Universidad Nacional del Santa

Facultad de Ingeniería

EAP de Ingeniería de Sistemas e Informática

MANUAL de ALGORITMOS Y ESTRUCTURA DE DATOS

IV CICLO

- - - Uso Interno - - -

Ing. Hugo Caselli Gismondi

Chimbote

7ma. Edición-2009

INDICE

INTRODUCCION

El presente documento tiene como objetivo fundamental servir como guía didáctica para la asignatura de Algoritmos y Estructuras de Datos para los alumnos del IV ciclo de la Escuela Académico Profesional de Ingeniería de Sistemas e Informática de la Universidad Nacional del Santa.

El contenido de cada entregable corresponde al silabus oficial del curso, y está dividido de acuerdo a los temas que se consideran fundamentales para los objetivos académicos propios para el conocimiento del Profesional de Ingeniería de Sistemas. Como son los algoritmos elementales, que forman parte de los grandes Sistemas de Información, dentro de estos sistemas es necesario hacer uso eficiente de estructuras de datos, que de acuerdo a ciertas necesidades son de gran utilidad, pues hacen que el trabajo del programador sea más fácil al momento de implementar alguna rutina dentro de cualquier sistema de información, haciéndolo con la soltura que da el conocimiento previo, sin tener que reinventar la pólvora, pues en la mayor parte de los casos, estos algoritmos y estructuras de datos serán de uso y aplicación común de ahora en adelante.

Es necesario anotar que por si solos, cada documento entregable no es suficiente para su total comprensión, sino que tiene que ir acompañada de una explicación detallada brindada por el profesor del curso.

SEMANA 1: Algoritmos, Estructura de Datos, TAD

1. ALGORTIMOS

¿Qué es un algoritmo?

Es la secuencia de pasos (método) que nos permite resolver un problema determinado

Además debe cumplir estas condiciones:

- *Finitud*: el algoritmo debe acabar tras un número finito de pasos. Es más, es casi fundamental que sea en un número razonable de pasos.
- *Definibilidad*: el algoritmo debe definirse de forma precisa para cada paso, es decir, hay que evitar toda ambigüedad al definir cada paso. Puesto que el lenguaje humano es impreciso, los algoritmos se expresan mediante un lenguaje formal, ya sea matemático o de programación para un computador.
- *Entrada*: el algoritmo tendrá cero o más entradas, es decir, cantidades dadas antes de empezar el algoritmo. Estas cantidades pertenecen además a conjuntos especificados de objetos. Por ejemplo, pueden ser cadenas de caracteres, enteros, naturales, fraccionarios, etc. Se trata siempre de cantidades representativas del mundo real expresadas de tal forma que sean aptas para su interpretación por el computador.
- *Salida*: el algoritmo tiene una o más salidas, en relación con las entradas.
- *Efectividad*: se entiende por esto que una persona sea capaz de realizar el algoritmo de modo exacto y sin ayuda de una máquina en un lapso de tiempo finito.

A menudo los algoritmos requieren una organización bastante compleja de los datos, y es por tanto necesario un estudio previo de las estructuras de datos fundamentales. Dichas estructuras pueden implementarse de diferentes maneras, y es más, existen algoritmos para implementar dichas estructuras. El uso de estructuras de datos adecuadas pueden hacer trivial el diseño de un algoritmo, o un algoritmo muy complejo puede usar estructuras de datos muy simples.

Uno de los algoritmos más antiguos conocidos es el algoritmo del matemático griego Euclides. El término algoritmo proviene del matemático Mohammed ibn Musa al-Khowarizmi, matemático Persa (actual Irán), que vivió aproximadamente entre los años 780 y 850 d.C. (Siglo IX) El describió la realización de operaciones elementales en el sistema de numeración decimal. De al-Khwarizmi se obtuvo la derivación algoritmo.

Clasificación de algoritmos

- *Algoritmo determinista*: en cada paso del algoritmo se determina de forma única el siguiente paso.
- *Algoritmo no determinista*: deben decidir en cada paso de la ejecución entre varias alternativas y agotarlas todas antes de encontrar la solución.

Todo algoritmo tiene una serie de características, entre otras que requiere una serie de recursos, algo que es fundamental considerar a la hora de implementarlos en una máquina. Estos recursos son principalmente:

- *El tiempo*: período transcurrido entre el inicio y la finalización del algoritmo.
- *La memoria*: la cantidad (la medida varía según la máquina) que necesita el algoritmo para su ejecución.

Obviamente, la capacidad y el diseño de la máquina pueden afectar al diseño del algoritmo.

En general, la mayoría de los problemas tienen un parámetro de entrada que es el número de datos que hay que tratar, esto es: N. La cantidad de recursos del algoritmo es tratada como una función de N. De esta manera puede establecerse un tiempo de ejecución del algoritmo que suele ser proporcional a una de las siguientes funciones:

Tiempo de ejecución constante. Significa que la mayoría de las instrucciones se ejecutan una vez o muy pocas.

- **N**: Tiempo de ejecución lineal. Un caso en el que N valga 40, tardará el doble que otro en que N valga 20. Un ejemplo sería un algoritmo que lee N números enteros y devuelve la media aritmética.
- **N^2**: Tiempo de ejecución cuadrático. Suele ser habitual cuando se tratan pares de elementos de datos, como por ejemplo un bucle anidado doble. Si N se duplica, el tiempo de ejecución aumenta cuatro veces. El peor caso de entrada del algoritmo Quick Sort se ejecuta en este tiempo.
- **logN**: Tiempo de ejecución logarítmico. Se puede considerar como una gran constante. La base del logaritmo (en informática la más común es la base 2) cambia la constante, pero no demasiado. El programa es más lento cuanto más crezca N, pero es inapreciable, pues logN no se duplica hasta que N llegue a N2.
- **N·logN**: El tiempo de ejecución es N·logN. Es común encontrarlo en algoritmos como Quick Sort y otros del estilo divide y vencerás. Si N se duplica, el tiempo de ejecución es ligeramente mayor del doble.
- **N^3**: Tiempo de ejecución cúbico. Como ejemplo se puede dar el de un bucle anidado triple. Si N se duplica, el tiempo de ejecución se multiplica por ocho.
- **2^N**: Tiempo de ejecución exponencial. No suelen ser muy útiles en la práctica por el elevadísimo tiempo de ejecución. El problema de la mochila resuelto por un algoritmo de fuerza bruta -simple vuelta atrás- es un ejemplo. Si N se duplica, el tiempo de ejecución se eleva al cuadrado.

- **Algoritmos polinomiales**: aquellos que son proporcionales a N^k. Son en general factibles.
- **Algoritmos exponenciales**: aquellos que son proporcionales a k^N. En general son infactibles salvo un tamaño de entrada muy reducido.

Notación O-grande

En general, el tiempo de ejecución es proporcional, esto es, multiplica por una constante a alguno de los tiempos de ejecución anteriormente propuestos, además de la suma de algunos términos más pequeños. Así, un algoritmo cuyo tiempo de ejecución sea $T = 3N^2 + 6N$ se puede considerar proporcional a N^2. En este caso se diría que el algoritmo es del orden de N^2, y se escribe $O(N^2)$

Los grafos definidos por matriz de adyacencia ocupan un espacio $O(N^2)$, siendo N el número de vértices de éste.

La notación O-grande ignora los factores constantes, es decir, ignora si se hace una mejor o peor implementación del algoritmo, además de ser independiente de los datos de entrada del algoritmo. Es decir, la utilidad de aplicar esta notación a un algoritmo es encontrar un límite superior del tiempo de ejecución, es decir, el peor caso.

A veces ocurre que no hay que prestar demasiada atención a esto. Conviene diferenciar entre el peor caso y el esperado. Por ejemplo, el tiempo de ejecución del algoritmo Quick Sort es de $O(N^2)$. Sin embargo, en la práctica este caso no se da casi nunca y la mayoría de los casos son proporcionales a N·logN. Es por ello que se utiliza esta última expresión para este método de ordenación.

Una definición rigurosa de esta notación es la siguiente:

Una función g(N) pertenece a O(f(N)) si y sólo si existen las constantes c_0 y N_0 tales que:

$|g(N)| <= |c_0 \cdot f(N)|$, para todo $N >= N_0$.

Clasificación de problemas

Los problemas matemáticos se pueden dividir en primera instancia en dos grupos:

- ***Problemas indecidibles***: aquellos que no se pueden resolver mediante un algoritmo.
- ***Problemas decidibles***: aquellos que cuentan al menos con un algoritmo para su cómputo.

Sin embargo, que un problema sea decidible no implica que se pueda encontrar su solución, pues muchos problemas que disponen de algoritmos para su resolución son inabordables para un computador por el elevado número de operaciones que hay que realizar para resolverlos. Esto permite separar los problemas decidibles en dos:

- Intratables: aquellos para los que no es factible obtener su solución.
- Tratables: aquellos para los que existe al menos un algoritmo capaz de resolverlo en un tiempo razonable.

Los problemas pueden clasificarse también atendiendo a su complejidad. Aquellos problemas para los que se conoce un algoritmo polinómico que los resuelve se denominan clase P. Los algoritmos que los resuelven son deterministas. Para otros problemas, sus mejores algoritmos conocidos son no deterministas. Esta clase de problemas se denomina clase NP. Por tanto, los problemas de la clase P son un subconjunto de los de la clase NP, pues sólo cuentan con una alternativa en cada paso.

2. ESTRUCTURAS DE DATOS

Es una colección de datos que pueden ser caracterizados por su organización y las operaciones que se definen de ella. Lo que se pretende con las estructuras de datos es facilitar un esquema lógico para manipular los datos en función del problema que haya que tratar y el algoritmo para resolverlo. En algunos casos la dificultad para resolver un problema radica en escoger la estructura de datos adecuada. Y, en general, la elección del algoritmo y de las estructuras de datos que manipulará estará muy relacionada.

Según su comportamiento durante la ejecución del programa distinguimos estructuras de datos:

- **Estáticas:** su tamaño en memoria es fijo. Ejemplo: arrays, conjuntos, cadenas.
- **Dinámicas:** su tamaño en memoria es variable. Ejemplo: listas (Pilas, colas), listas enlazadas con punteros, árboles, grafos, etc.

  A su vez este tipo de estructura se subdivide en
  - ✓ ***Lineales***.- son aquellas estructuras donde los datos se almacenan en zonas contiguas (sucesivas o adyacentes), una detrás de otras. Ejemplo: Listas enlazadas, Pilas, Colas.
  - ✓ ***No lineales***.- Aquí cada elemento puede tener diferentes "siguientes" elementos, se introduce el concepto de bifurcación, ya no hay linealidad. Ejemplo: Árboles, grafos.

Las estructuras de datos que trataremos aquí son los arrays, las pilas y las colas, los árboles, los grafos y algunas variantes de estas estructuras.

3. TIPOS ABSTRACTOS DE DATOS

a. Abstracción

La abstracción es la separación de las propiedades esenciales de aquellas que no lo son. Para resolver un problema real usualmente identificamos las características más relevantes del problema y a partir de ellas construimos una abstracción -un modelo- del problema, que sea manejable y nos permita obtener una solución.

En el contexto del diseño de programas también podemos utilizar la abstracción. Esto significa especificar la funcionalidad del programa en

términos de "alto nivel ". Una vez que se demuestra que esta especificación es correcta se pueden añadir más detalles en pasos o niveles sucesivos hasta el punto en que se obtiene una descripción detallada de "bajo nivel" del programa. Dicha descripción se puede implementar directamente empleando algún lenguaje de programación. El diseñador solo ve los detalles relevantes a un nivel particular del diseño.
Conforme avanza el proceso de diseño va surgiendo la necesidad de distintos tipos de datos así como de las operaciones que deben ser ejecutadas sobre ellos. Para tal fin, se emplea un tipo especial de abstracción conocido como abstracción de datos.

Ventajas de la abstracción en el diseño de programas:
- Limita la complejidad en cada paso del diseño de un programa
- Permite que el diseñador se concentre en los aspectos esenciales del diseño sin preocuparse de los detalles de la implementación.

b. Abstracción de datos y tipo abstracto de datos.

La abstracción de datos es una descripción de los datos requeridos por la aplicación y de las operaciones que deben ejecutarse sobre ellos, sin tomar en cuenta los detalles de representación de los datos ni de la implementación de las operaciones.
Por ejemplo, si un programa requiere el empleo de variables enteras, el programador no necesita saber cómo están almacenadas las variables enteras ni tampoco la manera en que están implementadas las operaciones sobre valores enteros.
Es claro entonces que en el proceso de diseño de los programas se debe emplear la abstracción de los tipos de datos o sea los tipos abstractos de datos.
Un tipo abstracto de datos (TAD) es un modelo matemático de los objetos de datos que constituyen un tipo de datos, así como de las operaciones que se ejecutan sobre dichos objetos.

Ventajas de la abstracción de datos:
- Permite al diseñador concentrarse en el uso de los datos en su aplicación sin preocuparse de los detalles de la representación en memoria de esos datos.
- Permite un diseño del programa que es independiente de cualquier representación específica de los datos que requiere la aplicación, de tal modo que cualquiera sea la representación escogida de los datos ésta pueda ser fácilmente modificada.

Ejemplos de TAD:
a) TAD CONJUNTO: colección de elementos que son manipulados por operaciones como la unión, intersección y la diferencia de conjuntos.
b) TAD ENTERO: es el conjunto de los enteros o sea {-1, -2, -3,…,-¥} È {1, 2, 3, …, ¥} y las operaciones de suma, resta , multiplicación y división sobre los números enteros.
c) TAD COMPLEJO: es la colección formada por todos los números complejos o sea elementos de la forma a + ib, donde a y b son reales, i es la constante imaginaria Ö-1. La operaciones permitidas son la suma, resta, multiplicación, división y conjugado de números complejos.

Notas: El TAD que incluye además a la operación producto cartesiano es diferente al TAD conjunto arriba especificado. El TAD que incluya a la

operación que calcula el resto de la división entera es un TAD diferente al especificado en b).

c. Tipo de datos

Un tipo de datos es la implementación de un TAD.

En un tipo de datos el modelo matemático que define a algún TAD es implementado de acuerdo a la sintaxis de algún lenguaje de programación.

Esta implementación o traducción del TAD consta de las declaraciones de las variables apropiadas y un procedimiento o función de acceso que implemente cada operación requerida por el TAD.

Ejemplos de tipos de datos:

- Tipo de datos integer en el lenguaje de programación Pascal es una implementación del TAD entero.
- Los tipos de datos int y long son dos implementaciones diferentes del TAD entero en C++.

Los tipos de datos anteriores son ejemplos de tipos de datos predefinidos o suministrados por el lenguaje de programación. Por lo general, el diseño de un programa necesitará de tipos de datos diferentes a los predefinidos, a los cuales llamaremos tipos de datos definidos por el usuario.

d. Encapsulamiento de datos

El programa de aplicación debe referirse a las propiedades esenciales de los datos y no a su representación en memoria. Es decir, la comunicación entre el programa de aplicación y la implementación de un TAD solo debe producirse a través de una interfaz, la cual está constituida por las funciones de acceso del TAD. Este agrupamiento de los datos y de las operaciones definidas sobre ellos formando una sola unidad y el ocultamiento de los detalles de la implementación se llama encapsulamiento de datos.

Ventajas del encapsulamiento de datos:

- Un tipo de datos debidamente encapsulado puede ser modificado sin afectar las aplicaciones que lo utilicen.
- Un tipo de datos correctamente encapsulado permite su reutilización en otras aplicaciones que la requieran sin necesidad de conocer los detalles de implementación del tipo de datos.

PROGRAMACION ORIENTADA A OBJETOS

1. Introducción

La programación orientada a objetos surgió en respuesta a la incapacidad de las otras técnicas de programación (programación procedural, programación modular o estructurada) para el desarrollo adecuado de programas de software largos y complejos. Podemos citar algunos problemas:

- Los programas procedurales se organizaban en torno a las funciones o procedimientos, los cuales definían los datos a emplear. En términos simples, los datos que eran requeridos por varias funciones deberían ser datos globales, lo que posibilitaba su modificación inadvertida por otras funciones. Esta situación causó frecuentes errores de programación.
- Los lenguajes procedurales (C, Pascal, Basic etc.) no eran suficientemente adecuados para el modelamiento del mundo real.

2. Objetivos de POO

a) Facilitar el desarrollo, extensión y mejoramiento de los programas de computadora.
b) Reutilización de las componentes de software.

3. Conceptos fundamentales

a. Clases y objetos

Una clase es la implementación de un TAD utilizando un lenguaje de programación orientado a objetos. Equivalentemente, un TAD es una clase sin tomar en cuenta los detalles de implementación. Una clase define conjuntamente a los elementos de datos y a las funciones de acceso de la implementación de un TAD.

Un programa orientado a objetos es una colección de clases. El cómputo en un programa OO se centra en la manipulación de objetos de ciertas clases. La clase es el modelo para crear objetos, en otras palabras, los objetos son las instancias de la clase, los ejemplares individuales de la clase. En un programa OO los objetos interactúan entre ellos mediante el envío de mensajes en que solicitan la ejecución de determinadas acciones.

Cada objeto contiene su propio conjunto de datos, llamados atributos (o variable miembro). Estas variables solo pueden ser accedidas a través de las funciones de acceso llamados métodos (o funciones miembro), las cuales son definidas por la clase. Estas funciones de acceso son compartidas por todos los objetos de la clase, mientras que cada objeto posee una copia individual de las variables miembro.

En resumen, los objetos son "cajas negras" que envían y reciben mensajes. Este enfoque agiliza el desarrollo de nuevo software y facilita el mantenimiento y reusabilidad del software existente.

b. Encapsulamiento de datos

El objeto es una unidad atómica que el usuario no puede partir. En forma conjunta el principio de la atomicidad y el ocultamiento de la información se conoce con el nombre de encapsulamiento de datos.

Este concepto es quizás el concepto más importante del enfoque orientado a objetos. Su importancia radica en que limita los efectos de los cambios colocando una muralla alrededor de cada pieza de dato facilitando de este modo el objetivo a) de la POO.

El usuario no puede manipular directamente las partes del objeto sino a través de las funciones de acceso suministradas con el objeto las cuales en conjunto se denominan la interfase del objeto. De este modo, el usuario no tiene que preocuparse de los detalles de cada operación.

c. Herencia

Tal como los ingenieros electrónicos reusan las componentes de hardware en sus diseños de hardware, los programadores debieran reusar las componentes de software, deben reusar las clases existentes en vez de reimplementarlas. El concepto de la herencia permite lograr el objetivo de reusabilidad de la POO.

Si un objeto no es exactamente lo que se necesita, la herramienta de la herencia permite añadir funcionalidad a los objetos ya creados, definiendo nuevas clases, las clases derivadas, a partir de las clases existentes. Una clase derivada hereda las variables miembros y las funciones miembros de la clase base. La clase derivada puede añadir nuevos miembros y redefinir cualquier función miembro.

Ejemplos:

1) De la clase Figura Geométrica se pueden derivar las clases Círculo, Rectángulo, Cuadrado. (¿Por qué?)

2) De la clase Empleado se puede derivar la clase Secretaria en la cual se podría añadir el atributo Velocidad de Tipeo.

d. Polimorfismo

Es la propiedad que indica la posibilidad de que una entidad tome diferentes formas. El polimorfismo permite referirse a objetos de diferentes clases mediante el mismo elemento de programa y realizar la misma operación de diferentes formas de acuerdo al tipo de objeto sobre el que actúa en ese momento.

***Ejemplo*:**

Supongamos que la clase Figura Geométrica incluyera la operación Cálculo de Área, entonces dicha operación se ejecuta de manera diferente si el objeto que recibe el mensaje es de la clase Círculo, Rectángulo ó Cuadrado.

SEMANA 2: Algoritmos de Ordenamiento y Búsqueda

2.1 ¿Qué es ordenamiento?

Es la operación de organizar datos en algún orden secuencial de acuerdo a un criterio que puede ser creciente o decreciente para los números o ascendente o descendente (alfabéticamente) para datos de caracteres. El propósito principal de un ordenamiento es el de facilitar las búsquedas de los miembros del conjunto ordenado.

Ejemplos de ordenamientos: Directorios telefónicos, índices de contenidos, bibliotecas y diccionarios, etc.

¿Cuándo conviene usar un método de ordenamiento?

Cuando se requiere hacer una cantidad considerable de búsquedas y es importante el factor tiempo.

2.2 Tipos de ordenamientos

Los 2 tipos de ordenamientos que se pueden realizar son: los internos y los externos.

a. Los internos

Son aquellos en los que los valores a ordenar están en memoria principal, por lo que se asume que el tiempo que se requiere para acceder cualquier elemento sea el mismo (a[1], a[500], etc.). Se agrupan de la siguiente manera:

- *Algoritmos de inserción*

  En este tipo de algoritmo los elementos que van a ser ordenados son considerados uno a la vez. Cada elemento es INSERTADO en la posición apropiada con respecto al resto de los elementos ya ordenados.
  - Inserción directa.
  - Inserción binaria.
  - Shell.
  - Hashing
- *Algoritmos de intercambio*

  En este tipo de algoritmos se toman los elementos de dos en dos, se comparan y se INTERCAMBIAN si no están en el orden adecuado. Este proceso se repite hasta que se ha analizado todo el conjunto de elementos y ya no hay intercambios.
  - Burbuja.
  - Quick sort.
- *Algoritmos de selección*

  En este tipo de algoritmos se SELECCIONA o se busca el elemento más pequeño (o más grande) de todo el conjunto de elementos y se coloca en su posición adecuada. Este proceso se repite para el resto de los elementos hasta que todos son analizados.
  - Selección directa.

b. Los externos

Son aquellos en los que los valores a ordenar están en memoria secundaria (disco, cinta, cilindro magnético, etc.), por lo que se asume que el tiempo que se requiere para acceder a cualquier elemento depende de la última posición accesada (posición 1, posición 500, etc.).

- Straight merging.
- Natural merging.
- Balanced multiway merging.
- Polyphase sort.
- Distribution of initial runs.

2.3 Eficiencia en tiempo de ejecución

Una medida de eficiencia es contar el número de comparaciones (C), contar el número de movimientos de ítems (M), estos están en función del número(n) de ítems a ser ordenados, se toma n como el número de elementos que tiene el arreglo o vector a ordenar y se dice que un algoritmo realiza $O(n^2)$ comparaciones cuando compara n veces los n elementos, $n \times n = n^2$

2.4 Método de Inserción Directa

Este método toma cada elemento del arreglo para ser ordenado y lo compara con los que se encuentran en posiciones anteriores a la de él dentro del arreglo. Si resulta que el elemento con el que se está comparando es mayor que el elemento a ordenar, se recorre hacia la siguiente posición superior. Si por el contrario, resulta que el elemento con el que se está comparando es menor que el elemento a ordenar, se detiene el proceso de comparación pues se encontró que el elemento ya está ordenado y se coloca en su posición (que es la siguiente a la del último número con el que se comparó).

Pseudo código

Este procedimiento recibe el arreglo de datos a ordenar A[] y altera las posiciones de sus elementos hasta dejarlos ordenados de menor a mayor. N representa el número de elementos que contiene A[].

```
Inicio
  A[0]? -999              Se inicializa bandera
  N?8; K?2           Se inicializa tamaño arreglo e
  Repetir hasta K=N            inicio inspección
      TEMP?A[K]
      PTR?K-1
      Repetir Mientras TEMP<A[PTR]
          A[PTR+1] ? A[PTR]
          PTR? PTR-1
      Fin repetir mientras
      A[PTR+1] ? TEMP
      K? K+1
  Fin repetir hasta
Fin
```

Ejemplo: sea el siguiente arreglo A[8] = { 7, 3, 4, 1, 8, 2, 6, 5 }, N=8

Prueba de Escritorio

A[0]	N	K	TEMP	PTR	A[PTR]	A[PTR+1]
-999	8	2	3	1	7	7
				0	-999	3
		3	4	2	7	7
				1	3	4
		4	1	3	7	7
				2	4	4
				1	3	3
				0	-999	1
		5	8	4	7	8
		6	2	5	8	8
				4	7	7

0	**-999**	-999	-999	-999	-999	-999	-999	-999
1	**7**	3	3	1				
2	**3**	7	4	3				
3	**4**	4	7	4				
4	**1**	1	1	7				
5	**8**	8	8	8				
6	**2**	2	2	2				
7	**6**	6	6	6				
8	**5**	5	5	5				

2.5 Método De Selección

El método de ordenamiento por selección consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición. Luego el segundo mas pequeño, y así sucesivamente hasta ordenar todo el arreglo.

Pseudo código

```
Inicio
  Repetir K desde 1 hasta N-1
       MIN?A[K]
       POS?K
       Repetir J desde K+1 hasta N
          Si MIN>A[J] entonces
            MIN ? A[J]
            POS? J
          Fin _ si
       Fin_ repetir
       TEMP ? A[K]
       A[K] ? A[POS]
       A[POS]? TEMP
  Fin_ repetir_ desde
Fin
```

Prueba de escritorio

K	MIN	POS	J	A[J]	TEMP	A[K]	A[POS]
1	7	1	2	3			
1	3	2	3	4			
1	3	2	4	1			
1	1	4	5	8			
1	1	4	6	2			
1	1	4	7	6			
1	1	4	8	5	7	1	7
2	3	2	3	4			
2	3	2	4	7			
2	3	2	5	8			
2	3	2	6	2			
2	2	6	7	6			
			8	5	3	2	3

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

2.6 Método Burbuja

El bubble sort, también conocido como ordenamiento burbuja, funciona de la siguiente manera: Se recorre el arreglo intercambiando los elementos adyacentes que estén desordenados. Se recorre el arreglo tantas veces hasta que ya no haya cambios. Prácticamente lo que hace es tomar el elemento mayor y lo va recorriendo de posición en posición hasta ponerlo en su lugar.

2.7 Método de Shell

Ordenamiento de disminución incremental. Nombrado así debido a su inventor Donald Shell. Ordena subgrupos de elementos separados K unidades (respecto de su posición en el arreglo) del arreglo original. El valor K es llamado incremento.
Después de que los primeros K subgrupos han sido ordenados (generalmente utilizando INSERCION DIRECTA), se escoge un nuevo valor de K más pequeño, y el arreglo es de nuevo partido entre el nuevo conjunto de subgrupos. Cada uno de los subgrupos mayores es ordenado y el proceso se repite de nuevo con un valor más pequeño de K. Eventualmente el valor de K llega a ser 1, de tal manera que el subgrupo consiste de todo el arreglo ya casi ordenado. Al principio del proceso se escoge la secuencia de decrecimiento de incrementos; el último valor debe ser 1. "Es como hacer un ordenamiento de burbuja pero comparando e intercambiando elementos." Cuando el incremento toma un valor de 1, todos los elementos pasan a formar parte del subgrupo y se aplica inserción directa. El método se basa en tomar como salto N/2 (siendo N el número de elementos) y luego se va reduciendo a la mitad en cada repetición hasta que el salto o distancia vale 1.

2.8 Búsqueda

La búsqueda es una de las operaciones que aparecen con más frecuencia en programación de ordenadores. Básicamente, se trata de buscar un elemento determinado dentro de una colección de N datos, que generalmente se representa mediante una estructura de tipo vector A[N], donde los tipo de datos pueden ser entero, real, etc. Esto significa, que dado un argumento de búsqueda, X, se trata de encontrar el índice i del vector A para el cual se verifica A(i)=X.

2.8.1 Búsqueda lineal

Es la solución más obvia y que se aplica cuando no hay ninguna información adicional acerca de los datos buscados. Consiste en recorrer secuencialmente el vector hasta encontrar el dato buscado, siendo por tanto las condiciones de parada: A(i)=X se halló el elemento buscado ó i=N se llegó al final del vector. Se debe comprobar después si la operación tuvo éxito, o simplemente se llegó al final del vector.

Pseudo código

```
Inicio
    POS? 0
    Repetir desde i = 1 hasta N
         Si A[i] = t  entonces
              POS? i
         Fin_ si
    Fin_ repetir_ desde
Fin
```

2.8.2 Búsqueda binaria

La operación de búsqueda puede ser mucho más eficiente si se sabe que los datos están previamente ordenados. Basta considerar como ejemplo un diccionario, en el que hacemos una búsqueda gracias a la ordenación alfabética de las palabras. En otro caso, sería algo completamente inutilizable.

La idea clave consiste en inspeccionar un elemento cualquiera, de índice m (normalmente en la mitad), y compararlo con el argumento X. Los posibles casos son:

- A(m)=X.- ha terminado la búsqueda y el índice buscado es m
- A(m)<X.- sabemos que todos los elementos a la izquierda de m son menores que X, y por tanto pueden ser eliminada esta región de la zona de búsqueda y considerar sólo la zona derecha (desde m+1 hasta N)
- A(m)>X.- sabemos que todos los elementos a la derecha de m son mayores que X, y considerar para la búsqueda sólo la zona izquierda (desde 1 hasta m-1)
- La repetición de este proceso de forma iterativa constituye este algoritmo. Para ello, se utilizan dos variables de índice, I y D, que marcan los extremos Izquierdo y Derecho de la zona de búsqueda considerada.

```
INICIO
     Leer X
     i?1; j?n-1;
     m? (i+j)\2    /* \ división entera */
     Mientras (a[m] <> X  y  i<j ) hacer
         Si X < a[m]  entonces
          j ? m-1
         caso contrario
          i ? m+1
         fin_si
         m? (i+j)\2 ;
     Fin_mientras
     Si i >= j entonces
          Escribir "Dato buscado no se encuentra"
       Caso contrario
          Escribir "Dato fue encontrado en
                    posición ", m;
     Fin_si
FIN
```

2.8.3. Búsqueda mediante transformación de claves (hashing)

Es un método de búsqueda que aumenta la velocidad de búsqueda, pero que no requiere que los elementos estén ordenados. Consiste en asignar a cada elemento un índice mediante una transformación del elemento. Esta correspondencia se realiza mediante una función de conversión, llamada función hash. La correspondencia más sencilla es la identidad, esto es, al número 0 se le asigna el índice 0, al elemento 1 el índice 1, y así sucesivamente. Pero si los números a almacenar son demasiado grandes esta función es inservible. Por ejemplo, se quiere guardar en un array la información de los 1000 usuarios de una empresa, y se elige el número de DNI como elemento identificativo. Es inviable hacer un array de 100.000.000

elementos, sobre todo porque se desaprovecha demasiado espacio. Por eso, se realiza una transformación al número de DNI para que nos de un número menor, por ejemplo coger las 3 últimas cifras para guardar a los empleados en un array de 1000 elementos. Para buscar a uno de ellos, bastaría con realizar la transformación a su DNI y ver si está o no en el array.

La función de hash ideal debería ser biyectiva, esto es, que a cada elemento le corresponda un índice, y que a cada índice le corresponda un elemento, pero no siempre es fácil encontrar esa función, e incluso a veces es inútil, ya que puedes no saber el número de elementos a almacenar. La función de hash depende de cada problema y de cada finalidad, y se pueden utilizar con números o cadenas, pero las más utilizadas son:

- ✓ **Restas sucesivas**.- esta función se emplea con claves numéricas entre las que existen huecos de tamaño conocido, obteniéndose direcciones consecutivas. Por ejemplo, si el número de expediente de un alumno universitario está formado por el año de entrada en la universidad, seguido de un número identificativo de tres cifras, y suponiendo que entran un máximo de 400 alumnos al año, se le asignarían las claves:

  1998-000 ⟶ 0 = 1998000-1998000
  1998-001 ⟶ 1 = 1998001-1998000
  1998-002 ⟶ 2 = 1998002-1998000
  ...
  1998-399 ⟶ 399 = 1998399-1998000
  1999-000 ⟶ 400 = 1999000-1998000+400
  ...
  yyyy-nnn ⟶ N = yyyynnn-1998000+(400*(yyyy-1998))

- ✓ **Aritmética modular**.- el índice de un número es resto de la división de ese número entre un número N prefijado, preferentemente primo. Los números se guardarán en las direcciones de memoria de 0 a N-1. Este método tiene el problema de que cuando hay N+1 elementos, al menos un índice es señalado por dos elementos (teorema del palomar). A este fenómeno se le llama colisión, y es tratado más adelante. Si el número N es el 13, los números siguientes quedan transformados en:

  13000000⟶ 0
  12345678⟶ 7
  13602499⟶ 1
  71140205⟶ 6
  73062138⟶ 6

- ✓ **Mitad del cuadrado**.- consiste en elevar al cuadrado la clave y coger las cifras centrales. Este método también presenta problemas de colisión:

  123*123=15129 ⟶51
  136*136=18496 ⟶84
  730*730=532900 ⟶29
  301*301=90601 ⟶06
  625*625=390625 ⟶06

- ✓ **Truncamiento.-** consiste en ignorar parte del número y utilizar los elementos restantes como índice. También se produce colisión. Por ejemplo, si un número de 8 cifras se debe ordenar en un array de 1000 elementos, se pueden coger la primer, la tercer y la última cifras para formar un nuevo número:

13000000 ⟶100
12345678 ⟶138
13602499 ⟶169
71140205 ⟶715
73162135 ⟶715

- ✓ **Plegamiento**.- consiste en dividir el número en diferentes partes, y operar con ellas (normalmente con suma o multiplicación). También se produce colisión. Por ejemplo, si dividimos los números de 8 cifras en 3, 3 y 2 cifras y se suman, dará otro número de tres cifras (y si no, se cogen las tres últimas cifras):

  13000000 ⟶130=130+000+00
  12345678 ⟶657=123+456+78
  71140205 ⟶118 ⟶1118=711+402+05
  13602499 ⟶259=136+024+99
  25000009 ⟶259=250+000+09

Tratamiento de colisiones.- Pero ahora se nos presenta el problema de qué hacer con las colisiones, qué pasa cuando a dos elementos diferentes les corresponde el mismo índice. Pues bien, hay tres posibles soluciones:

- Cuando el índice correspondiente a un elemento ya está ocupado, se le asigna el primer índice libre a partir de esa posición. Este método es poco eficaz, porque al nuevo elemento se le asigna un índice que podrá estar ocupado por un elemento posterior a él, y la búsqueda se ralentiza, ya que no se sabe la posición exacta del elemento.
- También se pueden reservar unos cuantos lugares al final del array para alojar a las colisiones. Este método también tiene un problema: ¿Cuánto espacio se debe reservar? Además, sigue la lentitud de búsqueda si el elemento a buscar es una colisión.
- Lo más efectivo es, en vez de crear un array de número, crear un array de punteros, donde cada puntero señala el principio de una lista enlazada. Así, cada elemento que llega a un determinado índice se pone en el último lugar de la lista de ese índice. El tiempo de búsqueda se reduce considerablemente, y no hace falta poner restricciones al tamaño del array, ya que se pueden añadir nodos dinámicamente a la lista

TRABAJO

Codificar en el Lenguaje de Programación JAVA los siguientes algoritmos:

- o Inserción
- o Selección
- o Método Shell
- o Método Quicksort
- o Búsqueda Hash

La presentación se realizará en el turno de Laboratorio respectivo.

Semana 03: Aplicaciones Algoritmos Tipo

Dinámica de Grupo N°01

Instrucciones.- Trabajar en grupos de acuerdo con las indicaciones del profesor.

Se tiene el siguiente algoritmo de ordenamiento:

METODO DE LA BURBUJA
(Algoritmo no optimizado)

```
burbuja(arreglo a){
     desde i =1 hasta longitud(a)
          desde  j = 0 hasta longitud(a)-1
               si ( a[ j ] > a[ j+1 ] ) entonces
                    aux  = a[ j ]
                    a[ j ]    = a[ j+1 ]
                    a[ j+1 ] = aux
               fin_si
          fin_desde_j
     fin_desde_i
fin_burbuja
```

Análisis del Algoritmo

- ✓ Estabilidad.- Este algoritmo nunca intercambia registros con claves iguales, por lo tanto es estable
- ✓ Requerimientos de Memoria.- Este algoritmo solo requiere de una variable adicional para realizar los intercambios.
- ✓ Tiempo de Ejecución.- El ciclo interno se ejecuta **n** veces para una lista de n elementos. El ciclo externo también se ejecuta n veces. Es decir que la complejidad es n*n = O($\mathbf{n^2}$). El comportamiento del caso promedio depende del orden de entrada de los datos, pero es solo un poco mejor que el del peor caso, y sigue siendo O($\mathbf{n^2}$).

Ventajas

- ✓ Fácil implementación
- ✓ No requiere memoria adicional

Desventajas

- ✓ Muy lento
- ✓ Realiza numerosas comparaciones
- ✓ Realiza numerosos intercambios.

Resolver

1. Se pide optimizar el algoritmo.
2. Puede ser que los datos queden ordenados antes de completar el ciclo externo. Modifique el algoritmo que verifique si se han realizado intercambios. Sino se han hecho entonces se termina con la ejecución, pues significa que los datos ya están ordenados.
3. Prueba de escritorio para cada caso.

Se tiene el siguiente algoritmo de ordenamiento:

Algoritmo Shaker Sort

```
 izq=2;  der=N;  k=N;
 Repetir
     Desde i = der hasta izq hacer
           si (A[i-1]>A[i])  entonces
               aux=A[i-1];
               A[i-1]=A[i];
               A[i]=aux;
               k=1;
           fin_si
      Fin_desde
      izq=k+1;

     Desde i = izq hasta der
           si (A[i-1]>A[i]) entonces
               aux=A[i-1];
               A[i-1]=A[i];
               A[i]=aux;
               k=1;
               fin_si
     Fin_desde
     der=k-1;
hasta que izq>der
```

Resolver

1. Efectuar la prueba de escritorio para el siguiente arreglo
   A[6] = { 50, 1, 20, 21, 2, 51}
2. Diga que hace el algoritmo?
3. ¿Cuántas comparaciones hace?
4. ¿Cuántos intercambios hace?
5. Corregir errores si los hubiere
6. ¿Cuál es el tiempo de ejecución del mejor caso y del peor caso?

Dinámica de Grupo N°02

Instrucciones.- Trabajar en grupos de acuerdo con las indicaciones del profesor.

Se tiene el siguiente algoritmo de ordenamiento:

COCKTAIL SORT

El ordenamiento de burbuja bidireccional (cocktail sort en inglés) es un algoritmo de ordenamiento que surge como una mejora del algoritmo ordenamiento de burbuja.

Pseudocódigo:

```
Procedimiento Ordenacion_Sacudida (v:vector, tam:entero)
 Variables
   i, j, izq, der, ultimo: tipoposicion;
   aux: tipoelemento;
 Inicio
   //Limites superior e inferior de elementos ordenados
   izq <- 2
   der <- tam
   ultimo <- tam

   Repetir
     //Burbuja hacia la izquierda}
     //Los valores menores van a la izquierda
     Para i <- der hasta izq hacer
         Si v(i-1) > v(i) entonces
             aux <- v(i)
             v(i) <- v(i-1)
             v(i-1) <- aux
             ultimo <- i
         Fin_si
     Fin_para

     izq <- ultimo+1

     //Burbuja hacia la derecha
     //Los valores mayores van a la derecha
     Para j <- izq hasta der hacer
         Si v(j-1) > v(j) entonces
             aux <- v(j)
             v(j) <- v(j-1)
             v(j-1) <- aux
             ultimo <- j
         Fin_si
     Fin_para
     der <- ultimo-1
   Hasta (izq > der)
 Fin
```

Análisis del Algoritmo

- ✓ Estabilidad.-
- ✓ Requerimientos de Memoria.-
- ✓ Tiempo de Ejecución.-

Ventajas

- ✓
- ✓

Desventajas

- ✓
- ✓
- ✓

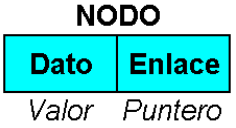
Tarea

1. Corregir errores si los hubiere
2. Prueba de escritorio para los siguientes datos:

   arreglo[] = {10,23,6,4,223,2,112,3,6,34}
3. ¿Cuántas comparaciones hace?
4. ¿Cuántos intercambios hace?
5. ¿Cuál es el tiempo de ejecución del mejor caso y del peor caso?

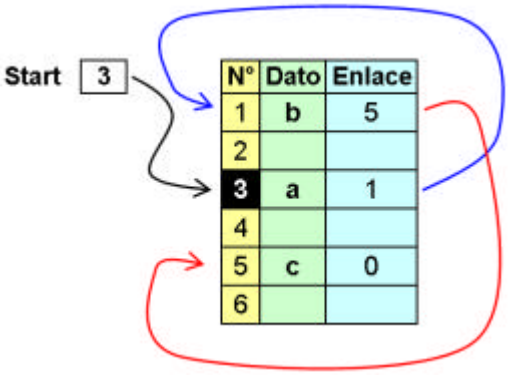
Semana 04: Listas. Listas Enlazadas. Operaciones Básicas

Una lista enlazada o lista unidireccional esta conformada por una colección lineal de elementos, llamados nodos, donde el orden de los mismos se establece mediante punteros. Cada nodo se divide en dos partes: Una primera parte que contiene la información asociada al elemento, y una segunda parte, llamada campo de enlace o campo de puntero al siguiente, que contiene la dirección del siguiente nodo de la lista.



Representación de Listas Enlazadas en Memoria

Lo haremos empleando dos arreglos lineales a los cuales llamaremos DATO y ENLACE, donde DATO[k] y ENLACE[k] contienen los valores y los punteros de cada nodo de la LISTA (enlazada) respectivamente. Este tipo de representación necesitará de una variable de trato especial para indicar el START del primer elemento de la lista, y una variable bandera que denominaremos NULO y nos permitirá indicar el final de la lista. Para efectos didácticos emplearemos valores positivos dentro de los ejemplos por lo tanto el valor de NULO será igual a cero.



Recorrido de una Lista Enlazada

Elaboraremos el algoritmo que permite procesar toda la información contenida en una lista enlazada. Para ello utilizaremos una variable PTR, que apuntará al nodo procesado a cada momento.

Pseudocódigo

```
INICIO
     PTR? START
     Mientras PTR <> 0 hacer
          Leer DATO[PTR]
          PTR? ENLACE[PTR]
     Fin _ mientras
FIN
```

Prueba de Escritorio

START	PTR	DATO[PTR]	ENLACE[PTR]
3	3	a	1
	1	b	5
	5	c	0

BUSQUEDA EN UNA LISTA ENLAZADA

Tenemos dos algoritmos que nos permitirán determinar la Posición de la primera aparición de un determinado DATO dentro de la lista. El primer algoritmo no necesita que la lista esta ordenada y el segundo si lo exige.

a. Búsqueda en Lista Enlazada No Ordenada

Aquí emplearemos una variable POS que indicara la posición de la primera aparición de VALOR el elemento buscado dentro de la LISTA.

Pseudocódigo

```
INICIO
  PTR? START
  Mientras PTR <> 0 hacer
       Si VALOR=DATO[PTR] entonces
            POS? PTR
            Salir
       Caso contrario
            PTR? ENLACE[PTR]
       Fin _ si
  Fin _ mientras
  POS? 0 (la búsqueda no tuvo éxito)
FIN
```

Start 3

N°	Dato	Enlace
1	s	4
2		
3	f	1
4	d	8
5		
6	p	0
7		
8	a	6
9		

Prueba de Escritorio

VALOR	START	PTR	DATO[PTR]	POS	ENLACE[PTR]
A	3	3	F		1
		1	S		4
		4	D		8
		8	A	8	

b. Búsqueda en Lista Enlazada Ordenada

Cuando la lista enlazada está ordenada, podemos buscar el elemento (VALOR) comparándolo uno a uno con cada DATO[PTR], y podemos terminar la búsqueda cuando VALOR sea mayor que DATO[PTR].

Pseudocódigo

```
INICIO
  PTR? START
  Mientras PTR <> 0 hacer
       Si VALOR>DATO[PTR] entonces
            PTR? ENLACE[PTR]
       Caso contrario
            Si VALOR = DATO[PTR] entonces
                 POS? PTR
            Caso contrario
                 POS? 0
            Fin _si
            Salir
       Fin _ si
  Fin _ mientras
  POS? 0 (la búsqueda no tuvo éxito)
FIN
```

Start 3

N°	Dato	Enlace
1	b	4
2		
3	a	1
4	c	8
5		
6	e	0
7		
8	d	6
9		

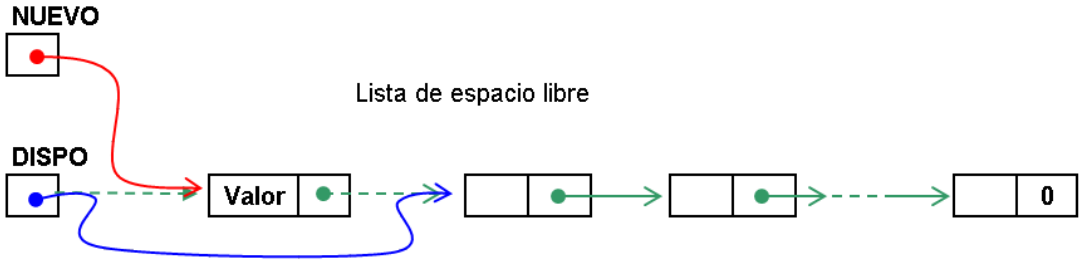
Prueba de Escritorio

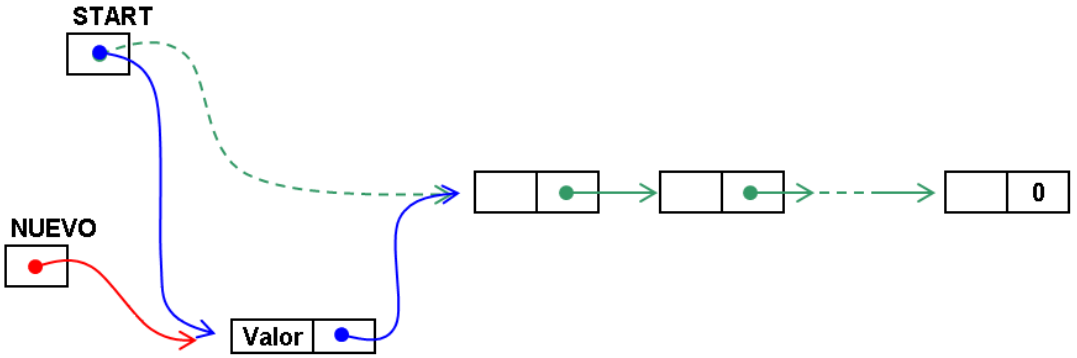
VALOR	START	PTR	DATO[PTR]	ENLACE[PTR]	POS
D	3	3	A	1	
		1	B	4	
		4	C	8	
		8	D		8

INSERCION EN UNA LISTA ENLAZADA

En forma básica podemos insertar un elemento al principio, continuación de un nodo determinado y cuando insertamos un nodo en una lista enlazada previamente ordenada.

a. Inserción al Principio de una Lista Enlazada



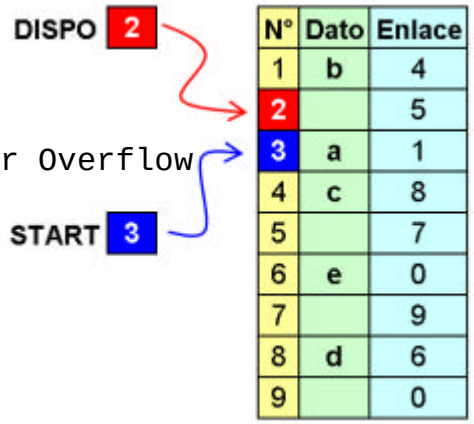


Variables

VALOR	elemento a insertar	DISPO	lista de disponibles
START	inicio de la lista	NUEVO	extrae nodo de DISPO
ENLACE	arreglo de enlaces	DATO	arreglo de DATOS

Pseudocódigo

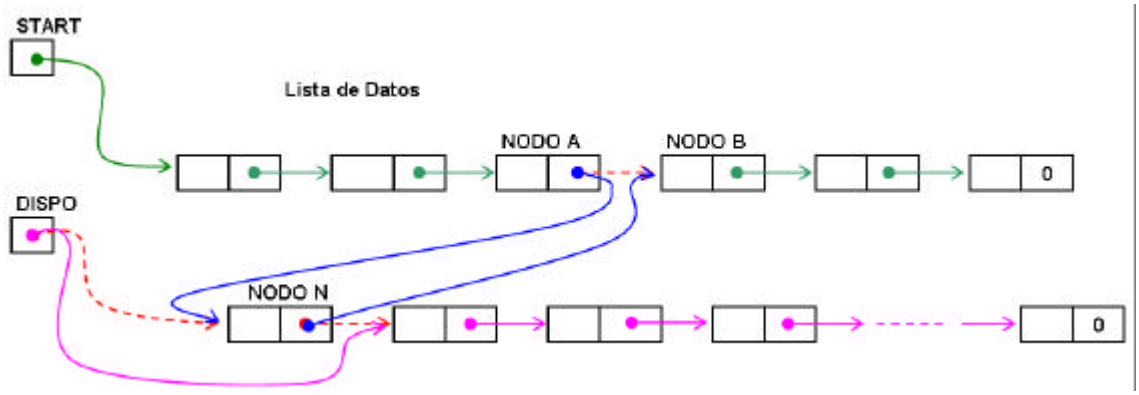
```
INICIO
   Si DISPO = 0 entonces Escribir Overflow
   NUEVO? DISPO
   DISPO? ENLACE[DISPO]
   DATO[NUEVO] ? VALOR
   ENLACE[NUEVO] ? START
   START? NUEVO
FIN
```



Prueba de Escritorio

VALOR	START	DISPO	NUEVO	ENLACE[DISPO]	DATO[NUEVO]	ENLACE[NUEVO]
G	3 2	2 5	2	5	G	3

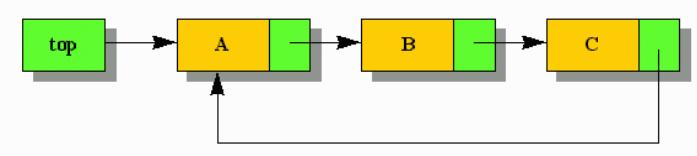
b. Inserción a continuación de un nodo determinado en una Lista Enlazada



Pseudocódigo

Algoritmo que INSERTA un elemento después de una posición determinada

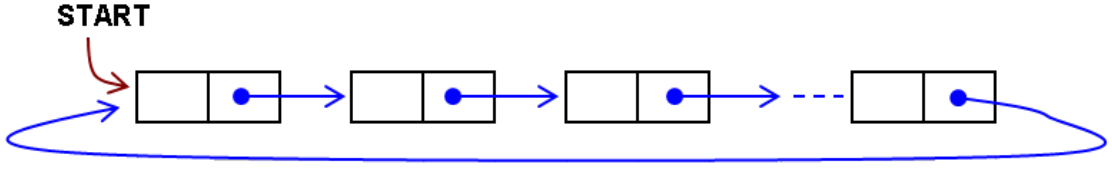
```
PRIMERO = 1          DISPO = 1
INICIO
     Leer VALOR; Leer LUGAR
     Si DISPO = (-1) entonces
          Escribir "Lista llena"
     Caso contrario
          NUEVO? DISPO
          DISPO? ENLACE[DISPO]
          DATO[NUEVO] ? VALOR
          Si PRIMERO <> DISPO entonces
               AUX? ENLACE[LUGAR]
               ENLACE[LUGAR]? NUEVO
               ENLACE[NUEVO] ? AUX
          Caso contrario
               ENLACE[LUGAR] ? NUEVO
          Fin _ si
     Fin _ si
FIN
```



LISTAS CIRCULARES

En las listas lineales siempre hay un último nodo que apunta al valor Nulo (bandera que indica final: Nil, cero, etc.)

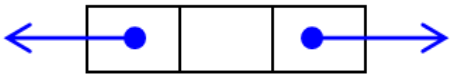
Para poder acceder a cualquier nodo de la lista, se modifica el algoritmo lineal y el último nodo en vez de a puntar a nulo apunta al primer nodo generándose una estructura circular.

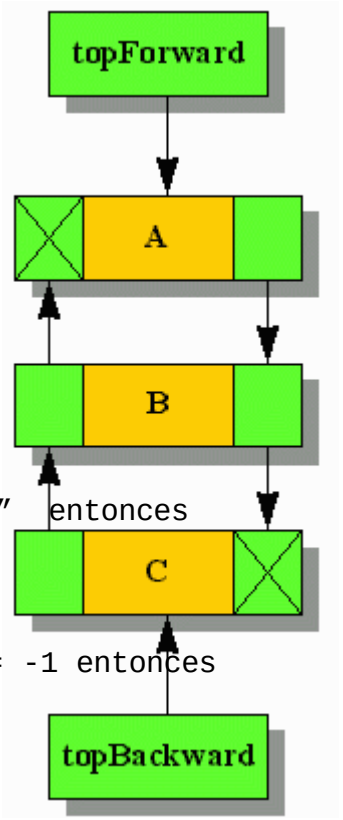


Una lista circular no tiene ni primero ni último nodo, pero resulta bastante útil establecer un primer nodo, al que denominaremos cabecera y que debe tener un valor que sea distinto al empleado por los otros nodos. Se debe tener cuidado de no producir bucles (loops) infinitos.

LISTAS DOBLEMENTE ENLAZADAS

Las listas simplemente enlazadas solo permiten el recorrido de la misma en un solo sentido, en algunos casos será necesario e poder avanzar en ambos sentidos, para ello cada en nodo de una lista doblemente enlazada existen 2 enlaces: uno al siguiente nodo y otro al anterior.



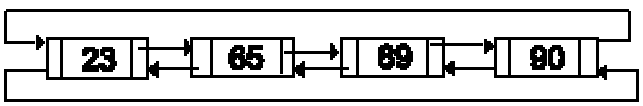


Pseudocódigo

Algoritmo que INSERTA un elemento en una
Lista Doblemente Enlazada

```
PRIMERO = 1          FINAL = 1  DISPO = 1

INICIO
     Leer VALOR; Leer LUGAR
     Si DISPO = (-1) entonces
          Escribir "Lista llena"
     Caso contrario
          NUEVO? DISPO
          DISPO? LIBRE[DISPO]
          DATO[NUEVO] ? VALOR
          Si PRIMERO = FINAL y DATO[PRIMERO] = ""  entonces
               SIG[NUEVO] ? SIG[LUGAR]
               ANT[NUEVO] ? ANT[LUGAR]
          Caso contrario
               Si PRIMERO = FINAL o SIG[LUGAR] = -1 entonces
                    AUXSIG? SIG[LUGAR]
                    SIG[LUGAR] ? NUEVO
                    SIG[NUEVO] ? AUXSIG
                    ANT[NUEVO] ? LUGAR
                    FINAL? NUEVO
               Caso contrario
                    AUXSIG? SIG[LUGAR]; AUXANT? ANT[SIG[LUG]]
                    ANT[SIG[LUGAR]] ? NUEVO;
                    SIG[LUGAR] ? NUEVO
                    SIG[NUEVO] ? AUXSIG; ANT[NUEVO] ? AUXANT
               Fin _ si
          Fin _ si
     Fin _ si
FIN
```

También se pueden manejar listas doblemente enlazadas circulares



Semana 05: Aplicaciones

Dinámica de Grupo N°03

Instrucciones **-** Trabajar en grupos de acuerdo con las indicaciones del profesor.

1. Escribir el pseudo código que efectué la búsqueda de un dato y a continuación elimine el dato del nodo anterior en una lista enlazada. Realice la prueba de escritorio para los siguientes datos: START = 8, dato referencia eliminación: s (ese).

Índice	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
DATO	a	o	s		u	c	i		e	n
Enlace	2	9	6		0	4	1		5	-1

2. Escribir el pseudo código que efectúe la inserción al inicio de la lista y entre dos nodos, así como el recorrido en la siguiente lista circular. INICIO = 4. DISPO = 1. Haga la prueba de escritorio para insertar la letra “p” al inicio de la lista y la letra “l” después de la letra “p”, al final el recorrido que palabra forma.

Índice	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
DATO	d		c		a		i		o	
Enlace	8	3	6	5	2	7	0	9	4	-1

3. Escribir el pseudo código que liste cada nota con su diferencia con respecto al promedio de notas. INICIO = 4. (5 puntos)

Índice	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
NOTAS	13	3	9	7	12	8	14	16	5	19
Enlace	8	6	-1	7	1	3	9	2	5	0

Además sobre ordenamiento

4. Tiene el siguiente algoritmo, efectué la prueba de escritorio, verifique su corrección y explique como trabaja cada parte del algoritmo, indique cuantas iteraciones y comparaciones realiza, diga como se llama el método. Compárelo con el método de la Burbuja Optimizado (escribir el Pseudocódigo) haga la prueba de escritorio para el mismo juego de valores y diga cual es más eficiente.

```
int A[] = {7, 3, 4, 1};
 int j, incremento, temp, cont = 0;
 incremento = 3;
 //
```

```
while (incremento > 0) {
    for (int i=0; i < A.length; i++) {
       j = i;
       temp = A[i];
       while ((j >= incremento) && (A[j-incremento] > temp))
        {
         A[j] = A[j - incremento];
         j = j - incremento;
        }
        A[j] = temp;
       }
    if (incremento/2 != 0)
        incremento = incremento/2;
    else if (incremento == 1)
        incremento = 0;
    else
        incremento = 1;
}
//
for (int i=0; i<A.length; i++){
    System.out.println(i+").- "+A[i]);
}
```

5. Tiene el siguiente algoritmo, efectué la prueba de escritorio, verifique su corrección y explique como trabaja el algoritmo, indique cuantas comparaciones realiza.

```
AlgortimoDeOrdenamiento {
    int dato[ ] = {11,3,6,2,9,1,8,10,4,7,5};
    desde (int i = 0; i < dato.length; i++)
        imprimir (i + "  " + dato[i]);

    int j, T, limit = dato.length, st = -1;

    mientras (st < limit) {
       boolean flipped = falso;
       st = st + 1;
       limit = limit - 1;
       for (j = st; j < limit; j++) {
          if (dato[j] > dato[j + 1]) {
             T = dato[j];
             dato[j] = dato[j + 1];
             dato[j + 1] = T;
             flipped = verdadero;
          }
       } if (no flipped) {
          salir;
       }
       for (j = limit; j >= st;) {
          if (dato[j] > dato[j + 1]) {
             T = dato[j];
             dato[j] = dato[j + 1];
             dato[j + 1] = T;
             flipped = verdadero;
          }
       }
       if (no flipped) {
          salir;
       }
    }
    for (int i=0; i<dato.length; i++)
       imprimir (i + "  " + dato[i]);
}
```

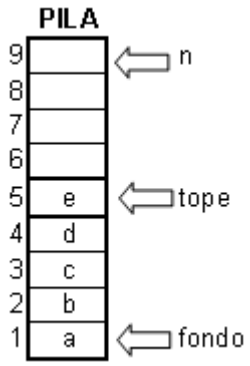
SEMANA 6: Examen de 1era Unidad

Semana 07: Estructura Lineal: Pilas. Operaciones Básicas.

La pila es una estructura lineal, es una lista o secuencia finita de elementos de algún conjunto donde las inserciones y eliminaciones se realizan por un solo extremo de la lista, llamada la cima o tope de la pila. A este tipo de estructura se le denomina LIFO (last in, first out), último en entrar primero en salir, por que eso es lo que ocurre cuando se opera una pila, el último elemento ingresado es el primero es ser sacado.
Las operaciones sobre pilas son:
1. Creación o limpieza: Inicializa pila al estado vacío.
2. Pila vacía: Determina si la pila está vacía.
3. Pila llena: Determina si la pila está llena.
4. Apilamiento (METER): Inserta un nuevo elemento a la pila (en la cima).
5. Desapilamiento (SACAR): Elimina el elemento en la cima de la pila.

Implementación: La pila se puede representar mediante un registro con dos campos:
- ✓ Un arreglo Elementos donde se almacenan los elementos de la pila puesto que son del mismo tipo.
- ✓ Una variable Cima que apunta al elemento que está en la cima de la pila.

Representación
Una pila la representaremos como una lista unidireccional o array lineal. PILA (n)



Donde:
tope: Indica la dirección del último elemento de la pila

n: Número máximo de elementos de la PILA

Si ***tope es igual a n***, la PILA está llena. Y si intentamos meter un elemento más a la PILA, habrá un desbordamiento de PILA (overflow)

Si ***tope es igual a 0***, la PILA está vacía. Y si intentamos sacar un elemento de la PILA, habrá un subdesbordamiento de PILA (underflow)

Aplicaciones de las Pilas
- ✓ Desde el sistema operativo se puede controlar la ejecución de todas las ordenes de un archivo batch
- ✓ Dentro de un programa se realizan llamadas a subprogramas entonces el programa principal debe recordar el lugar de donde se hizo la llamada para poder retornar allí cuando el subprograma halla terminado y poder continuar con la ejecución del programa
- ✓ Para formar cadenas de caracteres
- ✓ Para separar un texto en letras, dígitos y símbolos
- ✓ Para evaluar expresiones matemáticas.

ALGORITMO METER (PUSH)
Función : Añade nuevo elemento a la cima de la pila
Entrada : Pila, nuevo elemento
Precondiciones : Pila no está llena
Salida : Pila cambiada
Poscondiciones : Pila = pila original con nuevo elemento añadido a la cabeza

```
INICIO
   Leer X /* X elemento a insertar */
   Si TOPE >= N entonces
      Escribir "Pila Llena"
   Caso contrario
      TOPE? TOPE + 1
      PILA (TOPE) ? X
   Fin _ si
FIN
```

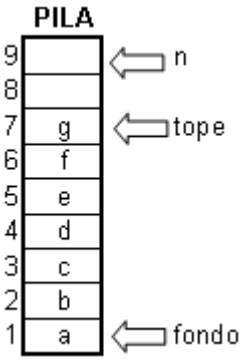
ALGORITMO SACAR (POP)

Función : Quita elemento de la cima de la pila y lo devuelve en variable
Entrada : Pila
Precondiciones : Pila no está vacía
Salida : Pila cambiada, Elemento sacado (en variable X)
Poscondiciones : Pila = pila original con elemento quitado de la cima

```
INICIO
   Si TOPE = 0 entonces
      Escribir "Pila Vacía"
   Caso contrario
      X ? PILA (TOPE)
      TOPE? TOPE - 1
   Fin _ si
FIN
```

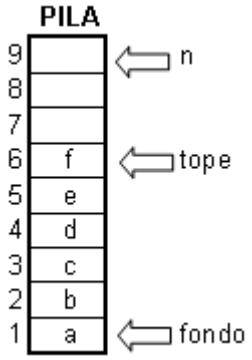
Ejemplo:
Meter letra **g** en PILA
Prueba de Escritorio

X	N	TOPE
g	9	6
		7

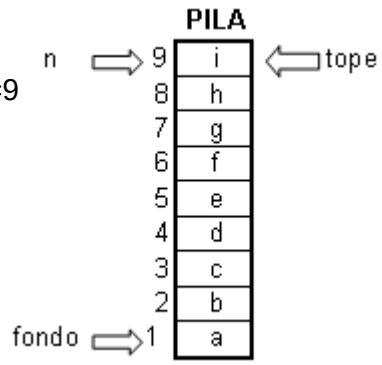


Sacar letra **g** de PILA
Prueba de Escritorio

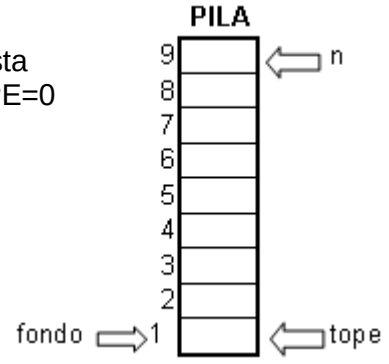
X	N	TOPE
g	9	7
		6



Si la pila esta llena, TOPE=9 se imprime mensaje "Pila Llena"



Si la pila esta vacía, TOPE=0 se imprime mensaje "Pila vacia"



Ejercicios

1.- Elabore el algoritmo (Pseudo código) que permita ingresar una cadena y luego la imprima en orden inverso, emplear una estructura PILA.

2.- Supongamos que un algoritmo (Pseudo código) requiere 2 pilas. A(n1) y B(n2). No disponemos de mucha memoria y para evitar desbordamientos, es decir que la cantidad de elementos de A sea mayor que n1 o que la cantidad de elementos de B sea mayor que n2. Empleamos un solo array C con (n1 + n2) elementos, con la particularidad que la Pila A mete sus datos por la izquierda desde el elemento n. Modifique las operaciones METER y SACAR para este caso.

Implemente ambos algoritmos en su herramienta de programación.

APLICACIONES DE PILAS

NOTACIÓN POLACA

Lleva ese nombre en honor a su descubridor el Polaco Jan Lukasiewicz.

Al evaluar expresiones aritméticas que incluyen valores constantes y símbolos de operaciones, para obtener el resultado debemos practicar varios barridos teniendo en cuenta la jerarquía de los operadores. El método Notación Polaca consiste en resolver la expresión en un solo barrido sin tener en cuenta las prioridades de los operadores ni los paréntesis.

Este método tiene 3 formas de representar las expresiones y se denominan. Infija, prefija y postfija.

a. Forma Infija
Denota la costumbre usual de escribir operadores binarios entre sus operandos.
A+B C*D

b. Forma Prefija
O Polaca, cuando los operadores se escriben antes que sus operandos.
+AB *CD

c. Forma Sufija
O Postfija, o Polaca inversa, cuando los operadores se escriben después que sus operandos.
AB+ CD*

Algunos interpretes de lenguajes de programación y calculadoras de bolsillo, tales como Hewlett Packard, contienen una pila para este fin y utilizan notación Postfija (Polaca inversa), en notación polaca nunca se necesitan paréntesis al escribir expresiones. El orden en que se realizan las operaciones esta determinado por la posición de los operadores y de los operandos en la expresión.

Conversión de Expresiones Infijas a Prefijas y a Sufijas

Pasos a seguir para efectuar la conversión

1. Las operaciones de prioridad más alta se convierten primero.
2. Después que una parte de la expresión ha sido convertida se trata como un solo operando.

Infija	Prefija	Sufija
(A+B)*C	[+AB]*C	[AB+]*C
	*+ABC	AB+C*
A+(B*C)	A+[*BC]	A+[BC*]
	+A*BC	ABC*+
(A+B)/(C-D)	[+AB]/[-CD]	[AB+]/[CD-]
	/+AB-CD	AB+CD-/

Ejercicios

1.- $$\frac{A+B^D}{Q-R*(S+\frac{T}{V})}$$

2.- $$A+(B*C-\frac{D}{E^F}*G)*H$$

3.- $$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

4.- 234567+*+*+

5.- 12, 8, 3, -, /, 1, 2, 4, 6, +, *, +, +

6.- $$S=\frac{n}{2}(2a+(n-1)*d)$$ Suma de los n primeros términos de una progresión aritmética

7.- $$G=a\frac{1-r^n}{1-r}$$ Suma de los n primeros términos de una progresión aritmética

Algoritmo de Transformación de una expresión Infija a Expresión Postfija

- ✓ Suponemos EXIN una expresión aritmética escrita en notación infija
- ✓ EXIN puede tener:
  - o Paréntesis izquierdos y derechos
  - o Operandos (dígitos 0-9 y letras A-Z)
  - o Operadores (^ potencia, * multiplicación, / división, + suma, - resta, de acuerdo con sus prioridades, para operadores de un mismo nivel se ejecutan de izquierda a derecha).
- ✓ El algoritmo encuentra la Expresión Postfija (EXPO) equivalente, haciendo uso de una PILA como intermediario de los operadores. El algoritmo termina cuando la PILA está vacía.

```
INICIO
   METER "(" en PILA/* Paréntesis izquierdo */
   AÑADIR ")" en EXIN    /* Al final */
   Examinar EXIN (de izquierda a derecha)
          hasta que PILA este vacía
     Si  se encuentra un operando AÑADIR en EXPO  fin_si
     Si
       se encuentra un paréntesis izquierdo METER en PILA
     fin_si
```

Si se encuentra un operador **entonces**
a) Repetidamente SACAR de PILA y AÑADIR a EXPO cada operador (de lo alto de la PILA) que tenga la misma precedencia o mayor que operador.
b) METER operador a PILA
fin_si

Si se encuentra un paréntesis derecho **entonces**
c) Repetidamente SACAR de PILA y AÑADIR a EXPO cada operador (de lo alto de la PILA, hasta que se encuentre un paréntesis izquierdo.
d) Eliminar el paréntesis izquierdo (no añadirlo a EXPO)
fin_si

fin_examinar

FIN

Prueba de escritorio

$$(A+B)\frac{D^E}{F}$$ EXIN = (A+B)*D^E/F

Índice	1	2	3	4	5	6	7	8	9	10	11	12
EXIN	(	A	+	B	)	*	D	^	E	/	F	)

Añadir bandera (Primer paso)

PASO	*EXIN*	*PILA*	*EXPO*
1	(	((	
2	A	((	A
3	+	((+	A
4	B	((+	AB
5	)	(	AB+
6	*	(*	AB+
7	D	(*	AB+D
8	^	(*^	AB+D
9	E	(*^	AB+DE
10	/	(/	AB+DE^*
11	F	(/	AB+DE^*F
12	)	Pila Vacía	AB+DE^*F/

Algoritmo de Evaluación de Expresiones Postfijas

VALOR = Resultado de la Evaluación de expresión aritmética postfija

EXPO = Expresión aritmética en notación postfija

PILA = Intermediario, mantiene operandos.

```
INICIO
   AÑADIR paréntesis derecho ")" al final de EXPO
   Examinar EXPO (de izquierda derecha)
        hasta que se encuentre ")"
     Si se encuentra un operando entonces
        METER en PILA
     fin_si
     Si se encuentra un operador entonces
        a) SACAR los 2 elementos superiores de la PILA (
           donde A sea el elemento superior y B el
           siguiente)
        b) Evaluar B operador A
        c) METER el resultado de (b) en PILA
     fin_si
   fin_examinar
   Hacer VALOR igual al elemento superior de la PILA
FIN
```

Prueba de Escritorio: EXPO = 234567+*+*+

Índice	1	2	3	4	5	6	7	8	9	10	11	12
EXPO	2	3	4	5	6	7	+	*	+	*	+	)

Añadir bandera (Primer paso)

PASO	EXPO	A	B	PILA
1	2			2
2	3			2, 3
3	4			2, 3, 4
4	5			2, 3, 4, 5
5	6			2, 3, 4, 5, 6
6	7			2, 3, 4, 5, 6, 7
7	+	7	6	2, 3, 4, 5, 13
8	*	13	5	2, 3, 4, 65
9	+	65	4	2, 3, 69
10	*	69	3	2, 207
11	+	207	2	209
12	)			

VALOR

Semana 08: Algoritmos con Pilas

1. Codificar el algoritmo de Transformación de una expresión Infija a Expresión Postfija en la herramienta de programación de laboratorio.

2. Codificar el algoritmo de Evaluación de Expresiones Postfijas en la herramienta de programación de laboratorio.

3. Cierto número de usuarios, n, envían simultáneamente un documento a la impresora común, la cual debe determinar su orden de impresión. Las longitudes de los documentos enviados son l1...ln, siendo li la longitud del documento enviado por el usuario i (la numeración de los usuarios es arbitraria). Suponiendo que el tiempo que se tarda en imprimir un documento es proporcional a su longitud, Escribir el algoritmo (Pseudo código) que indique el orden óptimo en que se deben imprimir de manera que se minimice el tiempo medio de espera de cada usuario. El tiempo de espera del usuario i-ésimo vendrá dado por el orden que haya establecido la impresora para su documento. Si su documento es el j-ésimo en imprimirse, su tiempo de espera será la suma de los tiempos de impresión de los j primeros documentos según ese orden (se incluye el suyo en la suma). Los usuarios debe estar enterados del tiempo que les tomará esperar.

4. Dados los caracteres (), [] y {}, y una cadena s; s esta balanceada si tiene alguno de estos formatos, s = "", (string nulo), s = (T), s = [T], s = {T}, s = TU en donde T y U son cadenas balanceadas (en otras palabras, para cada paréntesis, llave o corchete abierto existe un carácter de cierre correspondiente). Ejemplo {(a+b) [(c-d) ^2]}. Escribir el Algoritmo (Pseudo código) que use una PILA para ver si una cadena es balanceada.

5. Escribir el algoritmo (Pseudo código) que maneje tres pilas de DATOS (A, B y C) en un solo arreglo implementado como lista enlazada. Las pilas pueden decrecer o crecer en cada momento, pero el tamaño del arreglo no variará. Si una pila necesita más espacio solo tiene que tomarlo del arreglo que tiene 15 lugares en función a su lista de disponibles, si las 3 pilas completan los 15 lugares entonces se manda un mensaje de pila llena. Al ingresar un dato se deberá leer además el nombre de pila donde se desea colocarla. Contemplar la posibilidad de eliminar un dato de cualquier pila, lo que provoca incrementar la lista de disponibles.

6. En la SUNAT se considera una cola frente a una ventanilla en la cual si un usuario al llegar a su momento de atención no puede ser atendido (por que se olvidó un documento de fácil subsanación), se le reintegra a una cola en ventanilla especial en la posición Nº 5 si solo se olvido la copia de su DNI o en la posición N°10 si acaso hay más de 10 personas y se olvido llenar su formulario de Impuestos, o al final de la misma, en caso contrario. Se pide diseñar un procedimiento de MESAPARTES (lo que debe suceder cuando un cliente es atendido), ESPECIAL (vuelta a la cola en ventanilla especial) y otro de ENTRADA (inicial) en cola. Todo ello con vistas a que el reingreso sea sencillo, para lo que se sugiere controlar el Nº de personas en la cola, y en su caso, un puntero al elemento adecuado de la cola para ejecutar el reingreso. Efectúe la prueba de escritorio correspondiente .

6. Se tiene la siguiente fórmula: $$X = \frac{a + \sqrt{b^2 - c}}{\sqrt[3]{d} - \frac{e}{f}}$$

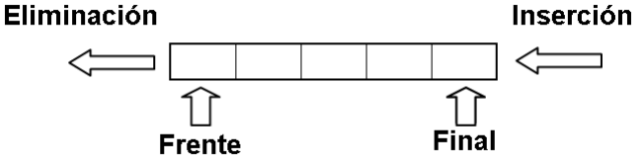
   Escribir su forma infija, prefija y postfija correspondiente.

7. Se tiene una pila con los primeros 10 números naturales, y una cola con los 10 siguientes, escriba el algoritmo en pseudo código que utilizando las operaciones básicas para cada una de estas estructuras permita obtener la sumatoria del producto de sus datos, atendiendo la regla LIFO para la pila y FIFO para la cola.

8. Dados los caracteres **(), [] y {},** y una cadena s; s esta balanceada si tiene alguno de estos formatos, **s = ""**, (string nulo), **s = (T), s = [T], s = {T}, s = TU** en donde T y U son cadenas balanceadas (en otras palabras, para cada paréntesis, llave o corchete abierto existe un carácter de cierre correspondiente). Ejemplo **s ={[(c-d) ^2]+ 1}**. Escribir el Algoritmo (Pseudo código) que use una PILA para ver si una cadena es balanceada.

Semana 09: Estructura Lineal: Colas. Operaciones Básicas.

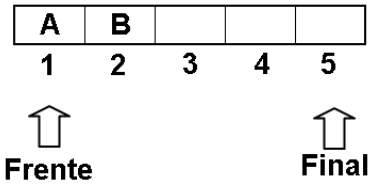
Una cola es una lista lineal en la cual las eliminaciones se realizan solo por un extremo llamado frente y las inserciones se realizan por el otro extremo llamado final.
Las colas también se denominan FIFO (first in first out), primero en entrar, primero en salir, porque el primer elemento en ingresar a la cola será el primero en salir.
Ejemplos de colas en la vida diaria, se ven en los cines, en los bancos, estadios, etc.
En informática hablamos de colas de prioridades, y podemos ver las colas de impresión para múltiples trabajos que tienen que esperar que de acuerdo al orden en que se dio la orden de impresión.
Gráficamente una cola se representa:



Algoritmo de Inserción en Colas

```
{ frente ? 0, final ? 0 }
Inicio
  Leer X; {Elemento a insertar]
  Si FINAL = N entonces
     Escribir "Cola Llena"
  Caso contrario
     FINAL ? FINAL + 1
     COLA(final) ? X
  Fin _ si
  Si FRENTE = 0 entonces
     FRENTE ? 1
  Fin _ si
Fin
```

Prueba de escritorio



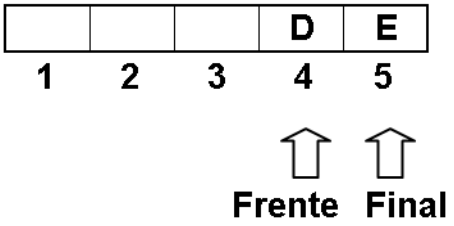
N	X	FRENTE	FINAL	COLA[FINAL]
5		0	0	
	A	1	1	A
	B	1	2	B
	C	1	3	C
	D	1	4	D
	E	1	5	E

Variante algoritmo anterior

```
{ frente ? 0, final ? 0 }
Inicio
  Leer X; {Elemento a insertar]
  Si FINAL = N entonces
     Escribir "Cola Llena"
  Caso contrario
     FINAL ? FINAL + 1
     COLA(final) ? X
     Si FRENTE = 0 entonces
          FRENTE ? 1
     Fin _ si
  Fin _ si
Fin
```

Algoritmo de Eliminación en Colas

```
Inicio
  Si FRENTE = 0 entonces
     Escribir "Cola vacía"
  Caso contrario
     X ? COLA(FRENTE)
     Si FRENTE = FINAL entonces
          FRENTE ? 0
          FINAL ? 0
     Caso contrario
          FRENTE ? FRENTE + 1
     Fin _ si
  Fin _ si
Fin
```

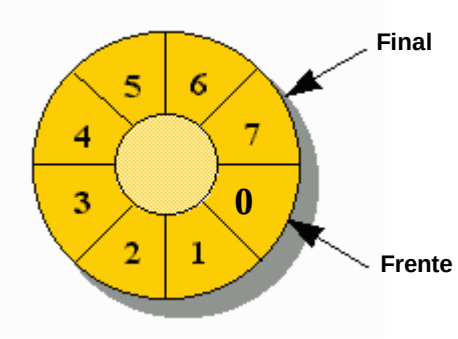


Prueba de escritorio

N	FRENTE	FINAL	COLA[FRENTE]	X
5	1	5	A	A
	2	5	B	B
	3	5	C	C
	4	5	D	D
	5	5	E	E
	0	0		

COLAS CIRCULARES

Los datos se van agregando y cuando tail (Cola) llegue al máximo se reasigna al inicio, y al dar de baja lo mismo sucede con head (cabecera), se va decrementando y cuando llegue al inicio se le reasigna al final de la cola.



BICOLAS

Una bicola o cola bidireccional, es una lista lineal en la que los elementos que se pueden añadir o quitar por cualquier extremo. Hay variantes:

- Bicolas de entrada restringida: Son aquellas donde la inserción sólo se hace por el final, aunque podemos eliminar al principio ó al final.
- Bicolas de salida restringida: Son aquellas donde sólo se elimina por el final, aunque se puede insertar al principio y al final.

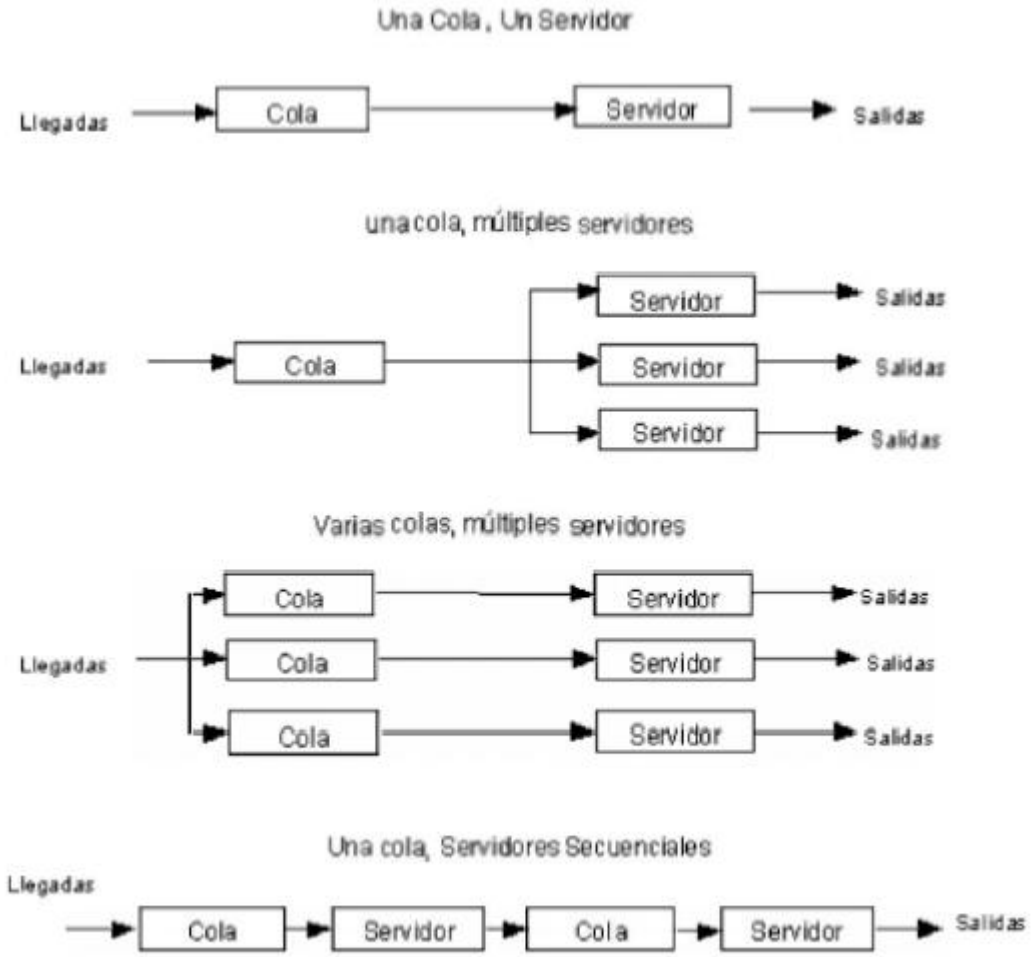
COLAS DE PRIORIDADES

Vienen a ser un conjunto de elementos a los cuales se les ha asignado una prioridad y de forma que el orden de en que los elementos son eliminados y procesados cumplen las siguientes reglas:

1. Un elemento de mayor prioridad es procesado de acuerdo al orden en que fueron añadidos a la cola.
2. Dos elementos con la misma prioridad son procesados de acuerdo con el orden en que fueron añadidos a la cola.

Ejemplo.- un sistema de tiempo compartido, donde los programas de mayor prioridad se procesan primero y los programas de igual prioridad forman una cola estándar

.



Semana 10: Algoritmos con COLAS. Aplicaciones

Instrucciones.- *Implementar los siguientes ejercicios la revisión de los mismos será en el Laboratorio utilice como lenguaje de programación JAVA*

1. Se considera una cola frente a una ventanilla en la cual si un cliente al llegar a la misma no puede ser atendido, se le reintegra a la cola a la posición nº 10, si hay más de 10 personas, o al final de la misma, en caso contrario. Se pide diseñar un procedimiento de ATENCIÓN (lo que debe suceder cuando un cliente es atendido), REINGRESO (vuelta a la cola) y otro de INGRESO (inicial) en cola. Todo ello con vistas a que el reingreso sea sencillo, para lo que se sugiere controlar el nº de personas en la cola, y en su caso, un puntero al elemento adecuado de la cola para ejecutar el reingreso.

2. Un ascensor está situado en un edificio de N plantas (la planta baja es la 0) y responde a dos tipos de peticiones de funcionamiento: las de los usuarios que están dentro del ascensor (peticiones internas) y las de los que están fuera (peticiones externas). Estas últimas sólo serán atendidas cuando no haya peticiones internas. Los dos tipos de peticiones se gestionan independientemente y responden a criterios de temporalidad, de manera que las llamadas van siendo atendidas según el orden de solicitud. Con este planteamiento, hay que escribir un programa que simule el funcionamiento del ascensor de forma que las peticiones se realicen mediante pulsaciones del teclado y, "al mismo tiempo", mover el ascensor a los pisos que se van solicitando. Se distinguirá entre peticiones interiores y exteriores según las teclas pulsadas (teclas: 0, 1, 2, .. N, indican peticiones interiores a dichos pisos; teclas: -0, -1, -2, ..., -N, indican peticiones exteriores en el mismo orden), la tecla 'N+1' se reserva para finalizar la simulación. La visualización del proceso se puede realizar utilizando un método MOVER_ASCENSOR que muestra por pantalla el movimiento del ascensor. Este método responde al siguiente perfil: MOVER_ASCENSOR (desde, hasta: 0..N); donde desde indica la posición actual del ascensor y hasta, el piso al que debe ir.

3. Considera que palabra es una variable de tipo Cola que contiene la entrada del usuario por teclado, P una pila de caracteres y el siguiente algoritmo:

   1. mientras haya más caracteres en palabra hacer
   2. apilar el primero de la cola en la pila P
   3. sacar de la cola

4. Fin_Mientras
5. Mientras la pila P no sea vacía
6. Escribir la cima de P
7. Desapilar de P
8. Fin_Mientras

¿Cuál es la salida para la entrada "examen"?

4. Escribe el algoritmo en pseudocódigo que lea una cadena de caracteres del teclado y decida si es palíndromo, es decir, si se lee igual de izquierda a derecha que de derecha a izquierda. Impleméntalo después en el Lenguaje de programación de su dominio. Ejemplo: daba le arroz a la zorra el abad es palíndromo

5. Un estacionamiento de las avionetas de un aeródromo es en línea, con una capacidad hasta 12 avionetas. Las avionetas llegan por el extremo izquierdo y salen por el extremo derecho. Cuando llega un piloto a recoger su avioneta, si ésta no está justamente en el extremo de salida (derecho), todas las avionetas a su derecha han de ser retiradas, sacar la suya y las retiradas colocadas de nuevo en el mismo orden relativo en que estaban. La salida de una avioneta supone que las demás se mueven hacia adelante, de tal forma que los espacios libres del estacionamiento estén por la parte izquierda. Escriba el Algoritmo (Pseudocódigo) para emular este estacionamiento tiene como entrada un carácter que indica una acción sobre la avioneta, y la matricula de la avioneta. La acción puede ser, llegada (E) o salida (S) de avioneta, En la llegada puede ocurrir que el estacionamiento esté lleno, si es así la avioneta espera hasta que quede una plaza libre, o hasta que se dé la orden de retirada (salida).

6. Elabore el pseudocódigo de las operaciones METER y SACAR de una bicola, donde la cola de un extremo guarda códigos de Radios y la del otro extremo códigos de Televisores

7. En un archivo de texto se encuentran los resultados de una competición de tiro al plato, de tal forma que en cada línea se encuentra Apellido, nombre, número de dorsal y número de platos rotos. Se debe escribir el Algoritmo (Pseudocódigo), que lea el archivo de la competición y determine los tres primeros. La salida ha de ser

los tres ganadores y a continuación los concursantes en el orden en que aparecen en el archivo (utilizar la estructura cola).

8. El despegue de aeronaves en un aeropuerto se realiza siguiendo el orden establecido por una cola de prioridades. Hay 5 prioridades establecidas según el destino de la aeronave. Destinos de menos de 500 km tienen la máxima prioridad, prioridad 1, entre 500 y 800 km prioridad 2, entre 801 y 1000 km prioridad 3, entre 1001 y 1350 km prioridad 4 y para mayores distancias prioridad 5. Cuando una aeronave recibe cierta señal se coloca en la cola que le corresponde y empieza a contar el tiempo de espera. Los despegues se realizan cada 6 minutos según el orden establecido en las distintas colas de prioridad. El piloto de una aeronave puede pasar el aviso a control de que tiene un problema, y no puede despegar por lo que pasa al final de la cola y se da la orden de despegue a la siguiente aeronave. Puede darse la circunstancia de que una aeronave lleve más de 20 minutos esperando, en ese caso pasará a formar parte de la siguiente cola de prioridad y su tiempo de espera se inicializa a cero.
   Escribir el algoritmo que simule este sistema de colas mediante una lista única, cada vez que despegue un avión saldrá un mensaje con las características del vuelo y el tiempo total de espera.

SEMANA 11: Examen de 2da Unidad

Semana 12: Estructura No Lineal: Arboles. Definición. Operaciones Básicas.

Hasta ahora se han estudiado estructuras lineales (estáticas o dinámicas) donde un elemento le sigue a otro. Los árboles representan estructuras no lineales y dinámicas. Se dice dinámica puesto que la estructura del árbol puede variar durante la ejecución del programa. No lineales porque a cada elemento del árbol puede seguirle varios elementos.

Generalidades

Un árbol es una estructura jerárquica aplicada sobre una colección de elementos u objetos llamados nodos; uno del os cuales se denomina raíz.

Formalmente se define un árbol de tipo **T**, como una estructura homogénea que es la concatenación de un elemento de tipo **T** junto con un número finito de árboles disjuntos llamados subárboles. Una forma particular de árbol puede ser la estructura vacía.

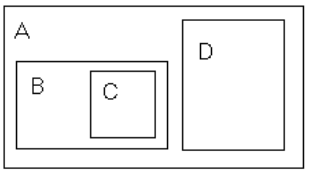
Aplicaciones

Se le utiliza para representar:

- Fórmulas matemáticas
- Registrar la historia de un campeonato
- Construir un árbol genealógico
- Análisis de circuitos eléctricos
- Numerar los capítulos y secciones de un libro.

Representación

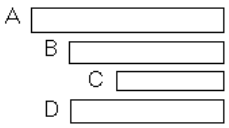
a) Diagramas de Venn



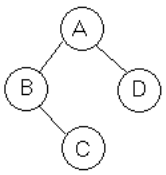
b) Anidación de paréntesis
(A((B(C))D))

c) Por notación decimal
1.A, 1.1.B, 1.1.1.C, 1.2D

d) Notación indentada

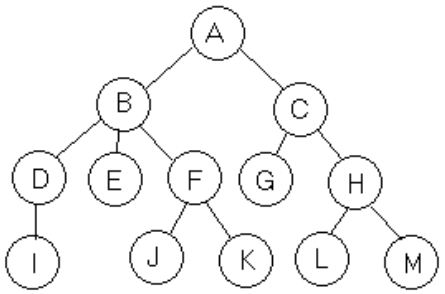


e) Grafos



Características y Propiedades de los Árboles

a) Todo árbol que no es vacío, tiene un único nodo raíz
b) Sí un nodo X es descendiente de un nodo Y, decimos que X es hijo de Y
c) Sí un nodo X es antecesor directo de un nodo Y, decimos que X es padre de Y
d) Todos los nodos de un mismo padre, son hermanos
e) Todo nodo que no tiene hijos es un nodo terminal
f) Todo nodo que no es raíz, ni terminal, es un nodo interior
g) Grado es el número de descendientes directos de un determinado nodo. Grado de un árbol es el máximo grado de todos los nodos de un árbol.
h) Nivel es el número de nodos que deben ser recorridos para llegar a un determinado nodo (desde el raíz)
i) Altura del árbol es el máximo número de niveles de entre todos las ramas del árbol más 1.



Raíz =
Hermanos =
Terminales =
Interiores =
Grado =
Grado del árbol =
Nivel = Altura =

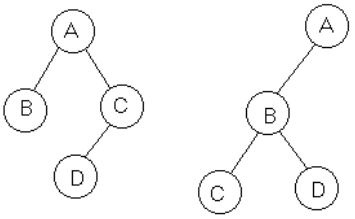
ÁRBOLES BINARIOS

Un árbol de grado 2, es un árbol donde cada nodo puede tener como máximo 2 subárboles, siendo necesario distinguir entre el subárbol derecho y el subárbol izquierdo. Los árboles binarios tienen múltiples aplicaciones:

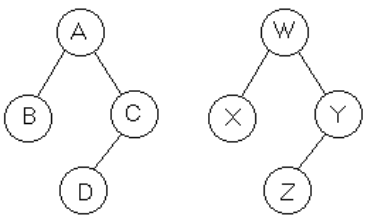
- Representa la historia de un campeonato donde hay un ganador, 2 finalistas 4 semifinalistas.
- Representa expresiones algebraicas construidas con operadores binarios.

Árboles Binarios: Distintos, Similares y Equivalentes

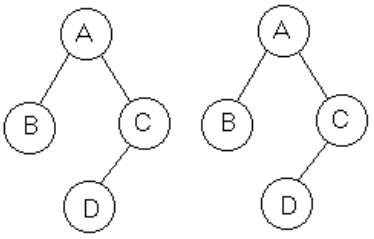
Son distintos cuando sus estructuras son diferentes



Son similares cuando sus estructuras son idénticas, pero la información que contienen sus nodos difieren entre sí.

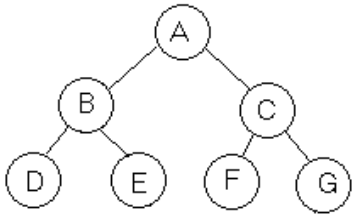


Son equivalentes cuando son similares y además los nodos contienen la misma información.



Árboles Binarios Completos

Aquellos en el que todos sus nodos excepto los terminales tiene 2 hijos.



Número de nodos = $2^h - 1$
=

Número de Nodos por nivel = 2^{h-1}
=

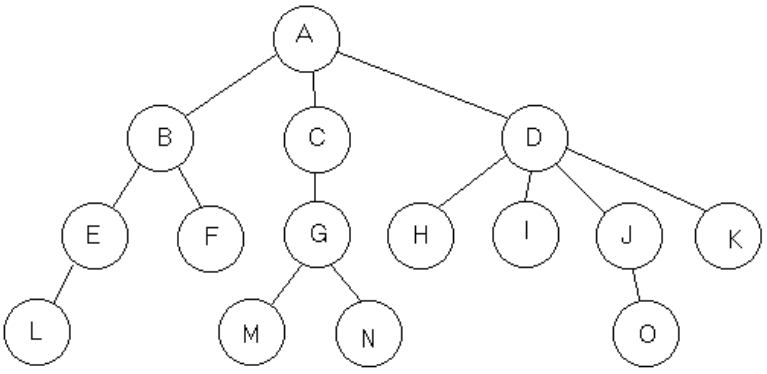
Conversión de un Árbol General en Árbol Binario

Los árboles binarios es la estructura de datos fundamental en la teoría de árboles. Por lo que estableceremos los mecanismos para convertir un árbol general en un árbol binario, ya que los árboles binarios son más fáciles de programar que los árboles generales

Pasos de Conversión

1. Deben enlazarse los hijos de cada nodo en forma horizontal (los hermanos)
2. Debe enlazarse en forma vertical el nodo padre con el hijo que se encuentra más a la izquierda. Además debe eliminarse el vínculo de ese padre con el resto de sus hijos.
3. Rotar el diagrama resultante, aproximadamente 45 grados hacia la izquierda obteniéndose el árbol binario correspondiente.

Ejemplo:



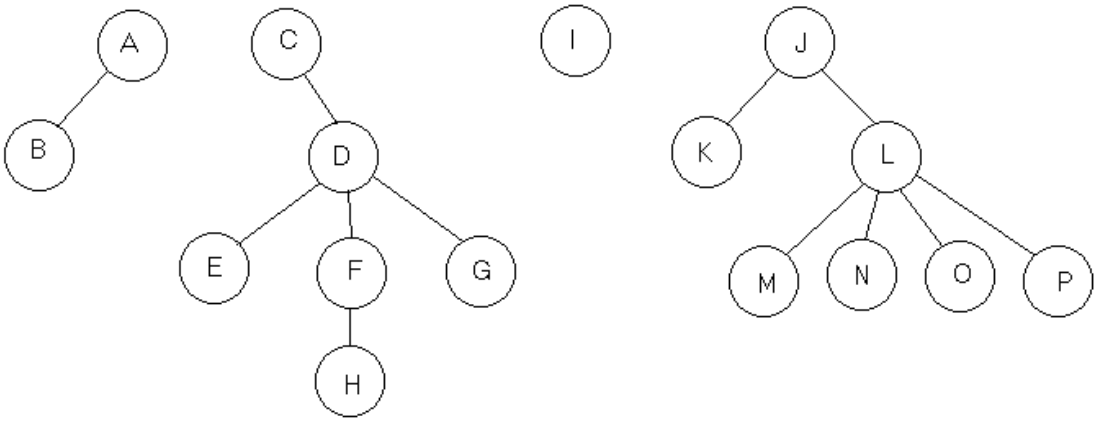
Resultados:

Conversión de un Bosque como Árbol Binario

Pasos de Conversión

1. Deben enlazarse en forma horizontal las raíces de los distintos árboles generales
2. Enlazar los hijos de cada nodo en forma horizontal (los hermanos)
3. Debe enlazarse en forma vertical el nodo padre con el hijo que se encuentre más a la izquierda. Además debe eliminarse el vínculo de ese padre con el resto de sus hijos.
4. Debe rotarse el diagrama resultante 45 grados hacía la izquierda y se obtendrá el árbol binario correspondiente.

Ejemplo:



REPRESENTACION DE LOS ÁRBOLES BINARIOS

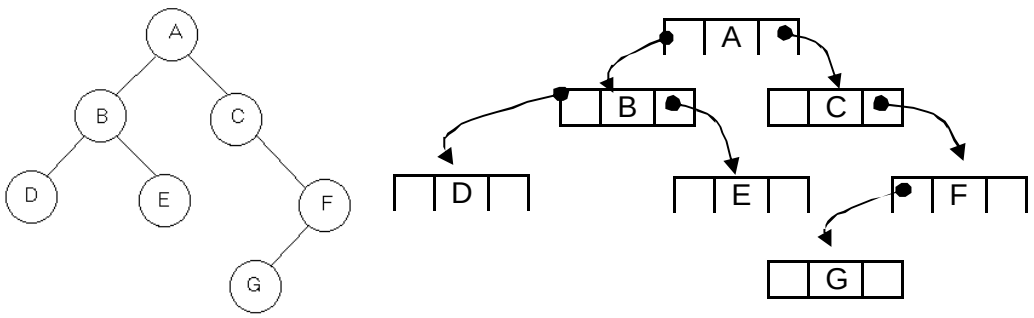
Existen 2 formas tradicionales de representar un árbol binario en memoria.

1. Por medio de datos tipo puntero (variables dinámicas)
2. Con arreglos o listas enlazadas

Cada nodo tiene la siguiente estructura:

IZQ	INFO	DER

Representación Enlazada de Árboles Binarios



Recorridos en Árboles Binarios

Recorrer significa visitar los nodos del árbol en forma sistemática, de tal manera que todos los nodos del mismo sean visitados una sola vez

Existen 3 formas diferentes de efectuar un recorrido:

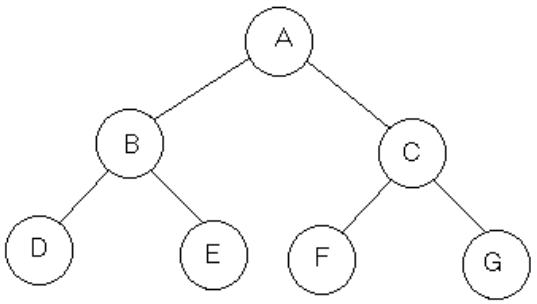
1. PREORDEN
   a. Visitar raíz
   b. Recorrer subárbol izquierdo
   c. Recorrer subárbol derecho

2. INORDEN
   a. Recorrer subárbol izquierdo
   b. Visitar raíz
   c. Recorrer subárbol derecho

3. POSTORDEN
   a. Recorrer subárbol izquierdo
   b. Recorrer subárbol derecho
   c. Visitar raíz

- El recorrido preorden se obtiene viajando hacia abajo por la rama más a la izquierda, hasta que se encuentre el nodo terminal, entonces se vuelve hacia atrás hasta la siguiente rama y así sucesivamente, en este recorrido el nodo terminal más a la derecha se inspecciona último.
- En el recorrido postorden cada descendiente de un nodo N es procesado antes que el nodo N.
- Los nodos terminales siempre se recorren en el mismo orden de izquierda a derecha (en todos los recorridos).

Ejemplo

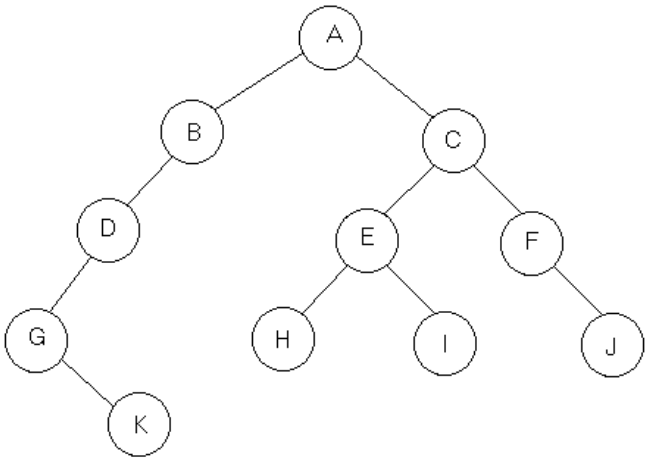


a) Preorden

b) Inorden

c) Postorden

Ejercicio:



a) Preorden

b) Inorden

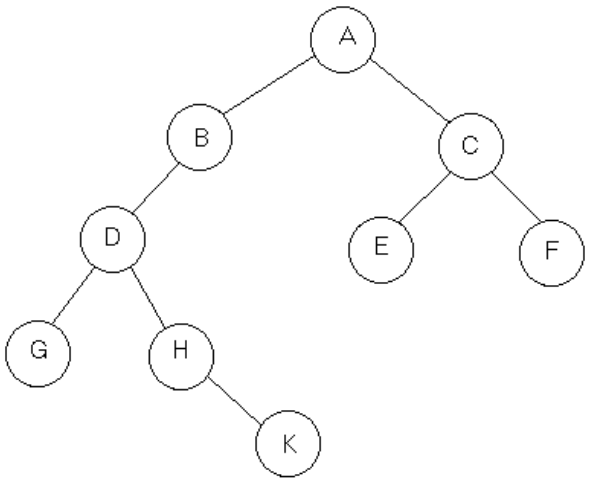
c) Postorden

Algoritmo de Recorrido PreOrden
(Empleando Pilas)

Suponga un árbol binario T representado por arrays lineales enlazados INFO, IZQ, DER. Emplear array PILA para mantener temporalmente las direcciones de los nodos.
PTR = puntero del árbol; SUP = puntero de la PILA

```
1)   Inicio
2)        SUP≠ 1, PILA[SUP] ≠ 0, PTR≠ RAIZ
3)        MIENTRAS PTR<>0
         Escribir INFO[PTR]
         SI DER[PTR]<>0
              ENTONCES
                   SUP≠ SUP+1; PILA[SUP] ≠ DER[PTR]
         FIN_si
         SI IZQ[PTR]<>0
              ENTONCES  PTR ≠ IZQ[PTR]
              SINO PTR ≠ PILA[SUP];  SUP≠ SUP-1
         FIN_si
         FIN_mientras
4)   FIN
```

		INFO	IZQ	DER	PILA
	1	D	9	11	
	2				
	3	B	1	0	
	4				
RAIZ---->	5	A	3	7	
	6	E	0	0	
	7	C	6	8	
	8	F	0	0	
	9	G	0	0	
	10				
	11	H	0	12	
	12	K	0	0	
	13				



ABDGHKCEF

Algoritmo de Recorrido InOrden
(Empleando Pilas)

```
1)   Inicio
2)         SUP≠ 1, PILA[SUP] ≠ 0, PTR≠ RAIZ
3)         MIENTRAS PTR<>0 hacer
             SUP≠ SUP+1; PILA[SUP] ≠ PTR
             PTR≠ IZQ[PTR]
         FIN_mientras
4)         PTR≠ PILA[SUP];  SUP≠ SUP-1
5)         MIENTRAS PTR<>0 hacer
                    Escribir INFO[PTR]
             SI DER[PTR]<>0
                 ENTONCES   PTR≠ DER[PTR];  Ir al paso(3)
             FIN_si
                    PTR≠ PILA[SUP];  SUP≠ SUP-1
         FIN_mientras
6)   Fin
```

GDHKBAECF

Algoritmo de Recorrido PostOrden
(empleando Pilas)

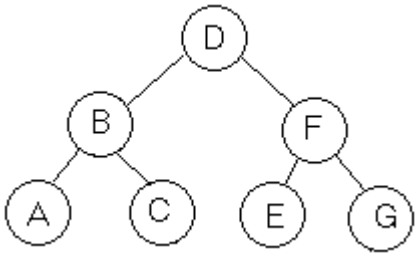
```
1)   Inicio
2)         SUP≠ 1, PILA[SUP] ≠ 0, PTR≠ RAIZ
3)         MIENTRAS PTR<>0 hacer
             SUP≠ SUP+1; PILA[SUP] ≠ PTR
             SI DER[PTR]<>0
                  ENTONCES
                  SUP≠ SUP+1; PILA[SUP] ≠ -DER[PTR]
             FIN_si
             PTR≠ IZQ[PTR]
         FIN_mientras
4)         PTR≠ PILA[SUP];  SUP≠ SUP-1
5)         MIENTRAS PTR>0 hacer
                    Escribir INFO[PTR]
                    PTR≠ PILA[SUP];  SUP≠ SUP-1
         FIN_mientras
6)      SI PTR<0
             ENTONCES  PTR≠ -PTR;  Ir al paso (3)
        FiN_si

7)   Fin
```

Semana 13: Aplicaciones de Arboles

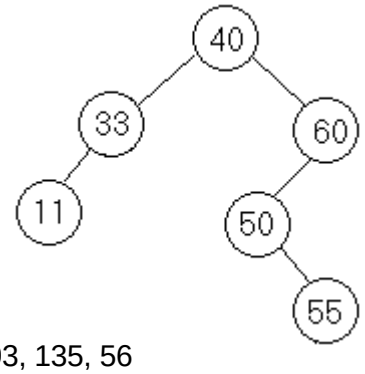
ARBOLES BINARIOS DE BUSQUEDA

Es una estructura sobre la cual se pueden realizar eficientemente las operaciones de búsqueda, inserción y eliminación. Al comparar notamos que en un arreglo lineal la localización de los datos se puede realizar rápidamente siempre y cuando el arreglo este ordenado, no sucede lo mismo con las operaciones de inserción y eliminación que resultan costosas (por el tiempo que consumen); En una lista enlazada las inserciones y las eliminaciones son fáciles pero una búsqueda puede resultar costosa.
Un Árbol Binario de Búsqueda o Árbol Binario Ordenado, es uno en el cual para todo nodo T del árbol, debe cumplirse que todos los valores de los nodos del subárbol izquierdo de T deben ser menores o iguales al valor del nodo T; de igual manera todos los valores de los nodos del subárbol derecho de T deben ser mayores al valor del nodo T.

Ejemplo 01: DFEBACG



Ejemplo 02: 40, 60, 50, 33, 55, 11



Ejercicio_01: 120, 87, 140, 43, 99, 130, 22, 65, 93, 135, 56

Ejercicio_02: J, R, D, G, T, E, M, H, P, A, F, Q

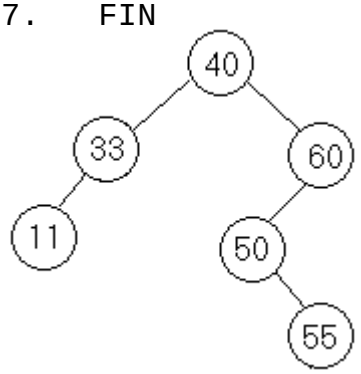
Algoritmo de Búsqueda en Árbol Binario Ordenado

Para un árbol binario ordenado representado por los arreglos INFO, IZQ, DER. El siguiente algoritmo encuentra la posición POS del elemento y la posición del padre (PAD).

Se debe tener en cuenta lo siguiente:

1. Si POS = 0 y PAD = 0 El árbol está vacío
2. Si POS <> 0 y PAD = 0 El elemento es la raíz del árbol
3. Si POS = 0 y PAD <> 0 Elemento no está en el árbol, pero puede ser añadido al árbol como hijo del nodo de posición PAD

```
1. Inicio
2. Si RAIZ = 0 entonces
     POS ? 0
     PAD ? 0                    (Árbol vacío)
     Salir
       Fin_si
3. Si ELEMENTO = INFO[RAIZ] entonces
     POS ? RAIZ
     PAD ? 0                    (Elemento está en la raíz)
     Salir
   Fin_si
4. Si ELEMENTO < INFO[RAIZ] entonces
          PTR ? IZQ[RAIZ]; SALVA ? RAIZ
     Caso contrario                (Inicializa PTR y SALVA)
          PTR ? DER[RAIZ]; SALVA ? RAIZ
   Fin_si

5. Mientras PTR<>0 hacer
     Si ELEMENTO = INFO[PTR] entonces
          POS ? PTR; PAD ? SALVA
          Salir
     Fin_si                                  (Busca
elemento)
     Si ELEMENTO < INFO[PTR] entonces
          SALVA ? PTR; PTR ? IZQ[PTR]
     Caso contrario
          SALVA ? PTR; PTR ? DER[PTR]
     Fin_si
       Fin_mientras

6.   POS ? 0; PAD ? SALVA          (Búsqueda sin resultado)

7.   FIN
```



RAIZ---->

	INFO	IZQ	DER
1			
2	55	0	0
3	50	0	2
4	60	3	0
5	40	6	4
6	33	7	0
7	11	0	0
8			

RAIZ = 5
Buscar 50

PTR	SALVA	INFO[PTR]	POS	PAD

Algoritmo de Inserción en Árbol Binario Ordenado

(Añade elemento en posición ordenada)

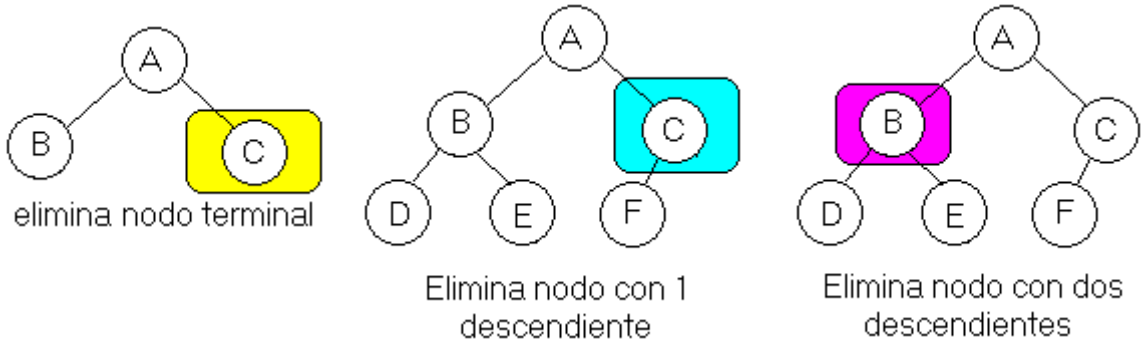
```
1) Inicio
2) Si POS <>O entonces
     Salir                          ------------------------------------
   Fin_si
3) Si DISPO = 0 entonces
     Escribir "desbordamiento"
     Salir
   Fin_si
4) Nuevo ? DISP; DISP ? IZQ[DISP]
   INFO[Nuevo] ? elemento
   POS ? Nuevo                      ------------------------------------
   IZQ[Nuevo] ? 0; DER[Nuevo] ? 0;
5) Si PAD = 0 entonces
          RAIZ  ? nuevo
     Caso contrario
          Si elemento < INFO[PAD] entonces
               IZQ[PAD] ? nuevo
          Caso contrario            ------------------------------------
               DER[PAD] ? nuevo
          Fin_si
6) Fin
```

POS = 0 PAD = 6 DISP = 1 elemento = 35

Eliminación de un Elemento

La eliminación debe conservar el orden de los elementos de l árbol, se tiene en consideración los siguientes casos:

- ✓ El elemento es un nodo terminal (o descendientes)
- ✓ El elemento tiene un descendiente
- ✓ El elemento tiene dos descendientes



ALGORITMO CASO 1 (Nodo terminal)

```
1.   INICIO
2.        Si IZQ[POS] = 0   y   DER[POS] = 0 entonces
               hijo  ? 0
          caso contrario
               Si IZQ[POS] <> 0 entonces
                    hijo  ? IZQ[POS]
               caso contrario
                    hijo  ? DER[POS]
               fin_si
          fin_si
3.        Si PAD<> 0 entonces
               Si POS = IZQ[PAD] entonces
                    IZQ[PAD] ? hijo
               Caso contrario
                    DER[PAD] ? hijo
               Fin_si
          Caso contrario
               RAIZ ? hijo
          Fin_si
4.        Salir
5.   FIN
```

ALGORITMO CASO 2 (Nodo que tiene dos hijos)

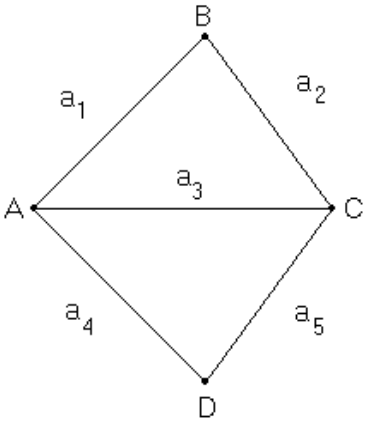
Elimina nodo N que tiene dos hijos, de la posición POS. El puntero PAD tiene la posición del Padre de N. Si PAD = 0 N es el nodo RAIZ.

SUC puntero que da la posición del sucesor inorden de N.
PADSUC puntero que da la posición del padre del sucesor inorden.

```
1.   INICIO
2.        PTR  ? DER[POS];    SALVA  ? POS
          Mientras  IZQ[PTR] <> 0 hacer
               SALVA  ? PTR
               PTR  ? IZQ[PTR]
          Fin_mientras
          SUC  ? PTR;   PADSUC  ? SALVA
3.        Eliminar sucesor inorden con algoritmo Caso 1
4.        Si PAD<> 0 entonces
               Si POS = IZQ[PAD] entonces
                    IZQ[PAD] ? SUC
               Caso contrario
                    DER[PAD] ? SUC
               Fin_si
          Caso contrario
               RAIZ ? SUC
          Fin_si
          IZQ[SUC]  ? IZQ[POS]
          DER[SUC]  ? DER[POS]
5.   FIN
```

Semana 14: Estructura No Lineal: GRAFOS
Definición. Operaciones Básicas.

Los grafos son estructuras de datos no lineal (*Los árboles binarios pueden representar estructuras jerárquicas con limitaciones de 2 subárboles por nodo. La limitación que puede ser superada por un grafo)*, consiste en un conjunto N de elementos llamados nodos (n), también se les conoce como puntos o vértices; Así como también por un conjunto A de líneas que unen un elemento con otro y se denominan aristas (a), cada arista se identifica por un único par ordenado [u, v], donde **u** y **v** son los extremos adyacentes de a.



Orden.- Número de elementos del grafo (N_G= 4)
N_G= { A, B, C, D}

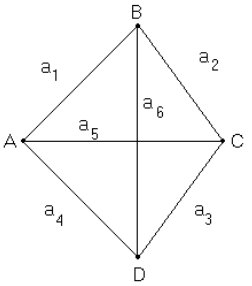
Grado.- Número de aristas que contiene el nodo
Ejemplo:
Grado (A): 3 grado (B): 2

Camino.- Es la secuencia de nodos que unen n nodo "u" con un nodo "v". Camino Simple si todos los nodos son distintos.

Ciclo.- Es un camino simple cerrado de longitud 3 ó más. Un ciclo de longitud k se llama k-ciclo.

Grafo conexo.- Es conexo, si y solo sí existe un camino simple entre cualquier par de nodos de G.

Grafo completo.- cuando cada nodo de G es adyacente a todos los demás nodos.

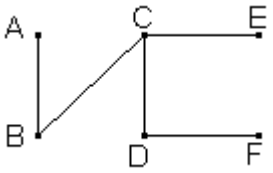


*Un grafo completo de **n** nodos tiene:*

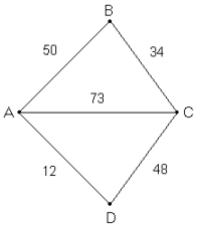
n(n-1)/2 aristas

E*jemplo:* *n = 4 nodos* ----->
Aplicando fórmula = 6 aristas

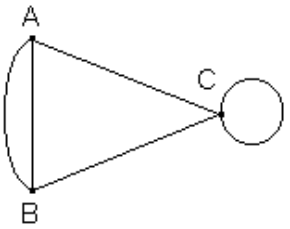
Grafo árbol.- En el cual existe un único camino simple entre cada dos nodos.



Grafo etiquetado.- Si sus aristas tienen datos asignados a los cuales se les denomina pesos.



Multígrafo.- Es aquel que permite aristas múltiples es decir que dos nodos tengan más de una arista y permite los bucles es decir cuando mas de una arista tiene los mismos



GRAFO DIRIGIDO.- Es aquel en el cual cada arista tiene una dirección asignada, es decir cada arista se identifica con un par ordenado (u, v) en vez del par desordenado [u, v]

Esto significa que:

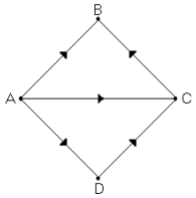
- Cada nodo empieza en "u" y termina en "v"
- "u" es el origen y "v" es el destino
- "u" es el predecesor y "v" es el sucesor de "u"
- "u" es adyacente hacia "v" y "v" es adyacente desde "u"

grado de salida.- Es el número de aristas que empiezan en el nodo "u"

grado de entrada.- Es el número de aristas que terminan en el nodo "u"

nodo fuente.- Si tiene grado de salida positivo y grado de entrada nulo.

nodo sumidero.- Si tiene grado de salida nulo y grado de entrada positivo.



Nodo fuente: A
Nodo sumidero: B

Representación de los Grafos

- Representación secuencial del grafo a través de una matriz de adyacencia A
- Representación enlazada, empleando listas enlazadas.

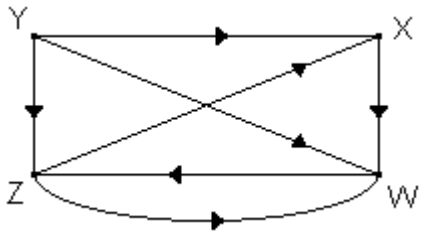
MATRIZ DE ADYACENCIA

Sea G un grafo dirigido simple de **m** nodos, cuyos nodos han sido ordenados y llamados $v_1, v_2, v_3, ..., v_m$.

, La matriz de adyacencia $A = (a_{ij})$ del grafo G es una matriz de m*n, donde cada:

$$a_{ij} = \begin{cases} 1 \text{ si } v_i \text{ es adyacente a } v_j, \text{ hay arista } (v_i, v_j) \\ 0 \text{ en caso contrario} \end{cases}$$

Ejemplo:



$v_1 = X$, $v_2 = Y$, $v_3 = Z$, $v_4 = W$

		v_1	v_2	v_3	v_4
		X	Y	Z	W
v_1	X	0	0	0	1
v_2	Y	1	0	1	1
v_3	Z	1	0	0	1
v_4	W	0	0	1	0

$$A=\begin{pmatrix} 0&0&0&1\\ 1&0&1&1\\ 1&0&0&1\\ 0&0&1&0 \end{pmatrix}$$

El número de 1(s) es igual al número de aristas.

$$A^2=\begin{pmatrix} 0&0&0&1\\ 1&0&1&1\\ 1&0&0&1\\ 0&0&1&0 \end{pmatrix}\begin{pmatrix} 0&0&0&1\\ 1&0&1&1\\ 1&0&0&1\\ 0&0&1&0 \end{pmatrix}=\begin{pmatrix} 0&0&1&0\\ 1&0&1&2\\ 0&0&1&1\\ 1&0&0&1 \end{pmatrix}\begin{matrix} \\ \\ \\ v_2 \end{matrix}$$

v_4

Hay dos caminos de longitud 2, de v_2 a v_4 *(y °w).*

$A^K = A^2$

Caminos de longitud 1

$$A^3=\begin{pmatrix} 0&0&0&1\\ 1&0&1&1\\ 1&0&0&1\\ 0&0&1&0 \end{pmatrix}\begin{pmatrix} 0&0&1&0\\ 1&0&1&2\\ 0&0&1&1\\ 1&0&0&1 \end{pmatrix}=\begin{pmatrix} 1&0&0&1\\ 1&0&2&2\\ 1&0&1&1\\ 0&0&1&1 \end{pmatrix}$$

$$A^4=\begin{pmatrix} 0&0&0&1\\ 1&0&1&1\\ 1&0&0&1\\ 0&0&1&0 \end{pmatrix}\begin{pmatrix} 1&0&0&1\\ 1&0&2&2\\ 1&0&1&1\\ 0&0&1&1 \end{pmatrix}=\begin{pmatrix} 0&0&1&1\\ 2&0&2&3\\ 1&0&1&2\\ 1&0&1&1 \end{pmatrix}$$

A es la matriz de adyacencia de un grafo
A^k da el número de caminos de longitud k desde v_i hasta v_j

$B_r = A + A^2 + A^3 + \ldots + A^r$

Esta matriz da el número de caminos de longitud **r** o menor de v_i al nodo v_j

Matriz de Caminos

$P = p_{ij}$ es la matriz de caminos de un grafo G
$p_{ij} = 1$ Si y solo sí hay un número positivo en la entrada ij de la matriz B_r

$$\underset{A}{\begin{pmatrix} 0&0&0&1\\ 1&0&1&1\\ 1&0&0&1\\ 0&0&1&0 \end{pmatrix}}\underset{A^2}{\begin{pmatrix} 0&0&1&0\\ 1&0&1&2\\ 0&0&1&1\\ 1&0&0&1 \end{pmatrix}}\underset{A^3}{\begin{pmatrix} 1&0&0&1\\ 1&0&2&2\\ 1&0&1&1\\ 0&0&1&1 \end{pmatrix}}\underset{A^4}{\begin{pmatrix} 0&0&1&1\\ 2&0&2&3\\ 1&0&1&2\\ 1&0&1&1 \end{pmatrix}}=\underset{B^4}{\begin{pmatrix} 1&0&2&3\\ 5&0&6&8\\ 3&0&3&5\\ 2&0&3&3 \end{pmatrix}}$$

$$\equiv \quad P=\begin{pmatrix} 1&0&1&1\\ 1&0&1&1\\ 1&0&1&1\\ 1&0&1&1 \end{pmatrix}$$

nodo v_2 es inalcanzable por los otros nodos

Algoritmo de Warshall

Algoritmo que es más eficiente que calcular las potencias de la matriz de adyacencia A y que permite encontrar las matriz sumatoria B

Para un grafo dirigido G con m nodos v_1, v_2, v_3, ..., v_m. Se halla la matriz de caminos P de la siguiente manera:

$P_{k\,(i,j)}=$ { 1 Si existe un camino simple de v_i a v_j que no usa otros nodos, aparte de posiblemente v_1, v_2, v_3, ..., v_m.
0 En otro caso

Warshall observó que: $P_k(i,j) = 1$
Cuando:

1. Existe un camino simple de v_i a v_j , $P_{k-1}(i,j) = 1$
2. Existe un camino simple de v_i a v_k y otro camino simple de v_k a v_i

∴ $P_{k-1}(i, k,) = 1$ y $P_{k-1}(k, ,j) = 1$

≡ Los elementos de la matriz P_k se obtiene de la siguiente manera:
$P_k(i,j) = P_{k-1}(i,j)$ ó ($P_{k-1}(i, k,)$ y $P_{k-1}(k, ,j)$)

Pseudocódigo

```
1) Inicio
2)    Repetir para i ← 1, m
      Repetir para i ← 1, m
        Si A[i, j] = 0
             Entonces P(i, j) ← 0
             Caso contrario P(i, j) ← 1
        Fin_si
      Fin_repetir_i
    Fin_repetir_j

3)    Repetir para k ← 1, m
      Repetir para i ← 1, m
        Repetir para j ← 1, m
          P(i, j) ← P(i, j) ó (P(i, k) y
        P(k, j))
        Fin_repetir_j
      Fin_repetir_i
    Fin_repetir_k
4) Fin
```

Algoritmo de Camino Mínimo

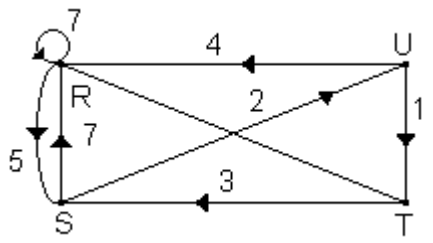
El grafo G de m nodos está en memoria a través de su matriz de pesos W, se encontrará la matriz de camino mínimo del nodo v_i a v_j .

INFINITO contiene un valor muy grande
MIN función de valor mínimo

Pseudocódigo

```
1) Inicio
2) Repetir para i ← 1, m
     Repetir para i ← 1, m
       Si w(i,j) = 0
         Entonces Q(i, j) ← INFINITO
         Caso contrario Q(i, j) ← w(i, j)
       Fin_si
     Fin_repetir_j
   Fin_repetir_i
3) Repetir para k ← 1, m
     Repetir para i ← 1, m
       Repetir para j ← 1, m
            Q(i, j) ← MIN( Q(i, j),  Q(i, k) + Q(k, j))
       Fin_repetir_j
     Fin_repetir_i
   Fin_repetir_k
4) Salir
```

Ejemplo:



$$W=\begin{array}{c} \\ R \\ S \\ T \\ U \end{array}\begin{array}{c} \begin{array}{cccc} R & S & T & U \end{array} \\ \begin{pmatrix} 7 & 5 & 0 & 0 \\ 7 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

$$Q_0=\begin{pmatrix} 7 & 5 & & \\ 7 & & & 2 \\ & 3 & & \\ 4 & & 1 & \end{pmatrix}\begin{pmatrix} RR & RS & -- & -- \\ SR & -- & -- & SU \\ -- & TS & -- & -- \\ UR & -- & UT & -- \end{pmatrix}$$

$$Q_1=\begin{pmatrix} 7 & 5 & & \\ 7 & 12 & & 2 \\ & 3 & & \\ 4 & 9 & 1 & \end{pmatrix}\begin{pmatrix} RR & RS & -- & -- \\ SR & SRS & -- & SU \\ -- & TS & -- & -- \\ UR & URS & UT & -- \end{pmatrix}$$

$$Q_4=\begin{pmatrix} 7 & 5 & 8 & 7 \\ 6 & 6 & 3 & 2 \\ 9 & 3 & 6 & 5 \\ 4 & 4 & 1 & 6 \end{pmatrix}\begin{pmatrix} RR & RS & RSUT & RSU \\ SUR & SUTS & SUT & SU \\ TSUR & TS & TSUT & TSU \\ UR & UTS & UT & UTS \end{pmatrix}$$

Semana 15: Aplicaciones GRAFOS

Gracias a la teoría de grafos se pueden resolver diversos problemas como por ejemplo la síntesis de circuitos secuenciales, contadores o sistemas de apertura. Se utiliza para diferentes áreas por ejemplo, Dibujo computacional, en todas las áreas de Ingeniería.

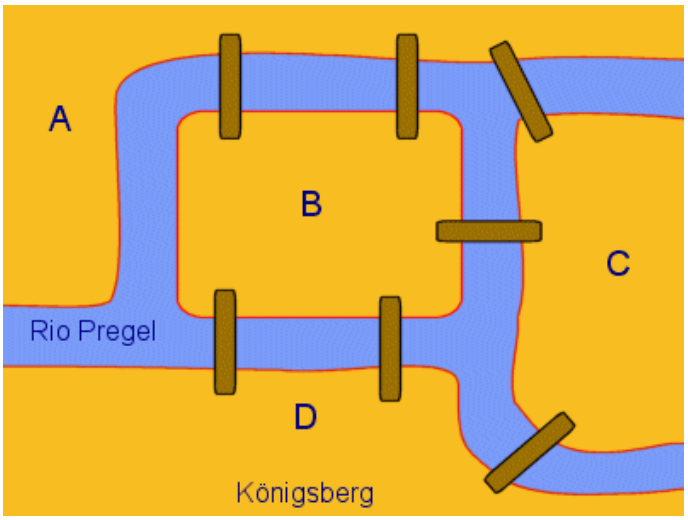
Los grafos se utilizan también para modelar trayectos como el de una línea de autobús a través de las calles de una ciudad, en el que podemos obtener caminos óptimos para el trayecto aplicando diversos algoritmos como puede ser el algoritmo de Floyd.

Para la administración de proyectos, utilizamos técnicas como PERT en las que se modelan los mismos utilizando grafos y optimizando los tiempos para concretar los mismos.

La teoría de grafos también ha servido de inspiración para las ciencias sociales, en especial para desarrollar un concepto no metafórico de red social que sustituye los nodos por los actores sociales y verifica la posición, centralidad e importancia de cada actor dentro de la red. Esta medida permite cuantificar y abstraer relaciones complejas, de manera que la estructura social puede representarse gráficamente. Por ejemplo, una red social puede representar la estructura de poder dentro de una sociedad al identificar los vínculos (aristas), su dirección e intensidad y da idea de la manera en que el poder se transmite y a quiénes.

Los grafos es importante por estudiando biología y hábitat. El vértice representa un hábitat y el edges representa los senderos de animales o las migraciones. Con esta información, científicos pueden entender como este cambiar afectar los especies en hábitats[1].

Puentes de Konigsberg



Uno de los primeros problemas que fueron modelados usando grafos fue el que confrontó Leonard Euler (1736). En la ciudad de Kaliningrado (antigua Königsberg) había siete puentes sobre el río Pregel. Uno de los puentes conectaba dos islas entre sí. Una de las islas estaba conectada a una ribera por dos puentes y otros dos puentes la conectaban con la otra costa. La otra isla poseía un puente hacia cada ribera. Euler se preguntó si sería posible comenzar un paseo desde cualquier punto y atravesar cada puente una y sólo una vez, regresando al punto departida.

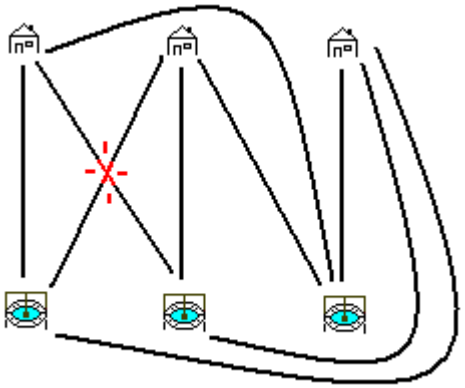
[1] http://es.wikipedia.org/wiki/Teor%C3%ADa_de_grafos#Aplicaciones

Grafos Planos

Cuando un grafo o multígrafo se puede dibujar en un plano sin que dos segmentos se corten, se dice que es plano.

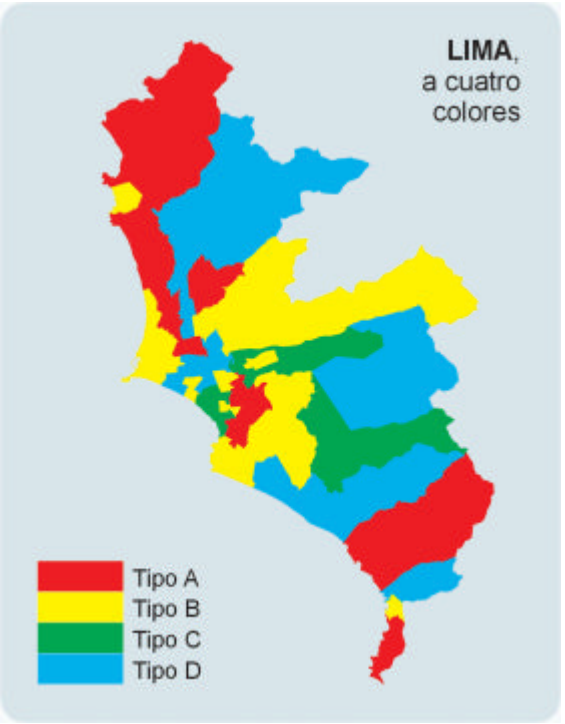
Aplicación

Se dibujan tres casas y tres pozos. Todos los vecinos de las casas tienen el derecho de utilizar los tres pozos. Como no se llevan bien en absoluto, no quieren cruzarse jamás. ¿Es posible trazar los nueve caminos que juntan las tres casas con los tres pozos sin que haya cruces? Respuesta NO.



Un grafo es plano si se puede dibujar sin cruces de aristas.

Teorema de los 4 colores



En 1852 Francis Guthrie planteó el problema de los 4 colores, resuelto hasta un siglo después por Kenneth Appel y Wolfgang Haken.

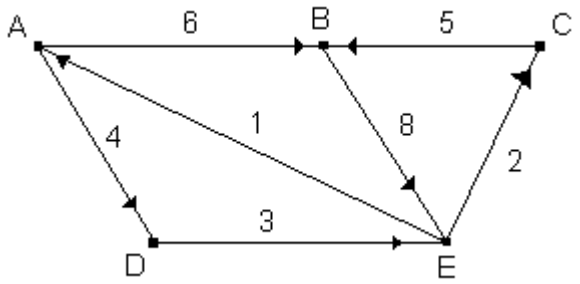
El teorema de cuatro colores establece que cualquier mapa geográfico puede ser coloreado con cuatro colores diferentes, de forma que no queden regiones adyacentes con el mismo color. Dos regiones se dicen adyacentes si comparten un segmento de borde en común, no solamente un punto.

La forma precisa de cada país no importa; lo único relevante es saber qué país toca a qué otro. Estos datos están incluidos en el grafo donde los vértices son los países y las aristas conectan los que justamente son adyacentes. Entonces la cuestión equivale a atribuir a cada vértice un color distinto del de sus vecinos.

Si se empieza por el país central a y se esfuerza uno en utilizar el menor número de colores, entonces en la corona alrededor de a alternan dos colores. Llegando al país h se tiene que introducir un cuarto color. Lo mismo sucede en i si se emplea el mismo método.

EJERCICIOS

1. Determinar las rutas mínimas de longitud K



2. Dada la matriz de adyacencia de un grafo G de 5 vértices (v_1, v_2, v_3, v_4, v_5):

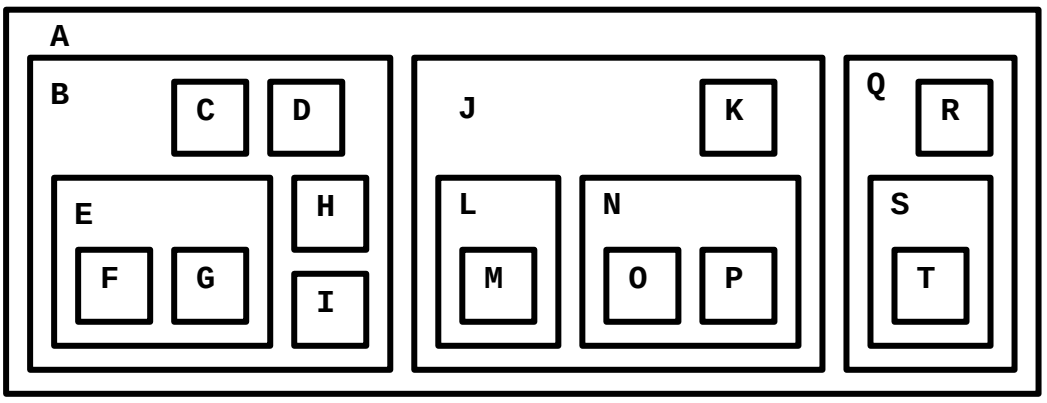
0	1	0	1	0
1	0	1	0	1
1	0	1	1	0
0	1	0	0	1
1	0	1	0	1

a. Dibujar el grafo.

b. Determine los grados de cada uno de los vértices y el número de aristas del grafo.

c. Determine e indique los caminos de longitud 3 del grafo.

3. Dado el siguiente diagrama de Venn que corresponde a la estructura de un árbol



d. Conviértalo a notación indentada.

e. Represéntelo como anidación de paréntesis.*)*

f. Represéntelo como un grafo, considerando los hijos únicos a la derecha

g. Convierta el árbol general en un árbol binario.

Del árbol binario generado calcule a través de un algoritmo (pseudo código):

h. La altura.

i. El número de nodos terminales.

REFERENCIAS BIBLIOGRAFICAS

1. Joyanes Aguilar, Luis. FUNDAMENTOS DE PROGRAMACIÓN : ALGORITMOS, ESTRUCTURAS DE DATOS Y OBJETOS. Madrid , McGraw-Hill , 2003. Código Biblioteca UNS: 005.1 J79.

2. Ceballos Sierra, Fco. Javier. JAVA 2 CURSO DE PROGRAMACIÓN. México D.F , Prentice Hall , 2006. Código Biblioteca UNS: 005.133 C42.

3. Froufe Quintas, Agustín. JAVA 2 MANUAL DEL USUARIO Y TUTORIAL. México D.F , Alfaomega , 2006. Código Biblioteca UNS: 005.133 F85.

4. Jaime Sisa, Alberto. ESTRUCTURA DE DATOS Y ALGORITMOS: CON ÉNFASIS EN PROGRAMACIÓN ORIENTADA A OBJETOS. Bogotá , Pearson , 2002. Código Biblioteca UNS: 005.1 S59

5. Hernández, Roberto; Lázaro, Juan Carlos; Dormido, Raquel; Ro. ESTRUCTURAS DE DATOS Y ALGORITMOS. Madrid , Pearson Educación , 2001. Código Biblioteca UNS: 005.73 E92.