BÚSQUEDA

5.6. PROBLEMAS

MAPA CONCEPTUAL



INTRODUCCIÓN

El uso y aplicación de los árboles binarios son variadas y se emplean para representar estructuras de datos en las cuales se tienen que tomar decisiones entre puntos diferentes.

Durante el estudio de esta Unidad, aprenderemos acerca de la eficiencia de los árboles de búsqueda, construir un árbol de búsqueda equilibrado, describir los tipos de movimientos que se realizan para equilibrar un árbol, realizar operaciones de inserción y eliminación de elementos del árbol.

Otro objetivo de esta unidad es conocer las características de los árboles *A*, utilizar su estructura para organizar búsquedas eficientes en bases de datos, implementar algoritmos de búsqueda de una clave, conocer las estrategias que se siguen para la inserción y eliminación de elementos, también se pretende distinguir entre relaciones jerárquicas y otras relaciones.

5.1. TERMINOLOGÍA

En términos matemáticos, *un árbol es cualquier conjunto de puntos, llamados vértices, y cualquier conjunto de pares de distintos vértices, llamados lados o ramas, a una secuencia de ramas*, se le conoce como ruta de cualquier vértice a cualquier otro vértice.

Un árbol es una estructura de datos no lineal, las estructuras de datos lineales se caracterizan por que a cada elemento le corresponde no más que un elemento siguiente. En las estructuras de datos no lineales, como el árbol, un elemento puede tener diferentes "siguientes elementos", introduciendo una estructura de bifurcación, también conocidas como estructuras multi enlazada.

En un árbol no hay lazos o sea, que no hay pasos que comiencen en un vértice y puedan volver al mismo vértice, un árbol puede tener un vértice o nodo llamado raíz, el cual cumple la función de vértice principal. La particularidad del nodo raíz es que no puede ser *hijo* de otro nodo.

Un árbol *A* es "un conjunto finito de uno o más nodos",[10] tales que:

- Existe un nodo especialmente designado y denominado RAIZ(v1) del árbol.
- Los nodos restantes (v2, v3, ..., vn) se dividen en m >= 0 conjuntos disjuntos denominados A1, A2, ..., Am, cada uno de los cuales es a su vez, un árbol. Estos árboles se llaman subárboles (subtree) del RAIZ. Observar la naturaleza recursiva de la definición de árbol.

Aclarado los conceptos anteriores, *en la estructura de datos definiremos a un árbol como un conjunto finito de elementos no vacio en el cual un elemento se denomina raíz y los restantes se dividen en m >= 0 subconjuntos separados, cada uno de los cuales es por sí mismo un árbol*. Cada elemento en un árbol se denomina nodo del árbol.

Mencionemos algunos ejemplos donde la estructura de datos árbol puede ser muy útil y que son de uso común:

[10] Goodrich / Tamassia , *Estructura de datos y algoritmos en java*, p. 348.

- Los sistemas de archivos (file system) de los sistemas operativos, compuestos por jerarquías de directorios y archivos.
- La jerarquía de clases en los lenguajes orientados a objetos.
- La jerarquía de países, provincias, departamentos y municipios que organiza al poder político de una república.

Entre los términos más comunes que se emplean al utilizar los árboles, se pueden destacar los siguientes:[11]

- Hijo. A es hijo de B, sí y sólo sí el nodo A es apuntado por B.
- Padre. A es padre de B, sí y sólo sí el nodo A apunta a B.
- Hermano. Cuando dos nodos descienden de un mismo nodo.
- Grado. Es el total de descendientes de un nodo.
- Nivel. Es el número de descensos que se recorren para llegar a otro nodo.
- Peso. Es el número de nodos sin contar la raíz.
- Longitud de camino. Es el número de arcos que deben ser recorridos para llegar desde la raíz al nodo X. Por definición la raíz tiene longitud de camino 1, y sus descendientes directos longitud de camino 2 y así sucesivamente.

5.2. ÁRBOLES BINARIOS Y REPRESENTACIONES GRÁFICAS

Un árbol binario es empleado para reconocer un conjunto de registros que tienen un identificador o clave única. Cuando el árbol está ordenado se identifica porque el nodo padre es mayor que las claves de su subárbol izquierdo, además de ser menor que todas las claves del subárbol derecho.

[11] *Cfr*. http://www.utpl.edu.ec/wikis/matematicas_discretas/index.php/Teoria_de_%C3%81rboles o http://hellfredmanson.over-blog.es/article-30369340-6.html o http://boards4.melodysoft.com/app?ID=2005AEDII0405&msg=15&DOC=21

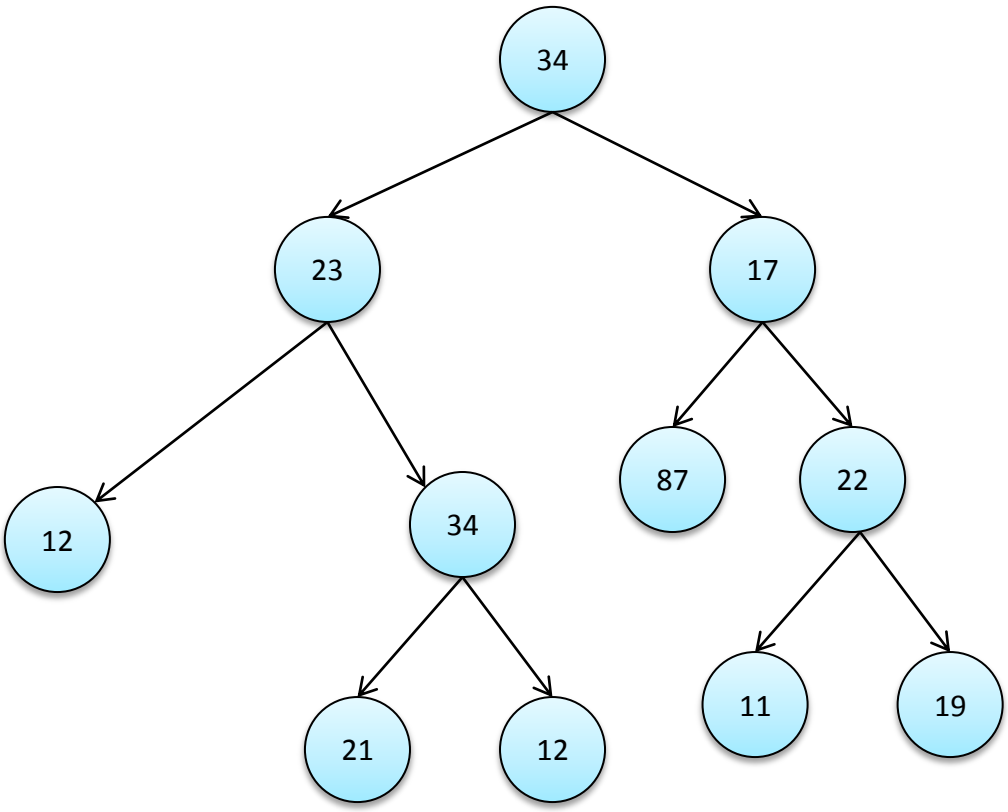


Fig. 5.1. Representación de un árbol.

Para representar un árbol binario puede hacerse de forma tradicional o por arreglos. La primera emplea dos punteros de tipo variables dinámicas o listas; la segunda emplea el uso de arreglos, aunque por dinámica la primera es la más empleada.

Los arboles binarios se pueden expresar mediante registros divididos en por lo menos tres campos, el primero y el último contienen la información de los arboles anterior y siguiente del árbol en cuestión, y un campo más para almacenar el valor,

Podemos mencionar cuatro tipos esenciales de arboles binarios:

1. B. Distinto.
2. B. Similares.
3. B. Equivalentes.
4. B. Completos.

A. B. Distinto. Cuando dos árboles tienen diferentes estructuras se les conoce como árboles distintos. Ejemplo:

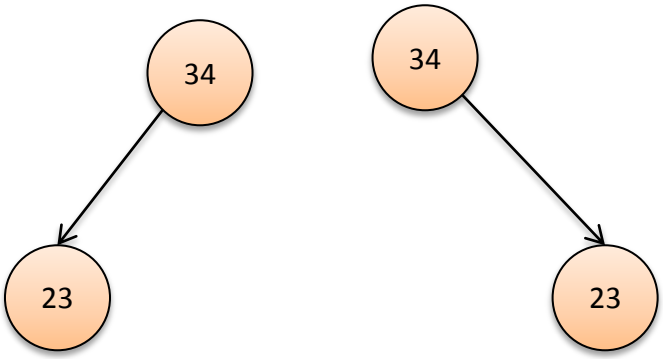


Fig. 5.2. Representación de árboles *distintos*

A. B. Similares. Cuando sus estructuras son iguales, pero los datos contenidos son distintos, se tiene un árbol binario similar. Ejemplo:

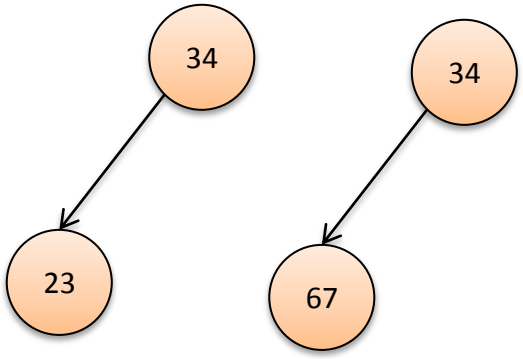


Fig. 5.3. Representación de árboles *similares*.

A. B. Equivalentes. Si los arboles son de tipo similar, pero además contienen los mismos datos, se tienen un árbol equivalente. Ejemplo:

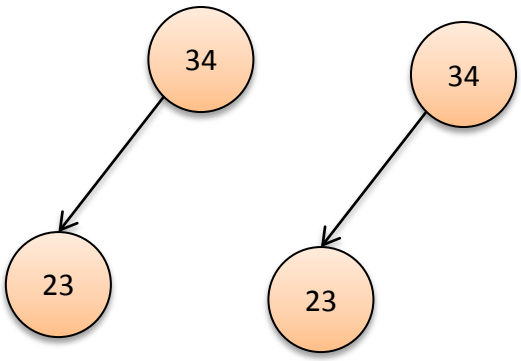


Fig. 5.4. Representación de árboles *equivalentes.*

A. B. Completos. Este tipo de árboles es aquel en el que sus diferentes niveles tienen dos hijos (izquierdo y derecho), acepción del último nodo.

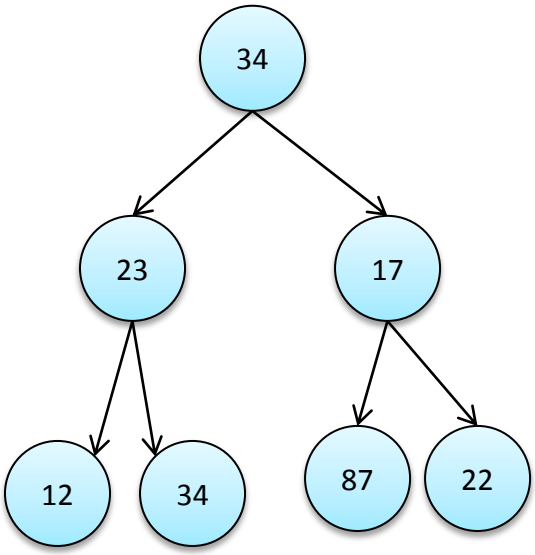


Fig. 5.5. Representación de arboles *completo.*

5.3. RECORRIDO DE UN ÁRBOL

Los árboles de grado superior a dos, reciben el nombre de árboles multicamino. Dentro de la estructura de un árbol, podemos manejar tres formas de recorrerlo, cada una de ellas contiene una secuencia distinta, la cual se presenta a continuación.

Inorden

- Recorrido del subárbol izquierdo en inorden.
- Buscar la raíz.
- Recorrido del subárbol derecho en inorden.

Preorden

- Examinar la raíz.
- Recorrido del subárbol izquierdo en preorden.
- Recorrido del subárbol derecho en preorden.

Postorden

- Recorrido del subárbol izquierdo en postorden.
- Recorrido del subárbol derecho en postorden.
- Examinar la raíz.

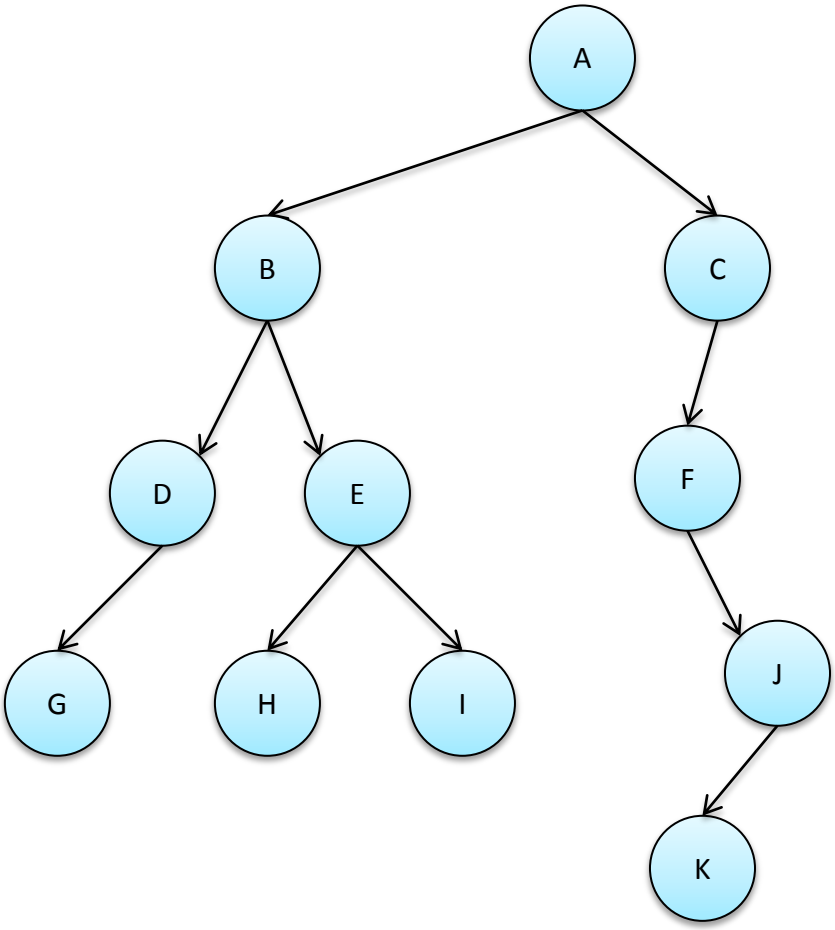


Fig. 5.6. Ejemplo de un árbol.

A continuación se presenta el Pseudocódigo sobre el recorrido del árbol de la figura 5.6, así como el resultado que se obtendría.

Inorden: GDBHEIACJKF
Preorden: ABDGEHICFJK
Postorden: GDHIEBKJFCA

Pseudocódigo:
funcion inorden(nodo)
inicio

```
    si(existe(nodo))
        inicio
            inorden(hijo_izquierdo(nodo));
            tratar(nodo);                    //Realiza una operación en nodo
            inorden(hijo_derecho(nodo));
        fin;
fin;
```

5.4. ÁRBOLES ENHEBRADOS

Otro tipo de árboles binarios que podemos encontrar son los enhebrados, denominado así porque contiene hebras hacia la derecha o a la izquierda. En la siguiente figura se ilustra el ejemplo de un árbol enhebrado a la derecha.

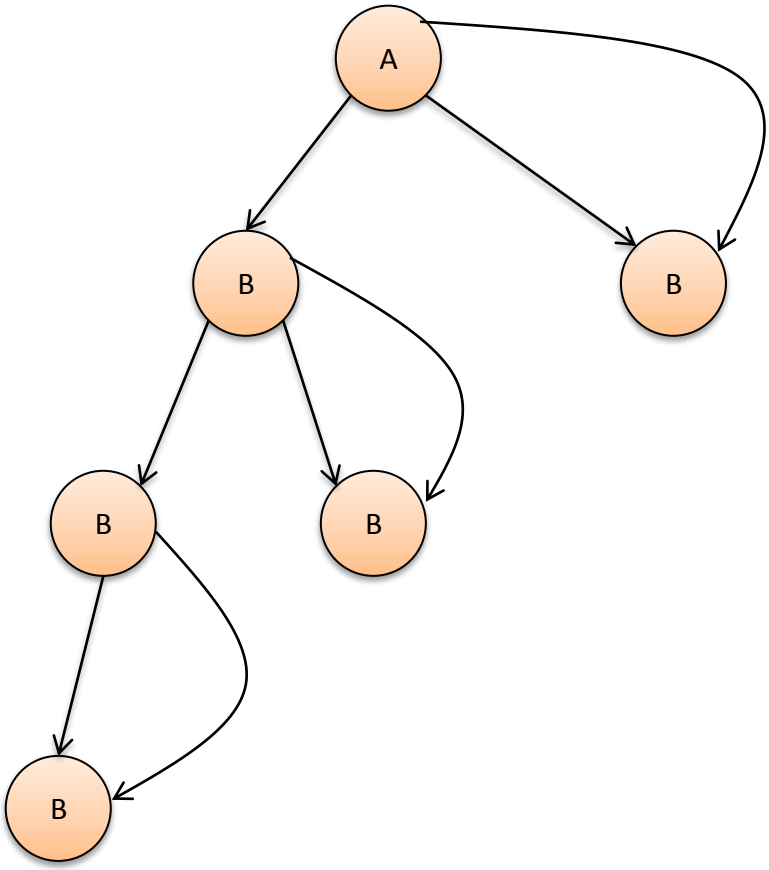


Fig. 5.7 Árbol enhebrado.

Los árboles enhebrados se utilizan para el mejor aprovechamiento de la memoria. Sus ventajas principales son las siguientes: no se requiere el uso de

pilas para el recorrido, el recorrido en orden puede hacerse de manera iterativa, por lo tanto no se necesita el uso de la recursividad para realizar los recorridos.

- Árbol enhebrado a la derecha. Su estructura fundamental contienen un puntero hacia la derecha, el cual dirige un nodo antecesor.
- Árbol enhebrado a la izquierda. Éste contiene un puntero hacia la izquierda, el cual dirige a un nodo antecesor en orden.

Las operaciones de recorrido en un árbol binario de búsqueda, implementadas mediante funciones recursivas o con un stack de los nodos a revisar, son generalmente costosas en tiempo de ejecución.

Para lograr recorridos eficientes en un árbol puede modificarse la estructura del nodo, agregando un puntero al padre, o bien añadiendo un par de punteros al sucesor y predecesor, formando de este modo listas doblemente enlazadas.

Una alternativa que demanda menos bits en cada nodo, es emplear un puntero derecho con valor nulo, para apuntar al nodo sucesor de éste, y de manera similar emplear un puntero izquierdo nulo para apuntar a su predecesor; obviamente esto requiere dos bits adicionales por nodo para indicar si el puntero referencia a un nodo hijo o al nodo sucesor o antecesor. Estos árboles se denominan enhebrados (threaded). La idea es utilizar de mejor forma los (n+1) punteros que tienen almacenados valores nulos en un árbol con *n* elementos.

Si sólo interesa acelerar la operación de encontrar al sucesor, se emplean hebras solamente en los punteros derechos de las hojas, para hilvanar los nodos con sus sucesores, dando origen a árboles enhebrados por la derecha (right-threaded tree). Situación que será analizada en esta Unidad.

Los recorridos en árboles enhebrados siguen siendo de costo O(n) pero existirá un considerable ahorro en tiempo al poder diseñar rutinas iterativas.

ACTIVIDAD DE APRENDIZAJE

1.- Realizar una investigación y entregar resumen, sobre el tema arboles enhebrados.

5.5. ÁRBOLES DE BÚSQUEDA

Los árboles de búsqueda son estructuras de datos que soportan las siguientes operaciones de conjuntos dinámicos:

- Search (Búsqueda).
- Minimum (Mínimo).
- Maximum (Máximo).
- Predecessor (Predecesor).
- Insert (Inserción).
- Delete (eliminación).

Los árboles de búsqueda se pueden utilizar así como diccionarios y como colas de prioridad, estas operaciones toman tiempo proporcional a la altura del árbol. Para un árbol completo binario, esto es Θ(lg n) en el peor caso; "sin embargo, si el árbol es una cadena lineal de *n* nodos, las mismas operaciones toman Θ(n) en el peor caso. Para árboles creados aleatoriamente, la altura es O(lg n), con lo cual los tiempos son Θ(lg n).La búsqueda consiste acceder a la raíz del árbol, si el elemento a localizar coincide con éste, la búsqueda ha concluido con éxito, si el elemento es menor, se busca en el subárbol izquierdo, y si es mayor en el derecho. Si se alcanza un nodo hoja y el elemento no ha sido encontrado, se supone que no existe en el árbol. Cabe destacar que la búsqueda en este tipo de árboles es muy eficiente, representa una función logarítmica.[12] El máximo número de comparaciones que necesitaríamos para saber si un elemento se encuentra en un árbol binario de búsqueda, estaría entre [log2(N+1)] y N, siendo N el número de nodos. "La búsqueda de un elemento en un ABB (Árbol Binario de Búsqueda) se puede realizar de dos formas, iterativa o recursiva."[13]

ACTIVIDAD DE APRENDIZAJE

1. Realizar una investigación y entregar un resumen sobre el tema árboles de búsqueda.

[12] *Cfr*. http://droperspace.blogspot.com/
[13] *Cfr*. http://droperspace.blogspot.com/

5.6. PROBLEMAS

A continuación se presentaran ejemplos prácticos sobre el uso de los árboles y su implementación en los lenguajes de programación.

El siguiente código, muestra la estructura de un árbol, presentando los datos en los tres momentos PreOrden, InOrden y PostOrden.[14]

```
class NodoBinario{
        int dato;
        NodoBinario Hizq, Hder;
        //Constructores
        NodoBinario (int Elem){
                dato = Elem;
                NodoBinario Hizq, Hder = null;
        }

        //Insercion de un elemento
        public void InsertaBinario (int Elem){
                if(Elem < dato){
                        if (Hizq == null)
                                Hizq = new NodoBinario(Elem);
                        else
                                Hizq.InsertaBinario(Elem);
                }
                else{
                        if (Elem > dato){
                                if (Hder == null)
                                        Hder = new NodoBinario (Elem);
                                else
                                        Hder.InsertaBinario(Elem);
                        }
                }
        }
```

[14] *Cfr*. http://www.scribd.com/doc/14767010/Arbol1

```
}

//Definicion de la clase Arbol
class Arbol{
	Cola Cola = new Cola();
	NodoBinario Padre;
	NodoBinario Raiz;

	//Constructor
	public Arbol(){
		Raiz = null;
	}

	//Insercion de un elemento en el arbol
	public void InsertaNodo(int Elem){
		if(Raiz == null)
			Raiz = new NodoBinario (Elem);
		else
			Raiz.InsertaBinario (Elem);
	}

	//Preorden Recursivo del arbol
	public void Preorden (NodoBinario Nodo){
		if(Nodo == null)
			return;
		else{
			System.out.print (Nodo.dato + " ");
			Preorden (Nodo.Hizq);
			Preorden (Nodo.Hder);
		}
	}


	//PostOrden recursivo del arbol
```

```
public void PostOrden (NodoBinario Nodo){
        if(Nodo == null)
                return;
        else{
                PostOrden (Nodo.Hizq);
                PostOrden (Nodo.Hder);
                System.out.print (Nodo.dato + " ");
        }
}

//Inorden Recursivo del arbol
public void Inorden (NodoBinario Nodo){
        if(Nodo == null)
                return;
        else{
                Inorden (Nodo.Hizq);
                System.out.print(Nodo.dato + " ");
                Inorden (Nodo.Hder);
        }
}

//Busca un elemento en el arbol
void Busqueda (int Elem, NodoBinario A){
        if((A == null) | (A.dato == Elem)){
                System.out.print(A.dato + " ");
                return;
        }
        else{
                if(Elem>A.dato)
                        Busqueda (Elem, A.Hder);
                else
                        Busqueda ( Elem, A.Hizq);
        }
}
```

```
        //Altura del arbol
        public int Altura (NodoBinario Nodo){
                int Altder = (Nodo.Hder == null? 0:1 + Altura (Nodo.Hder));
                int Altizq = (Nodo.Hizq == null? 0:1 + Altura (Nodo.Hizq));
                return Math.max(Altder,Altizq);
        }

        //Recorrido en anchura del arbol
        public void Anchura (NodoBinario Nodo){
                Cola cola= new Cola();
                NodoBinario T = null;
                System.out.print ("El recorrido en Anchura es: ");
                if(Nodo != null){
                        cola.InsertaFinal (Nodo);
                        while(!(cola.VaciaLista ())){
                                T = cola.PrimerNodo.datos;
                                cola.EliminaInicio();
                                System.out.print(T.dato + " ");
                                if (T.Hizq != null)
                                        cola.InsertaFinal (T.Hizq);
                                if (T.Hder != null)
                                        cola.InsertaFinal (T.Hder);
                        }
                }
                System.out.println();
        }
}

//Definición de la Clase NodoLista
class NodosListaA{
        NodoBinario datos;
    NodosListaA siguiente;
```

```
    //Construtor  Crea un nodo del tipo Object
        NodosListaA (NodoBinario  valor){
                datos =valor;
           siguiente = null;  //siguiente con valor de nulo
         }

        // Constructor Crea un nodo del Tipo Object y al siguiente nodo de la lista
        NodosListaA (NodoBinario valor, NodosListaA signodo){
                datos = valor;
        siguiente = signodo; //siguiente se refiere al siguiente nodo
        }
}

//Definición de la Clase Lista
class Cola{
        NodosListaA PrimerNodo;
        NodosListaA UltimoNodo;
        String Nombre;

        //Constructor construye una lista vacia con un nombre de List
        public Cola(){
                this ("Lista");
        }

        //Constructor
        public Cola (String s){
                Nombre = s;
           PrimerNodo = UltimoNodo =null;
        }

        //Retorna True si Lista Vacía
        public boolean VaciaLista() {
                return PrimerNodo == null;
        }
```

```
        //Inserta un Elemento al Frente de la Lista
        public void InsertaInicio (NodoBinario ElemInser){
                if(VaciaLista())
                PrimerNodo = UltimoNodo = new NodosListaA (ElemInser);
           else
                PrimerNodo = new NodosListaA (ElemInser, PrimerNodo);
        }

        //Inserta al Final de la Lista
        public void InsertaFinal(NodoBinario ElemInser){
                if(VaciaLista())
              PrimerNodo = UltimoNodo = new NodosListaA (ElemInser);
           else
                    UltimoNodo=UltimoNodo.siguiente =new NodosListaA
(ElemInser);
        }

        //Eliminar al Inicio
        public void EliminaInicio(){
                if(VaciaLista())
                System.out.println ("No hay elementos");

                // Restablecer  las referencias de PrimerNodo y UltimoNodo
                if(PrimerNodo.equals (UltimoNodo))
                PrimerNodo = UltimoNodo = null;
           else
                PrimerNodo = PrimerNodo.siguiente;
        }

        //Elimina al final
        public void EliminaFinal (){
           if(VaciaLista())
             System.out.println ("No hay elementos");
```

```
                // Restablecer  las referencias de PrimerNodo y UltimoNodo
                if (PrimerNodo.equals (UltimoNodo))
                PrimerNodo = UltimoNodo = null;
            else{
                NodosListaA Actual =PrimerNodo;
                        while (Actual.siguiente != UltimoNodo)
                                Actual = Actual.siguiente;

                        UltimoNodo =Actual;
                        Actual.siguiente = null;
            }
        }
}


class ArbolBinarioA{
        public static void main (String[]args){
                Arbol A = new Arbol();
                A.InsertaNodo (10);
                A.InsertaNodo (7);
                A.InsertaNodo (8);
                A.InsertaNodo (6);
                A.InsertaNodo (12);
                A.InsertaNodo (11);
                A.InsertaNodo (5);
                A.InsertaNodo (4);
                A.InsertaNodo (3);
                A.InsertaNodo (2);
                System.out.print("El recorrido en Preorden es: ");
                A.Preorden (A.Raiz);
                System.out.println();
                System.out.print("El recorrido en Inorden es: ");
                A.Inorden (A.Raiz);
```

```
                System.out.println();
                System.out.print("El recorrido en Postorden es: ");
                A.PostOrden (A.Raiz);
                System.out.println();
                System.out.println("La altura del arbol es: " + A.Altura (A.Raiz));
                A.Anchura (A.Raiz);
        }
}
```

Ejemplo 2. Árbol de ordenamiento número y colores

```
import java.io.*;

//definicion del arbol roji-negro
class arbolRojiNegro{
        String outm;
        String rota;
        public nodoRojiNegro raiz;
        public static final int rojo=0;
        public static final int negro=1;

        //crea el arbol
        public arbolRojiNegro(){
        }

        //crea el arbol con la llave
        public arbolRojiNegro(int codcu,int codca,String nom,int req){
                raiz = new nodoRojiNegro(codcu,codca,nom,req);
                raiz.color = negro;
        }

        //Inserta una llave en el arbol roji-negro
```

```
        public nodoRojiNegro Insertar(int codcu,int codca,String nom,int req,
nodoRojiNegro t,String out){
                raiz=t;
                try{
                        outm=out;
                        if (estaVacio()){
                                raiz = new nodoRojiNegro(codcu,codca,nom,req);
                                raiz.color = negro;
                                FileWriter fw = new FileWriter (outm+".txt", true);
                                BufferedWriter bw = new BufferedWriter (fw);
                                PrintWriter salida = new PrintWriter (bw);
                                salida.println("Elemento: " + codcu);
                                rota=rota+"Elemento: " +codcu+"\n";
                                salida.close();
                        }
                        else
                                raiz = insertaraux(codcu,codca,nom,req,t);
                }
                catch(Exception e){
                }
                return raiz;
        }

        //Ayuda al metodo de insertar
        public nodoRojiNegro insertaraux(int codcu,int codca,String nom,int req,
nodoRojiNegro t) {
                try{
                        //Insercion normal, y le asigna el padre en otra referencia
                        nodoRojiNegro y=null;
                        nodoRojiNegro x = t;
                        while (x != null){
                        y = x;
                        if (codcu < x.Codcur)
                        x = x.hIzq;
```

```
                    else
                    x = x.hDer;
                    }
                    nodoRojiNegro z = new
nodoRojiNegro(codcu,codca,nom,req,y);
                    if (codcu < y.Codcur)
                        y.hIzq = z;
                    else
                        y.hDer = z;
                    FileWriter fw = new FileWriter (outm+".txt", true);
                    BufferedWriter bw = new BufferedWriter (fw);
                    PrintWriter salida = new PrintWriter (bw);
                    salida.println("Elemento: " + z.Codcur);
                    salida.close();
                    rota=rota+"Elemento: " +z.Codcur+"\n";
                    //inserta en el arbol para arreglarlo
                    t = insertarArreglado(t, z);
                }
                catch(Exception e){
                }
                return t;
        }

        //Recibe la raiz con el elemento metido y el elemento
        //para arreglarlo
        public nodoRojiNegro insertarArreglado(nodoRojiNegro t, nodoRojiNegro
z){
                try{
                    FileWriter fw = new FileWriter (outm+".txt", true);
                    BufferedWriter bw = new BufferedWriter (fw);
                    PrintWriter salida = new PrintWriter (bw);
                    //si el padre z es rojo
                    while ((z.padre != null) && (z.padre.padre != null) &&
(z.padre.color == rojo)){
```

```
                    //si el padre es hijo izquierdo de abuelo
                      if (z.padre == z.padre.padre.hIzq){
                            nodoRojiNegro y = z.padre.padre.hDer;
                        //si el tio de z tambien es rojo
                        if (y!=null && y.color == rojo){
                                    //cambia al tio de z negro, al padre de z
negro
                                    salida.println("Cambio de color");
                                    rota=rota+"Cambio de color\n";
                                    z.padre.color = negro;
                                    y.color = negro;
                                    z.padre.padre.color = rojo;
                                    z = z.padre.padre;
                        }
                        else {
                          //Si z es hijo derecho
                                    if (z.padre.hDer!=null && z ==
z.padre.hDer) {
                                          salida.println("Rotacion
Izquierda");
                                          rota=rota+"Rotacion
Izquierda\n";
                                          z = z.padre;
                                          t = rotarIzq(t, z);
                                    }
                                    else{
                                          salida.println("Rotacion
Derecha");
                                          rota=rota+"Rotacion Derecha\n";
                                          //Caso 3
                                          z.padre.color = negro;
                                          z.padre.padre.color = rojo;
                                          t = rotarDer(t, z.padre.padre);
                                    }
```

```
        }
          }
    //Si el padre de z es hijo derecho
          else{
                  nodoRojiNegro y = z.padre.padre.hIzq;
                  //si el tio de z es rojo
                  if (y!=null && y.color == rojo) {
                    // cambiar colores
                    salida.println("Cambio de color");
                    rota=rota+"Cambio de color\n";
                    z.padre.color = negro;
                    y.color = negro;
                    z.padre.padre.color = rojo;
                    z = z.padre.padre;
                  }
                  else {
                    //si z es hijo izquierdo
                    if (z == z.padre.hIzq) {
                            salida.println("Rotacion Derecha");
                            rota=rota+"Rotacion Derecha\n";
                      z = z.padre;
                      t = rotarDer(t, z);
                    }
                    else{
                            salida.println("Rotacion Izquierda");
                            rota=rota+"Rotacion Izquierda\n";
                            //Caso 3
                            z.padre.color = negro;
                            z.padre.padre.color = rojo;
                            t = rotarIzq(t, z.padre.padre);
                    }
                  }
          }
    }
```

```
                salida.close();
        }
        catch (Exception e){

        }
        t.color = negro;
        return t;
}

//rotacion izquierda
public nodoRojiNegro rotarIzq(nodoRojiNegro t, nodoRojiNegro x) {
        nodoRojiNegro y = x.hDer;
        x.hDer = y.hIzq;
        if (y.hIzq != null)
          y.hIzq.padre = x;
        y.padre = x.padre;
        if (x.padre == null)
          t = y;
        else if (x == x.padre.hIzq)
          x.padre.hIzq = y;
        else
          x.padre.hDer = y;
        y.hIzq = x;
        x.padre = y;
        return t;
}

//rotacion derecha
public nodoRojiNegro rotarDer(nodoRojiNegro t, nodoRojiNegro x) {
        nodoRojiNegro y = x.hIzq;
        x.hIzq = y.hDer;
        if (y.hDer != null)
          y.hDer.padre = x;
        y.padre = x.padre;
```

```
        if (x.padre == null)
          t = y;
        else if (x == x.padre.hIzq)
          x.padre.hIzq = y;
        else
          x.padre.hDer = y;
        y.hDer = x;
        x.padre = y;
        return t;
}

//busca un elemento
public boolean Miembro(int x, nodoRojiNegro r){
        raiz=r;
        boolean si=false;
        nodoRojiNegro temp = raiz;
        while (temp != null && si==false) {
          if(x==temp.Codcur){
                si=true;
          }
          else{
                if (x < temp.Codcur)
                temp = temp.hIzq;
            else
                if(x > temp.Codcur)
                        temp = temp.hDer;
          }
        }
        return si;
}

//retorna true si el arbol esta vacio
public boolean estaVacio(){
        return (raiz == null);
```

```
        }

        //Imprime en inorden
        public void Imprimir(nodoRojiNegro t){
                if(estaVacio())
                  System.out.println("Arbol Vacio");
                else
                  imprimirArbol(t);
        }

        //auxiliar
        public void imprimirArbol(nodoRojiNegro t){
                if(t != null){
            imprimirArbol(t.hIzq);
            if(t.color==1)
                System.out.println(t.Codcur + " negro");
            else
                System.out.println(t.Codcur + " rojo");
            imprimirArbol(t.hDer);
                }
    }
}

//Nodo del arbol rojinegro
class nodoRojiNegro{
        int Codcur;             // la llave del arbol
        int Codcar;
        String Nombre;
        int Requisito;
        nodoRojiNegro padre;    // el padre del nodo
        nodoRojiNegro hIzq;     // Hijo izquierdo
        nodoRojiNegro hDer;     // Hijo derecho
        int color;              // Color
        // Constructores
```

```
        nodoRojiNegro(){
                padre=hIzq=hDer=null;
                color=0;
        }
        nodoRojiNegro (int codcu,int codca,String nom,int req){
                Codcur= codcu;
                Codcar= codca;
                Nombre= nom;
                Requisito= req;
                padre= hIzq = hDer = null;
                color= 0;
        }
        public nodoRojiNegro(int codcu,int codca,String nom,int req,
nodoRojiNegro pa){
                Codcur= codcu;
                Codcar= codca;
                Nombre= nom;
                Requisito= req;
                hIzq= hDer=null;
                padre= pa;
                color= 0;
        }
}

class ArbolRojiNegroA{
        public static void main(String args[]){
                arbolRojiNegro nuevo=new arbolRojiNegro();
                nuevo.Insertar(1,1,"pr",1,nuevo.raiz,"pr");
                nuevo.Insertar(2,1,"pr",1,nuevo.raiz,"pr");
                nuevo.Insertar(20,1,"pr",1,nuevo.raiz,"pr");
                nuevo.Insertar(8,1,"pr",1,nuevo.raiz,"pr");
                nuevo.Insertar(12,1,"pr",1,nuevo.raiz,"pr");
                nuevo.Insertar(4,1,"pr",1,nuevo.raiz,"pr");
                nuevo.Insertar(21,1,"pr",1,nuevo.raiz,"pr");
```

```
            nuevo.Insertar(18,1,"pr",1,nuevo.raiz,"pr");
            nuevo.Insertar(7,1,"pr",1,nuevo.raiz,"pr");
            nuevo.Insertar(52,1,"pr",1,nuevo.raiz,"pr");
            nuevo.Insertar(63,1,"pr",1,nuevo.raiz,"pr");
            nuevo.Insertar(17,1,"pr",1,nuevo.raiz,"pr");
            nuevo.imprimirArbol(nuevo.raiz);

        }
}
```

Ejemplo 3. Árbol ordenado

```
import javax.swing.*;
import java.io.*;

class ArbolHeap{
        // Arreglo en el cual se almacenan los elementos.
        int[] llave;
        // Posición en la cual se va a insertar.
        int posicion;

        //Contructor
        public ArbolHeap(){
                llave = new int[10];
                posicion = 0;
        }

        //Obtiene el hijo derecho según la posicion del padre
        public int hDer(int posPadre){
                return (2 * posPadre) + 1;
        }
```

```
        //Obtiene el hijo izquierdo según la posicion del padre
        public int hIzq(int posPadre){
                return (2 * posPadre);
        }

        //Inserta un elemento dentro del arbol de Heap.
        public void inserta(int nLlave){
                int padre;
                int auxiliar;
                int siguiente;
                siguiente = posicion;
                padre = (siguiente / 2);
                if(padre < 0)
                        padre = 0;
                llave[siguiente] = nLlave;
                // se acomodan los elementos para que el padre sea mayor que
cualquiera de los hijos.
                while((siguiente != 0) && (llave[padre] <= llave[siguiente])){
                        auxiliar = llave[padre];
                        llave[padre] = llave[siguiente];
                        llave[siguiente] = auxiliar;
                        siguiente = padre;
                        padre = (siguiente / 2);
                }
                posicion++;
        }

        //Ordena el árbol de manera que quede en una cola de prioridad.
        public void HeapSort(){
                int padre, hijo, llaveAnterior;
                int ultima = posicion-1;
                for(int i = 10; i >= 1;i--){
                        llaveAnterior = llave[ultima];
                        llave[ultima] = llave[0];
```

```
                ultima = ultima - 1;
                padre = 0;
                if((ultima >= 2) && (llave[2] > llave[1]))
                        hijo = 2;
                else
                        hijo = 1;
                while((hijo <= ultima) && (llave[hijo] > llaveAnterior)){
                        llave[padre] = llave[hijo];
                        padre = hijo;
                        hijo = padre * 2;
                        if(((hijo + 1) <= ultima) && (llave[hijo + 1] >
llave[hijo]))
                                hijo++;
                        this.mostrar();

                }
                llave[padre] = llaveAnterior;
                this.mostrar();
        }
    }

    //Muestra al arreglo
    public void mostrar(){
        try{
        FileWriter fw = new FileWriter ("ArbolHeap.txt", true);
        BufferedWriter bw = new BufferedWriter (fw);
        PrintWriter salida = new PrintWriter (bw);
                int i = 0;
                while( i <= 9){
                        salida.print(llave[i] +" ");
                        i++;
                }
                salida.println("");
                salida.println("------------------------");
```

```
                salida.close();
            }
            catch(Exception e){
            }
        }

        //Muestra sin mandarlo al archivo
        public String muestra(){
            String impresor="";
            int i = 0;
            while( i <= 9){
                if(llave[i]==0){
                    impresor=impresor+"--" +"  ";
                }
                else{
                    impresor=impresor+llave[i] +"  ";
                }
                i++;
            }
            return impresor;
        }
}

class ArbolHeapA{
        public static void main(String args[]){
            ArbolHeap nuevo=new ArbolHeap();
            nuevo.inserta(2);
            nuevo.inserta(7);
            nuevo.inserta(1);
            nuevo.inserta(9);
            nuevo.inserta(16);
            nuevo.inserta(3);
            nuevo.inserta(18);
            nuevo.inserta(10);
```

```
            nuevo.inserta(11);
            nuevo.inserta(22);
            System.out.println("Heap: "+nuevo.muestra());
            nuevo.HeapSort();
            System.out.println("Heap ordenado: "+nuevo.muestra());
        }
}
```

ACTIVIDAD DE APRENDIZAJE

1.- Elaboración de los ejercicios que se encuentran en esta sección, sobre la creación de árboles en los lenguajes de programación.

AUTOEVALUACIÓN

1.- En términos matemáticos, es cualquier conjunto de puntos, llamados vértices, y cualquier conjunto de pares de distintos vértices ( ) Hoja

2.- Es un conjunto finito de uno o más nodos ( ) A.B. Distintos

3.- Se le dice así al nodo sí y solo sí el nodo X es apuntado por Y. También se dice que X es descendiente directo de Y ( ) Árboles Binarios

4.- Se le dice así al nodo sí y solo sí el nodo X apunta a Y. También se dice que X es antecesor de Y. ( ) Árboles multicamino

5.- Se le llama así a aquellos nodos que no tienen ramificaciones (hijos). ( ) Hijo

6.- Es el número de arcos que deben ser recorridos para llegar a un determinado nodo. ( ) Altura

7.- Es el máximo número de niveles de todos los nodos del árbol. ( ) Padre

8.- se utilizan frecuentemente para representar conjuntos de datos cuyos elementos se identifican por una clave única ( ) Árbol

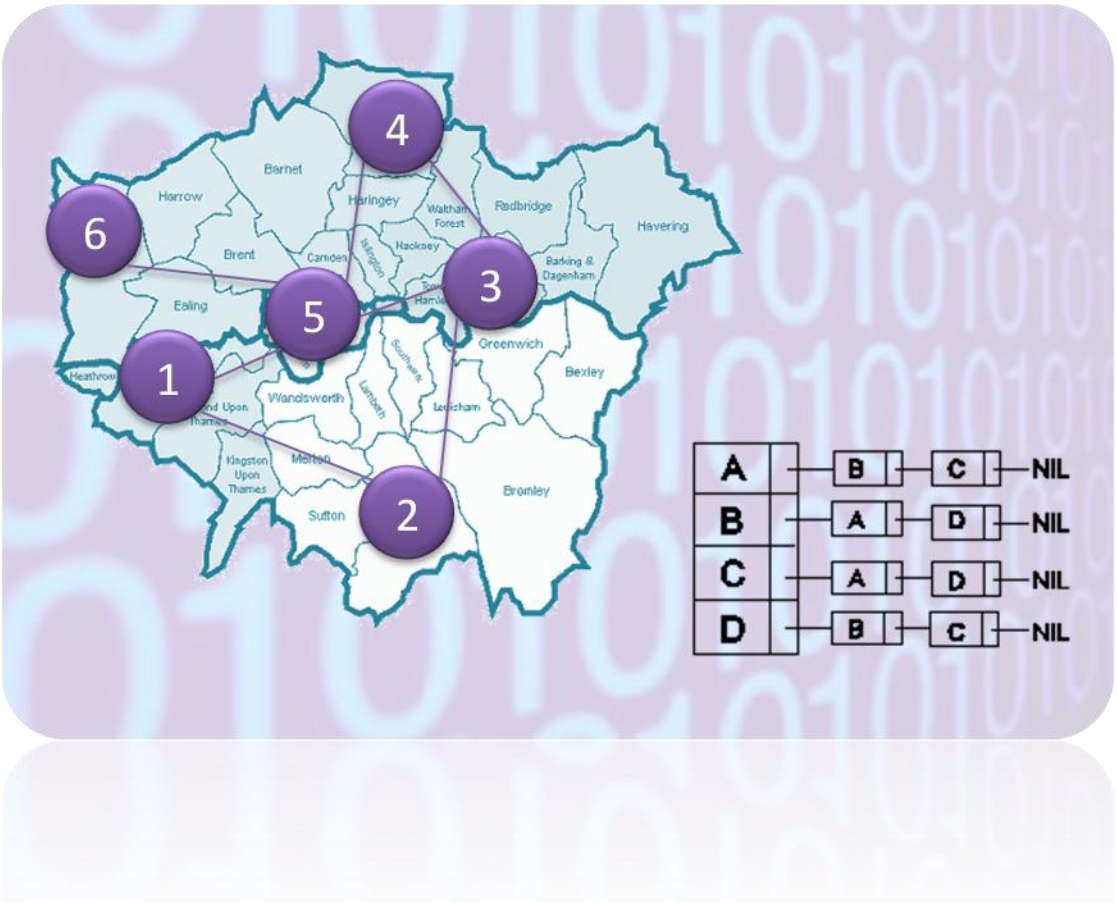
9.- Se dice que dos árboles son llamados así cuando sus estructuras son diferentes. ( ) Nivel

10.- Los árboles de grado superior a 2 reciben ( ) Árbol este nombre.

Respuesta: 5, 9, 8, 10, 3, 7, 4, 2, 6, 1

UNIDAD 6

GRAFOS



OBJETIVO

Comprender los conceptos básicos que rodean el uso de los grafos, su funcionalidad y representación.

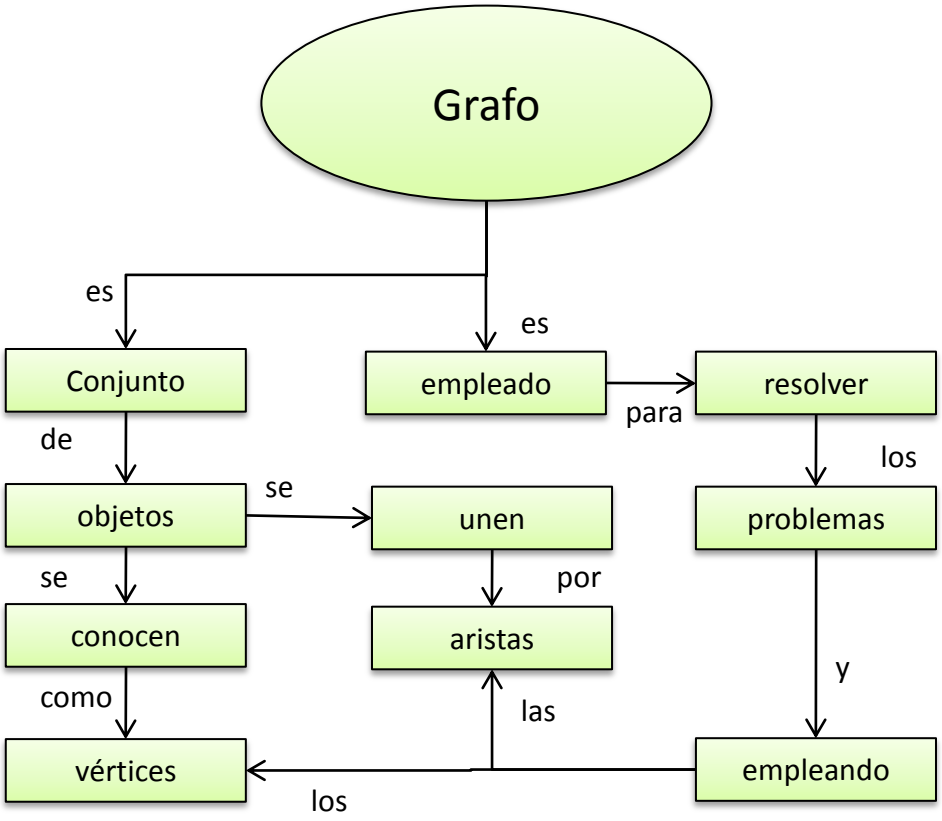
TEMARIO

6.1. TERMINOLOGÍA

6.2. CARACTERÍSTICAS GENERALES

6.3. REPRESENTACIÓN DE UN GRAFO

MAPA CONCEPTUAL



INTRODUCCIÓN

En la actualidad, se pueden observar bastantes cosas que pueden parecernos muy cotidianas: carreteras, líneas telefónicas, de televisión por cable, el transporte colectivo metro, circuitos eléctricos de nuestras casas, automóviles, y muchas otras; lo que no se piensa de modo habitual es que todo esto es parte de algo que, en matemáticas, se denomina como grafos.

En la presente Unidad se explicará qué son los grafos, los tipos, y algunas derivaciones de éstos, así como su representación gráfica y, en ciertos casos, la representación en un programa informático, así como en la memoria.

Se explicará de modo sencillo los conceptos y ciertas metodologías con un lenguaje accesible para un mejor entendimiento.

6.1. TERMINOLOGÍA

Dentro de las matemáticas y el área de ciencias de la computación, un grafo se puede definir como un conjunto de objetos llamados vértices (o nodos) unidos por aristas (o arcos), que permiten representar relaciones binarias entre elementos de un conjunto.

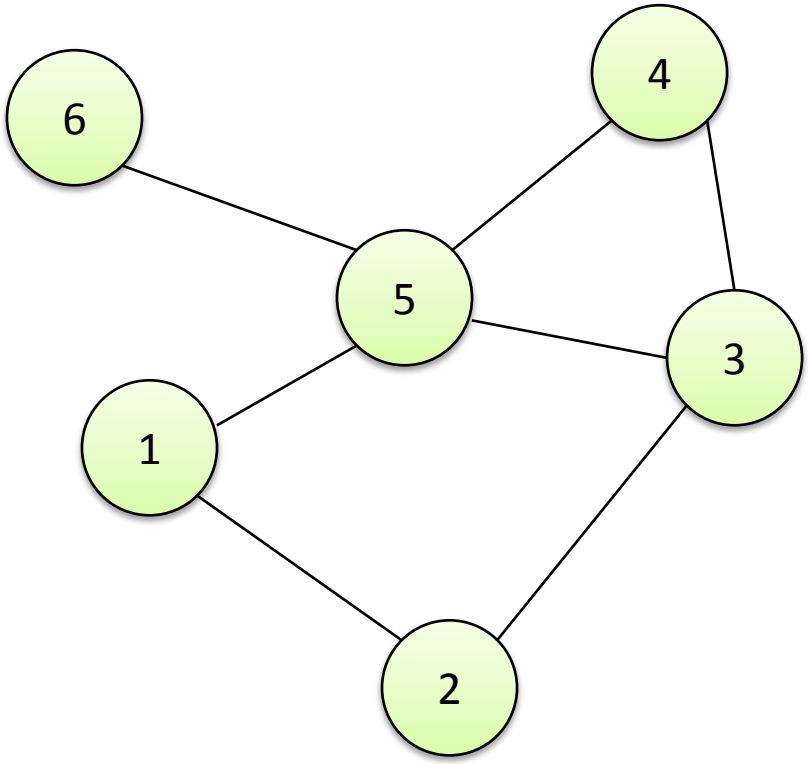


Fig. 6.1. Grafo con 6 vértices y 7 aristas

En algunos casos, los datos contienen relaciones entre ellos que no es necesariamente jerárquica. Dibujar un grafo para resolver un problema es un reflejo muy común, que no precisa conocimientos matemáticos. Por ejemplo, un comerciante de frutas tiene que visitar *n* poblados, conectados entre sí por carreteras, su interés previsible será minimizar la distancia recorrida (o el tiempo, si se pueden prever en atascos).

El grafo correspondiente tendrá como vértices las ciudades, como aristas las carreteras y la valuación será la distancia entre ellas. Otro ejemplo podría ser el de unas líneas aéreas que realizan vuelos entre las ciudades conectadas por líneas.

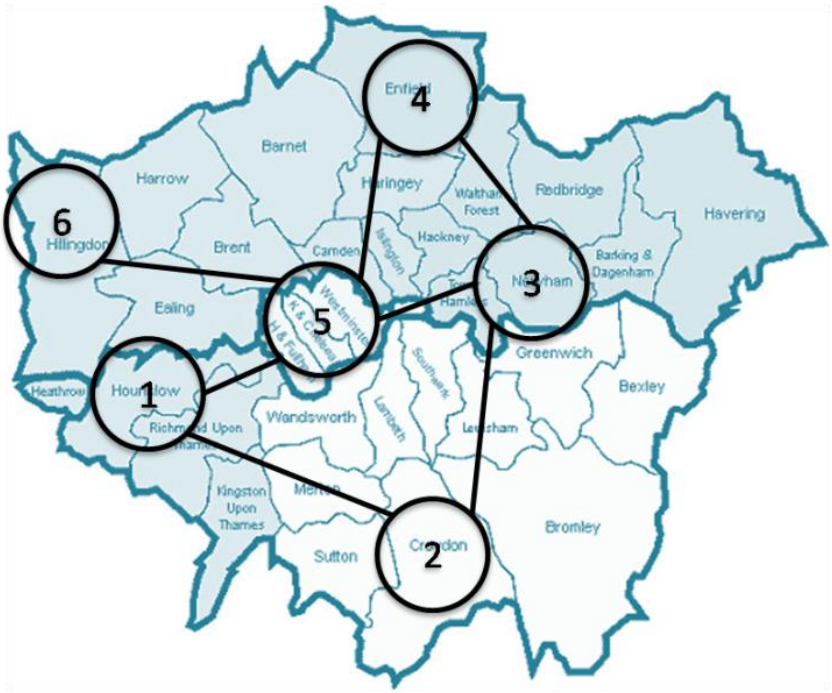


Fig. 6.2. Ejemplo sobre el uso de los grafos

6.2. CARACTERÍSTICAS GENERALES

En la mayoría de los textos de estructura de datos se utilizan los siguientes términos para hablar de los grafos:

- “Camino. Es una secuencia de vértices V1, V2, V3, ... , Vn, tal que cada uno de estos V1-&gtV2, V2-&gtV3, V1-&gtV3.
- Longitud de camino. Es el número de arcos en ese camino.
- Camino simple. Es cuando todos sus vértices, excepto tal vez el primero y el último son distintos.
- Ciclo simple. Es un camino simple de longitud por lo menos de uno que empieza y termina en el mismo vértice.
- Aristas paralelas. Es cuando hay más de una arista con un vértice inicial y uno terminal dados.
- Grafo cíclico. Se dice que un grafo es cíclico cuando contiene por lo menos un ciclo.
- Grafo acíclico. Se dice que un grafo es acíclico cuando no contiene ciclos.
- Grafo conexo. Un grafo G es conexo, si y sólo si existe un camino simple en cualesquiera dos nodos de G.

- Grafo completo o Fuertemente conexo. Un grafo dirigido G es completo si para cada par de nodos (V,W) existe un camino de V a W y de W a V (forzosamente tendrán que cumplirse ambas condiciones), es decir que cada nodo G es adyacente a todos los demás nodos de G.
- Grafo unilateralmente conexo. Un grafo G es unilateralmente conexo si para cada par de nodos (V,W) de G hay un camino de V a W o un camino de W a V.
- Grafo pesado o etiquetado. Un grafo es pesado cuando sus aristas contienen datos (etiquetas). Una etiqueta puede ser un nombre, costo o un valor de cualquier tipo de dato. También a este grafo se le denomina red de actividades, y el número asociado al arco se le denomina factor de peso.
- Vértice adyacente. Un nodo o vértice V es adyacente al nodo W si existe un arco de m a n.
- Grado de salida. El grado de salida de un nodo V de un grafo G, es el número de arcos o aristas que empiezan en V.
- Grado de entrada. El grado de entrada de un nodo V de un grafo G, es el número de aristas que terminan en V.
- Nodo fuente. Se le llama así a los nodos que tienen grado de salida positivo y un grado de entrada nulo.
- Nodo sumidero. Se le llama sumidero al nodo que tiene grado de salida nulo y un grado de entrada positivo."[15]

También un grafo es un par ordenado G = (V,E) donde V es un conjunto de vértices o nodos, y E es un conjunto de arcos o aristas, que relacionan estos nodos. Normalmente V suele ser finito. Muchos resultados importantes sobre grafos no son aplicables para grafos infinitos. Se llama orden de G a su número de vértices, | V |

6.3. REPRESENTACIÓN DE UN GRAFO

"Hay tres maneras de representar un grafo en un programa: mediante matrices, mediante listas y mediante matrices dispersas.

[15] Véase http://boards4.melodysoft.com/app?ID=2005AEDII0405&msg=15&DOC=21

Representación mediante matrices: La forma más sencilla de guardar la información de los nodos es mediante la utilización de un vector que indexe los nodos, de modo que los arcos entre los nodos se pueden ver como relaciones entre los índices. Esta relación entre índices se puede guardar en una matriz, que se denomina de adyacencia."[16]

"Representación mediante listas: En las listas de adyacencia se guarda por cada nodo, además de la información que pueda contener el propio nodo, una lista dinámica con los nodos a los que se puede acceder desde él. La información de los nodos se puede guardar en un vector, al igual que antes, o en otra lista dinámica."[17]

"Representación mediante matrices dispersas: Para evitar uno de los problemas que se tienen con las listas de adyacencia, que es la dificultad de obtener las relaciones inversas, se pueden utilizar las matrices dispersas, que contienen tanta información como las matrices de adyacencia, pero, en principio, no ocupan tanta memoria como las matrices, pues al igual que en las listas de adyacencia, sólo se representan los enlaces que existen en el grafo.

Los grafos se representan en memoria secuencial mediante matrices de adyacencia. Una matriz de adyacencia, es una de dimensión n*n, en donde n es el número de vértices que almacena valores booleanos, donde matriz M[i,j] es verdadero si y sólo si existe un arco que vaya del vértice y al vértice j."[18]

Véase el siguiente grafo dirigido con su matriz de adyacencia, que se obtuvo a partir del grafo

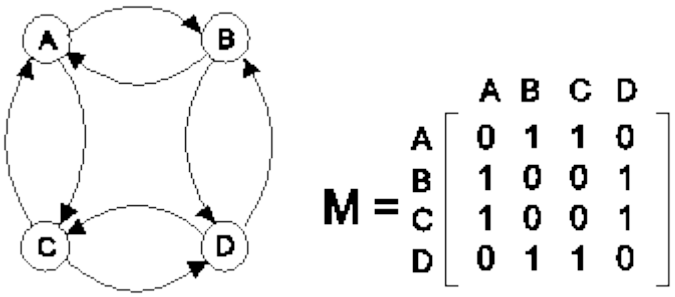


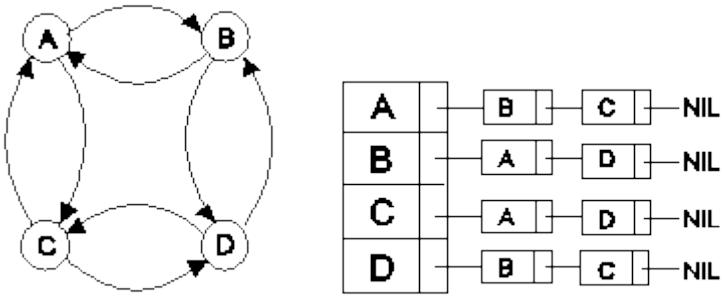
Fig. 6.3. Grafo dirigido con matriz.

"Los grafos se representan en memoria enlazada mediante listas de adyacencia.

[16] *Cfr*. http://www.scribd.com/doc/39500798/GRAFOS
[17] *Cfr*. http://www.scribd.com/doc/39500798/GRAFOS
[18] *Cfr*. http://www.scribd.com/doc/39500798/GRAFOS

Una lista de adyacencia tiene la siguiente definición: Para un vértice i es una lista en cierto orden formada por todos los vértices adyacentes [a,i]. Se puede representar un grafo por medio de un arreglo donde cabeza de i es un apuntador a la lista de adyacencia al vértice i.

Véase el siguiente grafo dirigido con su lista de adyacencia."[19]



ACTIVIDAD DE APRENDIZAJE

1.- Desarrolla una aplicación de forma libre (elegida por el alumno), donde demuestre el uso de los grafos.

[19] *Cfr*. http://www.scribd.com/doc/39500798/GRAFOS

AUTOEVALUACIÓN

1.- Se puede definir como un conjunto de objetos llamados vértices unidos por aristas. ( ) Grafo a cíclico

2.- Es una secuencia de vértices V1, V2, V3, ... , Vn, tal que cada uno de estos V1-&gtV2, V2-&gtV3, V1-&gtV3 ( ) Ciclo simple

3.- Es el número de arcos en ese camino ( ) Grafo cíclico

4.- Es cuando todos sus vértices, excepto tal vez el primero y el último son distintos. ( ) Aristas paralelas

5.- Es un camino simple de longitud por lo menos de uno que empieza y termina en el mismo vértice. ( ) Grafo conexo

6.- Es cuando hay más de una arista con un vértice inicial y uno terminal dados. ( ) Grafo

7.- Se dice que un grafo de este tipo cuando contiene por lo menos un ciclo. ( ) Camino simple

8.- Se dice que un grafo de este tipo cuando no contiene ciclos. ( ) Camino

9.- Un grafo G es conexo, si y solo si existe un camino simple en cualesquiera dos nodos de G. ( ) Longitud de camino

Respuesta.- 8, 5, 7, 6, 9, 1, 4, 2, 3

BIBLIOGRAFÍA

- Adam Drozdek, *Estructura de datos y algoritmos en java*, México, Thomson Learning, 2000.
- Alfred V. Aho / Jonh E. Hopcroft, *Estructura de datos y algoritmos*, Madrid, Pearson Educación, 1983.
- Antonio Garrido / Joaquin Fernández, *Abstracción y estructuras de datos en c++*, Madrid, Delta publicaciones 2006.
- Goodrich / Tamassia, *Estructura de datos y algoritmos en java*, México, CECSA, 2002.
- Harvey M. y Paul J. Deitel, *Como programar en java*, México, Deitel, 2004.
- Narciso Martí, *Estructura de datos y métodos algorítmicos*, Madrid, McGraw Hill, 2003.
- Roman Martinez/Elda Quiriga, *Estructura de datos: referencia practicas*, México, Thomson Learning, 2001.
- Steven Holzner, *La biblia de java 2*, España, Anaya Ilustrada, 2000.
- Osvaldo Cairó /Silvia Guardati, *Estructuras de datos*, México, McGraw Hill, 2006.

GLOSARIO[20]

Altura. Es el máximo número de niveles de todos los nodos del árbol.

Árbol.- Estructura jerárquica aplicada sobre una colección de objetos llamados nodos, en la que uno de ellos se conoce como nodo raíz y cuya relación entre nodos se identifican como padre-hijo, hermano, etc.

Árbol Abarcador.- Es un árbol libre que conecta todos los vértices de V.

Árbol balanceado.- Conocido también como árbol AVL, es un árbol binario de búsqueda en la cual, para todo nodo se árbol, la altura de los sub árboles izquierdo y derecho no debe diferir en más de una unidad.

Árbol de multicamino.- Árbol en el que cada nodo puede tener más de dos descendientes directos y cuyas ramas están ordenadas.

Arreglo.- Colección finita, homogénea y ordenada de elementos.

Arreglo de N dimensiones.- Aquel en el cual cada uno de sus elementos debe identificarse por n índices que marque su posición exacta dentro del arreglo.

Arreglo paralelo.- Estructura formada por dos o más arreglos, cuyos elementos se corresponden, por lo general en relación de uno a uno.

Búsqueda.- Operación que permite recuperar datos previamente almacenados.

Búsqueda externa.- Aquella en la que todos los datos se encuentran en archivos residentes en dispositivos de almacenamiento.

Búsqueda interna.- La que se realiza con los datos residentes en la memoria principal de la computadora.

Camino.- Un camino P de longitud n desde un vértice v a un vértice w se define como la secuencia de n vértices que se deben seguir para llegar del nodo origen al nodo destino.

class.- Definición de una clase.

Colisión.- La que se origina al utilizar una función hash, cuando dos elementos tienen la misma dirección en memoria.

Conjunto.- Es un dato estructurado integrado por un grupo de objetos del mismo tipo.

[20] Creado con información de alguno de los siguientes sitios: http://231mequipo2.blogspot.com/ o http://nancynohemi.webuda.com/index.php?option=com_content&view=article&id=40&Itemid=41 o http://hellfredmanson.over-blog.es/article-30369340-6.html

Dato Estructurado.- Esta formado por varios componentes, cada uno de los cuales puede ser a su vez un dato estructurado.

Dato simple.- Aquel que hace referencia a un único valor a la vez y que ocupa una casilla en memoria.

Estructura dinámica de datos.- Aquella que permite la asignación de espacio en memoria durante la ejecución de un programa, conforme lo requieren las variables.

Fifo.- Iníciales en ingles de Firt In, Firt Out, el primero que entra es el primero en salir.

for.- Permite la construcción de ciclos.

Grado. Es el número de descendientes directos de un determinado nodo.

Grado del árbol es el máximo grado de todos los nodos del árbol.

Gráficas.- Estructura de datos que permite representar diferente tipo de relaciones entre los objetos.

Grafica completa.- Se dice que una gráfica es completa si cada vértice de v de G es adyacente a todos los demás vértices de G.

Gráfica conexa.- Se dice que una gráfica es conexa si existe un camino simple entre dos de sus nodos cualesquiera.

Gráfica dirigida.- Se caracteriza por que sus aristas tienen asociada una dirección.

Gráficas no dirigidas.- Su característica principal es que sis aristas son pares no ordenados de vértices.

Hermano. Dos nodos serán hermanos si son descendientes directos de un mismo nodo.

Hijo.- X es hijo de Y, sí y solo sí el nodo X es apuntado por Y. También se dice que X es descendiente directo de Y.

Hoja. Se le llama hoja o terminal a aquellos nodos que no tienen ramificaciones (hijos).

int.- Definición de un objeto tipo Entero.

JOptionPane.- Superclase que contienen cuadros de mensajes de entrada y salida.

Lifo.- Iníciales de la expresión en ingles Last In, Firt Out, último en entrar primero en salir.

Lista.- Colección de elementos llamados nodo, cuyo orden se establece por medio de punteros.

Lista circular.- Aquella en la que su último elemento apunta al primero.

Lista doblemente ligada.- Colección de elementos llamados nodos, en la cual cada nodo tiene dos punteros, uno apuntando al nodo sucesor y otro al predecesor.

Lista invertida.- Lista que contiene las claves de los elementos que posee un determinado atributo.

Matriz.- Estructura de datos que permite organizar la información en renglones y columnas.

Métodos de búsqueda.- Se caracteriza por el orden en el cual se expande los nodos.

Multilista.- Estructura que permite almacenar en una o varias listas, las direcciones de los elementos que poseen uno o más atributos específicos, la cual facilita la búsqueda.

Nivel. Es el número de arcos que deben ser recorridos para llegar a un determinado nodo. Por definición la raíz tiene nivel 1.

Nodo interior. Es un nodo que no es raíz ni terminal.

Notación infija.- Se dice que una expresión aritmética tienen notaciones infijas cuando sus operadores están entre los operandos, por ejemplo A+B.

Notación postfija.- Se dice que una expresión aritmética tienen notaciones infijas cuando sus operadores están al final de los operandos, por ejemplo AB+.

Notación prefija.- Se dice que una expresión aritmética tienen notaciones infijas cuando sus operadores están al inicio de los operandos, por ejemplo +AB.

Ordenación Externa.- En esta forma de ordenación los datos se toman de archivos residentes en dispositivos de almacenamiento secundarios, tales como discos, cintas, etc.

Ordenación interna.- Se aplica en la ordenación de arreglos y se realiza con todos los datos de estos alojados en la memoria principal de la computadora.

Ordenar.- Organizar un conjunto de datos y objetos en una secuencia especifica, por lo general ascendente o descendente.

Padre. X es padre de Y sí y solo sí el nodo X apunta a Y. También se dice que X es antecesor de Y.

Peso. Es el número de nodos del árbol sin contar la raíz.

Pila.- Lista de elementos a la cual se puede insertar o eliminar elementos sólo por uno de sus extremos.

public .- Definición de un objeto de tipo público, que puede ser empleado en cualquier parte del código del programa.

Puntero.- Dato que almacena una dirección de memoria, en donde se almacena una variable.

Recursión.- Herramienta de programación que permite definir un objeto en términos de él mismo.

Registro.- Es un dato estructurado en el cual sus componentes pueden ser de diferentes tipos, incluso registro o arreglos.

String.- Definición de un objeto tipo Cadena.