EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

Dirección General de Educación Tecnológica Industrial y de Servicios

Dirección Académica e Innovación Educativa

Subdirección de Innovación Académica

Departamento de Planes, Programas y Superación Académica

Cuadernillo de Aprendizajes Esenciales

Módulo I

Programación

México 2021 Año de la Independencia

Aprendizajes esenciales			
Carrera:	PROGRAMACIÓN	Semestre:	2o
Módulo/Submódulo:	**Módulo I. Desarrolla software de aplicación con programación estructurada.** **Submódulo 1**. Construye algoritmos para la solución de problemas		

Aprendizajes esenciales o Competencias esenciales 1er parcial	Estrategias de Aprendizaje	Productos a Evaluar
1. Desarrolla la solución de un problema por medio de algoritmos	1. Lee el **Anexo** A que te introduce a la competencia profesional "Desarrolla la solución de un problema por medio de algoritmos" **2. Realiza los ejercicios** ahí indicados en el Anexo A, en tu libreta.	Algoritmo del área de un triangulo Algoritmo si un número es positivo o negativo

ANEXO A

Introducción

Un algoritmo es la representación en un conjunto de pasos finitos (inicio-fin) que llevan a cabo un proceso para resolver un problema.

En la vida diaria nosotros llevamos a cabo una serie de pasos cuando realizamos una actividad o tomamos una decisión.

Ejemplo: Para cocinar unos huevos estrellados ¿qué pasos seguimos?

1. INICIO
2. Poner el sartén en la estufa
3. Incorporarle aceite comestible
4. Prender el fuego
5. Caliente el aceite bajar la intensidad del fuego
6. Incorporar los huevos
7. Estando en su punto de cocción según el gusto con una espátula se voltean los huevos
8. Ya en su punto se apaga el fuego
9. Con la espátula o paleta se sacan los huevos del sartén y se depositan en un plato
10. Colocar la espátula en el fregadero.
11. Degustar los huevos, buen provecho.
12. FIN

Si tu observas son un conjunto de pasos **finitos ,** es decir, que termina en algún momento. Los pasos nos van indicando qué proceso y el orden en que debemos hacerlo (**preciso**) para obtener huevos estrellados. No se debe omitir nada, debe ser claro y que se entienda por quien lo lee (**legible**) y; que pueda hacerlo varias veces y obtener el mismo resultado (**definido**).

Nuestro ejemplo es un algoritmo y sus **características son**: **finito, preciso, legible y definido.**

Los algoritmos constan de 3 partes:

1. **La entrada,** es decir, los **datos** que necesito para realizar ese conjunto de pasos.

En nuestro ejemplo serían materias primas: aceite, huevos
Utensilios de cocina: sartén, y espátula
Losa: plato
Electrodoméstico: estufa
Servicio de gas

2. **El proceso,** es decir, las **operaciones lógicas** que deben de ejecutar con los **datos** proporcionados.
   En nuestro ejemplo preparar el sartén y freír los huevos.
3. **La salida,** el **resultado** obtenido del **proceso.**

En nuestro ejemplo el resultado son los huevos estrellados después de freírlos.

En conclusión, un algoritmo resuelve paso a paso un problema, cuantas veces sea llevado a cabo.

La solución de un problema lo podemos representar en: Algoritmo Natural, Diagrama de Flujo y Pseudocódigo.

Trataremos cada uno de ellos.

Otro ejemplo, cuando tomamos decisiones.
Planteamiento del Problema:
He terminado mi secundaria y tengo que decidir en qué escuela continuar mis estudios de preparatoria.

La siguiente tabla te permite describir qué necesitas y qué esperas como resultado para el desarrollo del algoritmo, como resultado de leer y comprender el planteamiento del problema.

ENTRADA ¿qué necesitas?	**ALGORITMO (ENTRADA-PROCESO-SALIDA)**	**SALIDA ¿qué esperas?**
1. Escuelas de mi entorno. 2. Oferta que me ofrecen: cursar el bachillerato, cursar a su vez una especialidad técnica y beca. 3. Distancia y costo de traslado. 4. Costo de inscripción. 5. Posibilidades económicas de la familia.	1. INICIO 2. [**Datos-entrada-** de las escuelas] CETis cabecera municipal. Ofrece Bachillerato, especialidades y beca. Distancia de 20’ Costo de traslado ida y vuelta $ 35.00 Inscripción voluntaria $ 600.00. Bachillerato en mi localidad. Ofrece Bachillerato y beca. Distancia 5’ Costo de traslado $0.00 Inscripción $ 800.00 3. [**Proceso** de toma de decisión] **Si** existen las posibilidades económicas para el CETis? **Si (entonces)** Ingreso al CETis por representar una opción que me ofrece además salir también como técnico en alguna especialidad. **No**	Inscripción al CETis O Inscripción al Bachillerato

	Ingreso al Bachillerato de la U de C. 4. [**Salida – resultado**-] Inscripción al CETis.	

Ahora bien, desarrollemos ejemplos de algoritmos relacionados a tu preparación académica.

EJEMPLOS:

1. **Planteamiento del problema:**
   Obtener la suma de 4 números y la media.

ENTRADA ¿qué necesitas?	**ALGORITMO (ENTRADA-PROCESO-SALIDA)**	**SALIDA ¿qué esperas?**
1. Los 4 números 2. Realizar la suma 3. El resultado de la suma dividirlo entre 4 para obtener la media	1. INICIO 2. [Los 4 números] 2, 4, 8, 10 3. [Realizar la suma y obtener la media] suma=2+4+8+10 media= suma/4 4. [Resultado] Suma= 24 Media= 6 5. FIN	Suma= 24 Media= 6

2. **Planteamiento del problema.**
   Dado un número saber si es múltiplo de 2.

ENTRADA ¿qué necesitas?	**ALGORITMO (ENTRADA-PROCESO-SALIDA)**	**SALIDA ¿qué esperas?**
1. Un número 2. El número dividirlo entre 2 3. Si el residuo de la división es 0, entonces el número es múltiplo de 2	1. INICIO 2. [Dar el número] 7 3. [Obtener el residuo de la división y decir si es múltiplo o no]] **residuo=7%2** **Si <u>residuo=0</u>** (entonces) El número 7 es múltiplo de 2 (RESULT) **No** El número 7 no es múltiplo de 2 (RESULT) 4. FIN	El número 7 es múltiplo de 2 O El número 7 no es múltiplo de 2

Explicación:

El signo de **%** en programación **es un operador** que nos da como resultado el **residuo de una división**: **7%2 = 1**

En una condición con operadores relacionales siempre va:

Dato1 **Operador** Dato2

Variable 1 **Operador** Variable2

La **variable es** un nombre que damos para guardar un valor, en nuestro ejercicio es la palabra residuo.

residuo=0 es una condición y utilizamos los **operadores relaciones** para indicar la condición, como son:

Mayor	>	a > b	**a** es mayor que **b**
Menor	<	a < b	**a** es menor que **b**
Mayor o igual	>=	a >= b	**a** es mayor o igual a **b**
Menor o igual	<=	a <= b	**a** es menor o igual a **b**
Diferente	!=	a != b	**a** es diferente a **b**
Igual	=	a = b	**a** es igual a **b**

ACTIVIDADES DE APRENDIZAJE (TAREAS)

Los algoritmos que acabo de ejemplificar se conocen como **Algoritmos Naturales porque se asemejan a nuestra lengua.**

Instrucciones:

Los siguientes planteamientos de problemas represéntalos en algoritmo natural, en tú libreta.
Lee detenidamente y comprende lo que se te plantea.
Presenta cada uno en una tabla como en los ejemplos proporcionados.

Planteamiento del problema:

1. Obtener el área de un triángulo.
2. Dado un número determinar si el número es positivo o negativo.

Aprendizajes esenciales o Competencias esenciales 2º parcial	Estrategias de Aprendizaje	Productos a Evaluar
2. Desarrolla la solución de un problema por medio de Diagramas de Flujo	1. Lee el **Anexo B** que te introduce a la competencia profesional "Desarrolla la solución de un problema por medio de Diagramas de Flujo" **2. Realiza los ejercicios** ahí indicados en el Anexo B, en tu libreta.	Diagrama de flujo del área de un triángulo Diagrama de Flujo si un número es positivo o negativo

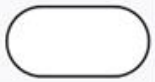
ANEXO B

Introducción

Un algoritmo podemos representarlo gráficamente en un Diagrama de flujo.
Por lo tanto, un **Diagrama de Flujo es** la representación gráfica de un algoritmo, es un algoritmo en su forma gráfica.
Las **características** de un Diagrama de Flujo son:

1. Sintético es decir, representar el proceso de forma comprensible.
2. Simbolizada es decir, representarlo mediante símbolos.

Simbología y significado:

Forma ANSI/ISO	Nombre	Descripción
	Línea de flujo (Flecha)[4]	Muestra el orden de operación de los procesos. Una línea saliendo de un símbolo y apuntando a otro.[3] Las fechas se agregan si el flujo no es el estándar de arriba hacia abajo, de izquierda a derecha.[4]
	Terminal[3]	Indica el inicio o fin de un programa o subprocesos. Se representa como un stadium,[3] óvalo o rectángulo redondeado. Usualmente contienen la palabra "Inicio" o "Fin", o alguna otra frase señalando el inicio o fin de un proceso, como "presentar consulta" o "recibir producto".
	Proceso[4]	Representa un conjunto de operaciones que cambiar el valor, forma o ubicación de datos. Representado como un rectángulo.[4]
	Decisión[4]	Muestra una operación condicional que determina cuál de los dos caminos tomará el programa. [3] La operación es comúnmente una pregunta de sí/no o una prueba de verdadero/falso. Representada como un rombo.(rombo).[4]
	Anotación[3] (Comentario)[4]	Indica información adicional acerca de un paso en el programa. Representado como un rectángulo abierto con una línea (que puede ser punteada) conectándolo con el símbolo correspondiente del diagrama de flujo. [4]
	Proceso Predefinido[3]	Muestra, por su nombre, un proceso que ha sido definido en otro lugar. Representado como un rectángulo con un doble lateral en cada lado. .[3]
	Conector de Página[3]	Pares de conectores etiquetados reemplazan líneas largas o confusas en la página del diagrama. Representados como pequeños círculos con una letra dentro. [3] [6]
	Conector fuera de página[3]	Un conector etiqueta para usar cuando el objetivo es otra página. Representado con la forma de un plato de "Home" (béisbol) pentágono.[3] [6]

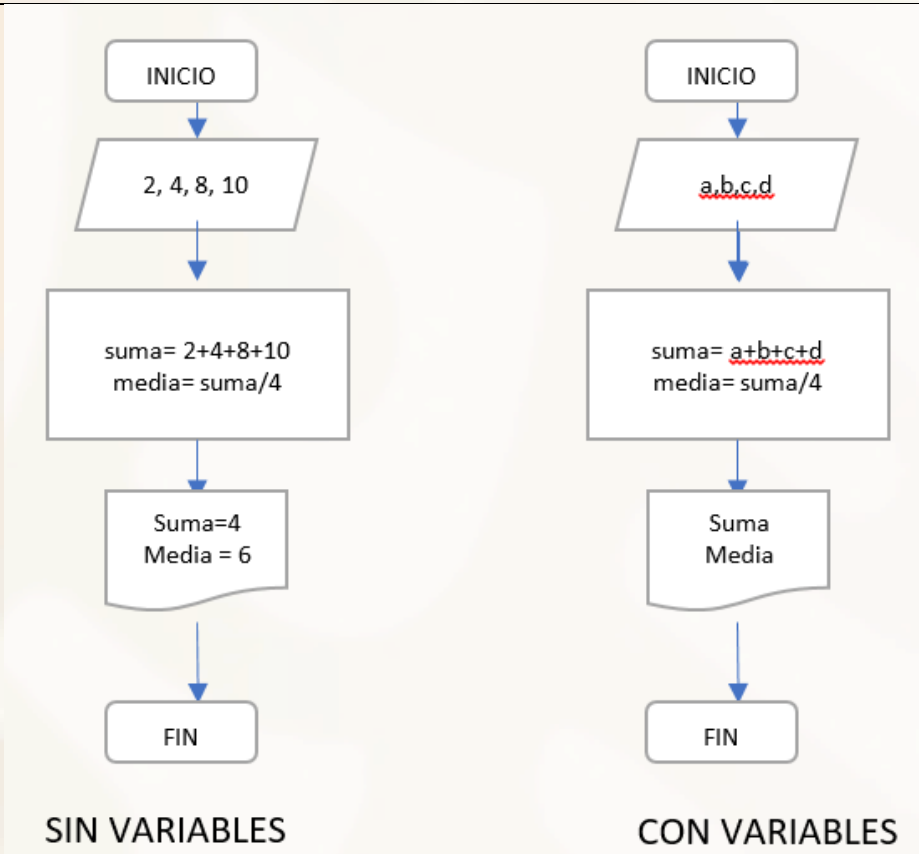
Ahora bien, los ejemplos de algoritmos desarrollados en el Anexo A los vamos a representar en su forma gráfica en Diagramas de Flujo.

Trabajaremos con **VARIABLES** para que nuestro algoritmo pueda ejecutarse varias veces con valores diferentes.

Representaré ambos diagramas de flujo para tu comprensión de lo que se te pide de trabajar con variables.

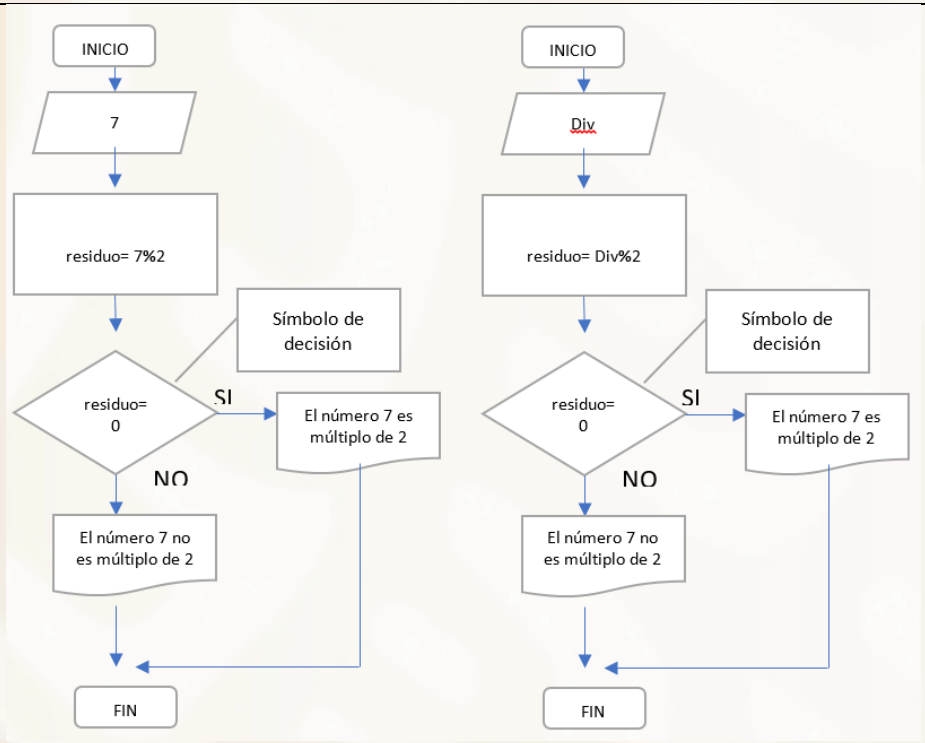
Obtener la suma de 4 números y la media.

1. INICIO
2. [Los 4 números]
   2, 4, 8, 10
3. [Realizar la suma y obtener la media]
   suma=2+4+8+10
   media= suma/4
4. [Resultado]
   Suma= 24
   Media= 6
5. FIN



Dado un número saber si es múltiplo de 2.

1. INICIO
2. [Dar el número]
   7
3. [Obtener el residuo de la división y decir si es múltiplo o no]]
   residuo=7%2
   Si <u>residuo=0</u> (entonces)
   El número 7 es múltiplo de 2 (RESULTADO)
   No
   El número 7 no es múltiplo de 2 (RESULTADO)
4. FIN



ACTIVIDADES DE APRENDIZAJE (TAREAS)
Los algoritmos que acabo de ejemplificar son en su representación gráfica en Diagrama de Flujo.

Instrucciones:
Los algoritmos que hiciste en el Anexo A represéntalos en Diagrama de Flujo en tú libreta.
Los Diagramas de Flujo deberás hacerlos empleando VARIABLES.

Planteamiento del problema:
1. Obtener el área de un triángulo.
2. Dado un número determinar si el número es positivo o negativo.

Aprendizajes esenciales o Competencias esenciales 3er parcial	Estrategias de Aprendizaje	Productos a Evaluar
3. Realiza Pseudocódigo	1. Lee el **Anexo C** que te introduce a la competencia profesional "Realiza Pseudocódigo" **2. Realiza los ejercicios** ahí indicados en el Anexo C, en tu libreta. 3. Responde el siguiente **cuestionario** en tú libreta. 1. ¿Qué es un algoritmo? 2. ¿Cuál es el propósito o finalidad de los algoritmos? 3. ¿De cuántas formas podemos representar los algoritmos? 4. Menciona las diferencias entre los diferentes tipos de algoritmos: **Algoritmo** **Diagrama de Flujo** **Pseudocódigo**	Algoritmo Pseudocódigo del área de un triángulo Algoritmo Pseudocódigo si un número es positivo o negativo Cuestionario

ANEXO C

Introducción

Representar un problema en un Algoritmo Pseudocódigo nos va introduciendo a los Lenguajes de Programación con los que hacemos programas para las computadoras o dispositivos móviles. De un Algoritmo Pseudocódigo lo traducimos a un lenguaje de programación.

Un **Algoritmo Pseudocódigo es** la representación de la solución de un problema y usa algunas palabras reservadas similares a los lenguajes de programación, pero a la vez no representa a ninguno en específico.

La estructura básica es:

Cabecera
- Nombre del Programa
- Declarar constantes
- Declarar variables

Cuerpo
- Inicio
- Instrucciones
  - Entrada de datos
  - Proceso
  - Salida
- Fin

Palabras más comunes a utilizar:

Read datos de entrada que nos del usuario y se guardan en una variable
Write muestra mensajes o resultados en pantalla

Vayamos a los ejemplos.

1.Obtener la suma de 4 números y la media.

Suma_Media ←

Nombre del Programa: abreviado, sin espacios y que nos refleje que hace el pseudocódigo.

1. [Inicio]
2. [**Declarar** constantes y Variables]

a, b, c, d
suma
media

3. [Leer datos de **entrada**]

Write "Dame el valor de a"
Read a
Write "Dame el valor de b"
Read b
Write "Dame el valor de c"
Read c
Write "Dame el valor de d"
Read d

Se declaran las variables que vamos a necesitar.

Con **write** le enviamos un mensaje a la pantalla del usuario para que sepa que le estamos pidiendo, el mensaje va entre comillas. **Read** permite que el usuario proporcione los valores desde el teclado, de esta manera, cada variable va almacenar el valor que le da el usuario.

4. [**Cálculo u operación**]

suma=a+b+c+d
media= suma/4

En este paso del pseudocódigo se realizan los **cálculos u operaciones**, las variables ya traen valores que el usuario le dio en el paso 3.

5. [Resultado]

Write "La Suma es: ", suma
Write "La media es: , media

En el paso 5 le mostramos al usuario el **resultado final.**
Con write le enviamos un mensaje al usuario del resultado que se obtuvo y separado por coma (,) la variable que almacena el resultado.

6. FIN

2.Dado un número saber si es múltiplo de 2
Múltiplo_2

1. [Inicio]
2. [Declarar constantes y variables]
   num, residuo
3. [Leer datos de **entrada**]
   Write "Dame un número"
   Read num
4. [**Cálculo u operación**]
   residuo= num%2
   Si residuo=0
   Write "El número 7 es múltiplo de 2"
   No
   Write "El número 7 no es múltiplo de 2 "
5. FIN

Observa que se declaran 2 variables:
num para almacenar el dato que el usuario nos va dar
residuo para almacenar el resultado de la operación num%2

Todas las variables que se usan en un algoritmo se declaran.

ACTIVIDADES DE APRENDIZAJE (TAREAS)

Instrucciones:

Los algoritmos que hiciste en el Anexo A represéntalos en Algoritmo Pseudocódigo en tú libreta.
Deberás hacerlo empleando VARIABLES.

Planteamiento del problema:

1. Obtener el área de un triángulo.
2. Dado un número determinar si el número es positivo o negativo.

Aprendizajes esenciales			
Carrera:	PROGRAMACIÓN	Semestre:	2o.
Módulo/Submódulo:	**Módulo I. Desarrolla software de aplicación con programación estructurada.** **Submódulo 2**. Aplica estructuras de control con un lenguaje de programación		

Aprendizajes esenciales o Competencias esenciales 1er parcial	Estrategias de Aprendizaje	Productos a Evaluar
1. Describiendo la estructura general de un programa	1. El alumno contestará las siguientes preguntas en su libreta **a. ¿Qué un programa?** **b. ¿Qué es la programación?** **c. ¿Qué es un Pseudocódigo?** **d. ¿Qué entiendes por estructura?** **Puedes apoyarte en la siguiente información:** Inicialmente el término Programa sirve para denotar aquella agrupación de actividades que tanto en secuencia o simultáneas son ejecutadas por un equipo de individuos a fin de que se cumpla un objetivo. Un programa informático es una serie de comandos ejecutados por el equipo. Sin embargo, el equipo solo es capaz de procesar elementos binarios, es decir, una serie de 0 y 1. Por lo tanto, necesitamos un lenguaje de programación para escribir de manera legible, es decir, con comandos que el ser humano pueda comprender (por ser similares a su propio lenguaje) los comandos que el equipo deberá ejecutar. Estos programas se traducen después a un lenguaje máquina (en binario) a través de un compilador. La programación informática es el arte del proceso por el cual se limpia, codifica, traza y protege el código fuente de programas computacionales, en otras palabras, es indicarle a la computadora lo que tiene que hacer. La programación se guía por una serie de normas y un conjunto de órdenes, instrucciones y expresiones que tienden a ser semejantes a una lengua natural acotada. Por lo cual recibe el nombre de lenguaje de programación.	1. Cuestionario.

	Pseudocódigo (o falso Lenguaje). Es comúnmente utilizado por los programadores para omitir secciones de Código o para dar una explicación del paradigma que tomó el mismo programador para hacer sus códigos, esto quiere decir que el pseudocódigo no es programable sino facilita la programación. El principal objetivo del pseudocódigo es el de representar la solución a un algoritmo de la forma más detallada posible, y a su vez lo más parecida posible al lenguaje que posteriormente se utilizará para la codificación del mismo. La programación estructurada es un paradigma de programación basado en utilizar funciones o subrutinas, y únicamente tres estructuras de control: Secuencia: ejecución de una sentencia tras otra. Selección o condicional: ejecución de una sentencia o conjunto de sentencias, según el valor de una variable booleana. Iteración (ciclo o bucle): ejecución de una sentencia o conjunto de sentencias, mientras una variable booleana sea verdadera. Este paradigma se fundamente en el teorema correspondiente, que establece que toda función computable puede ser implementada en un lenguaje de programación que combine sólo estas tres estructuras lógicas o de control.	
	2. Contestar la siguiente pregunta **¿Cuál es la estructura general de un programa?** con los datos obtenidos, realizar una pirámide de información en el cuaderno. **Puedes apoyarte en la siguiente información:** Un programa puede considerarse como una secuencia de acciones (instrucciones) que manipulan un conjunto de objetos (datos). **Bloques de un programa**: • Bloque de declaraciones: en él se especifican todos los objetos que utiliza el programa (constantes, variables, tablas, registros, archivos, etc.). • Bloque de instrucciones: constituido por el conjunto de operaciones que se han de realizar para la obtención de los resultados deseados. **Partes principales de un programa:**	2. Pirámide de información

	Dentro del bloque de instrucciones de un programa se pueden diferenciar tres partes fundamentales. En algunos casos, estas tres partes están perfectamente delimitadas, pero en la mayoría sus instrucciones quedan entremezcladas a lo largo del programa, si bien mantienen una cierta localización geométrica impuesta por la propia naturaleza de las mismas. • **Entrada de datos:** la constituyen todas aquellas instrucciones que toman datos de un dispositivo externo, almacenándolos en la memoria central para que puedan ser procesados. • **Proceso o algoritmo:** está formado por las instrucciones que modifican los objetos a partir de su estado inicial hasta el estado final, dejando éstos disponibles en la memoria central. • **Salida de resultados:** conjunto de instrucciones que toman los datos finales de la memoria central y los envían a los dispositivos externos.	
	3. Investigar **¿Qué es el lenguaje C? ¿Quién fue su desarrollador? Identificar 3 ventajas y 3 desventajas del mismo**. Realizar un mapa conceptual con lo obtenido. **Puedes apoyarte en la siguiente información:** Lenguaje de programación C. También conocido como "Lenguaje de programación de sistemas" desarrollado en el año 1972 por Dennis Ritchie para UNIX un sistema operativo multiplataforma. El lenguaje C es del tipo lenguaje estructurado como son Pascal, Fortran, Basic. Sus instrucciones son muy parecidas a otros lenguajes incluyendo sentencias como if, else, for, do y while... . Aunque C es un lenguaje de alto nivel (puesto que es estructurado y posee sentencias y funciones que simplifican su funcionamiento) tenemos la posibilidad de programar a bajo nivel (como en el Assembler tocando los registros, memoria etc.). Para simplificar el funcionamiento del lenguaje C tiene incluidas librerías de funciones que pueden ser incluidas haciendo referencia la librería que las incluye, es decir que si queremos usar una función para borrar la pantalla tendremos que incluir en nuestro programa la librería que tiene la función para borrar la pantalla. La programación en C tiene una gran facilidad para escribir código compacto y sencillo a su misma vez. En el lenguaje C no tenemos procedimientos como en otros lenguajes solamente tenemos funciones los procedimientos los simula y esta terminante mente prohibido escribir funciones, procedimientos y los comandos en mayúscula todo se escribe en minúsculas **Desventajas del lenguaje C**: No es un lenguaje visual, no puede ser deducido de forma intuitiva, como por ejemplo el Visual Basic. Para el uso de funciones anidadas necesita de extensiones. No tiene instrucciones de entrada y salida, ni para el manejo de cadenas de caracteres.	3. Mapa Conceptual

	4. Realizar una búsqueda del significado de las siguientes palabras: **Algoritmo**, **Datos**, **Variable**, **Constante**, **Sentencia**, **Compilar**, **Ejecutar**, **Iteración**, **Ciclo** y **Condición** finalmente realizar un glosario. **Puedes apoyarte en la siguiente información**: En informática, un algoritmo es una secuencia de instrucciones secuenciales, gracias al cual pueden llevarse a cabo ciertos procesos y darse respuesta a determinadas necesidades o decisiones. Se trata de conjuntos ordenados y finitos de pasos, que nos permiten resolver un problema o tomar una decisión. Una instrucción condicional nos permite plantear la solución a un problema considerando los distintos casos que se pueden presentar. De esta manera, podemos utilizar un algoritmo distinto para enfrentar cada caso que pueda existir en el mundo. En programación se denomina bucle a la ejecución repetidas veces de un mismo conjunto de sentencias. Normalmente en cada nueva ejecución varía algún elemento. En programación, un tipo de dato informático o simplemente tipo es un atributo de los datos que indica al ordenador (y/o al programador) sobre la clase de datos que se va a trabajar. Esto incluye imponer restricciones en los datos, como qué valores pueden tomar y qué operaciones se pueden realizar. Los tipos de datos comunes son: números enteros, números con signo (negativos), números de coma flotante (decimales), cadenas alfanuméricas, estados (booleano), etc. En programación, las variables son espacios reservados en la memoria que, como su nombre indica, pueden cambiar de contenido a lo largo de la ejecución de un programa. Una variable corresponde a un área reservada en la memoria principal del ordenador. En programación, una constante es un valor que no puede ser alterado durante la ejecución de un programa. Una constante corresponde a una longitud fija de un área reservada en la memoria principal del ordenador, donde el programa almacena valores fijos. Las sentencias son los elementos básicos en los que se divide el código en un lenguaje de programación. Al fin y al cabo, un programa no es más que un conjunto de sentencias que se ejecutan para realizar una cierta tarea.	4. Glosario

	El programa escrito en un lenguaje de programación (fácilmente comprensible por el programador) es llamado programa fuente y no se puede ejecutar directamente en una computadora. La opción más común es compilar el programa obteniendo un módulo objeto, aunque también puede ejecutarse en forma más directa a través de un intérprete informático. El código fuente del programa se debe someter a un proceso de traducción para convertirlo a lenguaje máquina o bien a un código intermedio, generando así un módulo denominado "objeto". A este proceso se le llama compilación. En informática, ejecutar es la acción de iniciar la carga de un programa o de cualquier archivo ejecutable. En otras palabras, la ejecución es el proceso mediante el cual una computadora lleva a cabo las instrucciones de un programa informático. Para la programación, por su parte, la iteración consiste en reiterar un conjunto de instrucciones o acciones con uno o varios objetivos. Para citar un ejemplo, muchas páginas web están preparadas para adaptarse a cambios en su estructura, tales como alteraciones estéticas o del número de secciones accesibles, cuyos enlaces se muestran en forma de pestañas	
	5. Realizar una investigación en la cual identifiques ✓ ¿cuál es la función principal del lenguaje C y C++? ✓ ¿cuáles son sus costos? ✓ ¿en qué sistema operativo se pueden instalar? ✓ ¿en qué año salieron al mercado? ✓ ¿Quién fue su creador? para finalizar elaborar una tabla comparativa de los 2 lenguajes Puedes apoyarte en la siguiente información: El lenguaje C, está orientado a la programación estructurada. ¿En qué consiste la programación estructurada? Pues, básicamente, en trabajar con código secuencial, con un conjunto de sentencias o instrucciones que se ejecutan una por una. Las podemos clasificar en: • Instrucciones condicionales. • Instrucciones de iteración (bucle de instrucciones). El concepto de estructurada viene de trabajar con funciones. En cambio, C++ también está orientado a la Programación POO (Programación orientada a Objetos). Esta es la diferencia más grande entre los dos idiomas.	5. Tabla Comparativa

	El lenguaje C es de código Abierto u Open Source se refiere al código fuente del software que es abiertamente accesible y que puede ser cambiado y distribuido por cualquier persona. No se requiere ninguna licencia adicional en ningún momento. C ha tenido distintos usos a lo largo de la historia, con aplicaciones en sistemas operativos, compiladores y desarrollo de software. El lenguaje C puede ser utilizado en diferentes sistemas entre los principales son los siguientes: Windows, MacOS, Linux, Unix. El lenguaje C tiene otros lenguajes que se consideran sus antecesores (BCPL, B) y comenzó a utilizarse en los años 70. Su fecha de "nacimiento" como lenguaje de uso extendido suele decirse que es 1978 cuando Brian Kernighan y Dennis Ritchie publicaron el libro The C Programming Language, popularmente denominado "La Biblia de C". En este libro se definía de forma clara y precisa este lenguaje de programación. El 14 de octubre del año 1985 salió publicada la primera guía de referencia de C++, por lo que es considerada como la fecha de "nacimiento" de este lenguaje de programación. El C++ fue diseñado por Bjarne Stroustrup en el año 1980 (en los míticos Laboratorios Bell) como una extensión del lenguaje de programación C, diseñado para ser un "lenguaje de uso general": puede correr sobre cualquier plataforma, y está en todos lados, sobre todo en videojuegos y sistemas integrados. Este éxito llevó a que en el año 1990 se reunieran las organizaciones ANSI e ISO con el objeto de definir un estándar que formalizara al lenguaje, proceso que culminó en el año 1998 cuando salió aprobado ANSI C++.	
	6. Todo lenguaje de programación tiene sus propias palabras reservadas, has la lectura de la siguiente información y construye un concepto propio de lo entendido. **Puedes apoyarte en la siguiente información**: Las palabras reservadas en programación, o palabras clase, tienen un significado especial para el compilador de cualquier lenguaje de programación, estas palabras pueden identificar los tipos de datos que se pueden usar, además de las diferentes rutinas de programación que permite cada lenguaje.	6. Concepto Personal
Aprendizajes esenciales o Competencias esenciales 2º parcial	Estrategias de Aprendizaje	Productos a Evaluar
2. Elaborando un programa que incluya instrucciones de entrada, proceso y salida	1. Los tipos de datos son esenciales para la elaboración de un programa dentro de los más usados encontramos la siguiente lista: ✓ Carácter ✓ Entero	1. Tabla

	✓ Cadena ✓ Flotante ✓ Booleanos Con base a la información anterior elaborar una tabla en tu libreta en la que se muestre el nombre del tipo de dato, su función y dos ejemplos. Puedes apoyarte en la siguiente información: **Carácter**.- En este tipo de dato se encuentran todos los caracteres conocidos, una letra, un número, un símbolo especial. Por lo tanto, está conformado por los DÍGITOS:'0', '1', '2', ... , '9'; LETRAS: 'a', 'b', 'c', ... , 'z'; MAYÚSCULAS: 'A', 'B', 'C', ... , 'Z'; y CARACTERES ESPECIALES: '%', '*', '?', ... , '/'. En algunos lenguajes de programación como Java y C#, se utiliza la comilla simple (' ') para identificar un carácter, sin embargo, esto puede cambiar dependiendo del lenguaje de programación. EJEMPLO: opción= '1' **Entero**.- Este tipo dato corresponde a aquellas variables que exclusivamente pueden recibir VALORES SIN PARTE DECIMAL. Generalmente se utilizan en las variables que contienen cantidades de elementos que no pueden fraccionarse, como el número de personas, el número de edificios, entre otros. EJEMPLO: nroEstudiantes: 40 **Cadena.-** Constituyen conjuntos de caracteres, es decir la UNIÓN DE VARIOS CARACTERES, que pueden ser palabras o frases. El valor de este tipo de datos se encierra generalmente entre comillas (" "). EJEMPLO: nombre: "Sandra Torres" **Flotantes.-** Los tipos de datos de coma flotante son tipos de datos aproximados. El sistema redondea el significante si hay presente más precisión de la que puede representar. EJEMPLO: precio: 19.70 **Booleanos.-** Los booleanos o tipos de datos lógicos, únicamente reciben dos valores: true ó false. Se utilizan generalmente como banderas, para identificar si se realizó o no un proceso. EJEMPLO: aprobó= true	
	2. En programación para la realización de un programa se divide en 3 etapas las cuales son entrada, proceso y salida. Las **entradas** son todos aquellos insumos que se requieren para el adecuado procesamiento de los datos y que se definirán como variables.	2. Algoritmos realizados

Los **procesos** son los diversos métodos o instrucciones mediante las cuales se realizan cambios a las entradas para convertirlas en un resultado.

Los **salida** son los valores o resultados que se generan después de una operación o proceso.

Un ejemplo de algoritmo es el siguiente:

Problema: Encontrar el cuadrado de un número

ENTRADA
Inicio
Entero a, cuadrado,
Escriba ("Digite el numero para el que desea hallar el cuadrado");
Lea (a); ← **ENTRADA**
Cuadrado = a * a; ← **PROCESO**
Escriba ("el cuadrado del número es: "); ← **SALIDA**
Escriba (cuadrado);
Fin

Utilizando el ejemplo proporcionado en tu libreta elaborar la estructura de algoritmo de los siguientes problemas:

- ✓ Encontrar la suma de 2 números
- ✓ Encontrar la multiplicación de 2 números
- ✓ Encontrar el área de un triangulo
- ✓ Encontrar el área de un cuadrado

3. Cuadro Sinóptico

3. Lee la tabla y selecciona 15 palabras reservadas utilizadas en lenguaje C identificando su significado y en tu libreta elabora un cuadro sinóptico en el que se plasmen las palabras.

Puedes apoyarte en la siguiente información:

El lenguaje C está formado por un conjunto pequeño de palabras clave (reservadas) o comandos (keywords), y una serie de operadores. Hay cerca de 40 palabras clave, frente a las 150 del BASIC o 200 que poseen otros lenguajes, como el COBOL y el PASCAL.

A este conjunto de palabras se les denomina "palabras reservadas".

Auto	Especificador de clase de almacenamiento
Break	Instrucción
Case	Instrucción
Char	Especificador de tipo
Const	Modificador de clase de almacenamiento
Continue	Instrucción
Default	Etiqueta
Do	Instrucción
Doublé	Tipo de dato
Else	Instrucción
Extern	Especificador de clase de almacenamiento
If	Instrucción
Leng	Especificador de tipo de dato
Return	Instrucción
Short	Especificador de tipo
Static	Especificador de clase de almacenamiento
Struct	Especificador de tipo
Switch	Instrucción

4. Las estructuras de decisión son llamadas así precisamente porque tienen la funcionalidad de tomar acciones en base al resultado lógico de una decisión.

Entre las más utilizadas encontramos las siguientes:

La instrucción if es, por excelencia, la más utilizada para construir estructuras de control de flujo.

Switch es otra de las instrucciones que permiten la construcción de estructuras de control. A diferencia de if, para controlar el flujo por medio de una sentencia switch se debe de combinar con el uso de las sentencias case y break.

La sentencia do es usada generalmente en cooperación con while para garantizar que una o más instrucciones se ejecuten al menos una vez.

La sentencia break se usa para forzar un salto hacia el final de un ciclo controlado por for o por while.

4. Programas realizados

A continuación, podrás ver un ejemplo del uso de la sentencia IF en un programa realizado en C para acceder a un sistema mediante una clave de acceso para el usuario.

```
 #include <stdio.h> ------------------------- Librería

  Int main()------------------ Función Principal
  {
     int usuario,clave=18276;
     printf("Introduce tu clave: ");
     scanf("%d",&usuario);
     if(usuario==clave)
       printf("Acceso permitido");
     else
       printf("Acceso denegado");
}
```

Ejemplo 2: Programa para calcular si eres mayor de edad y puedes votar

```
#include <stdio.h>
int main(){
  int edad;
  printf("Escriba su edad: ");
  scanf("%d", &edad);
  if (edad >= 18){
    printf("Ya puedes votar");
  } else{
    printf("Todavia eres un niño");
  }
}
```

Tomando como base los ejemplos anteriores elaborar en tu libreta el código para los siguientes programas:

- ✓ Programa para calcular si eres adulto mayor de más de 60 años
- ✓ Programa para calcular si sueldo es mayor a 3000 pesos
- ✓ Programa para saber si eres hombre o mujer

5. Con los ejemplos de los programas anteriores observaste algunas palabras propias del lenguaje, realiza la lectura de la información que se te proporciona y de las siguientes palabras escribe su significado en tu libreta:	5. Conceptos de palabras reservadas

	✓ include ✓ Main ✓ Printf ✓ Scanf Puedes apoyarte en la siguiente información: Antes del proceso de compilación, el preprocesador es llamado primero a ejecutarse y buscar llamadas de instrucción al pre-procesador, la instrucción include le indica al preprocesador que cuando este se ejecute, el compilador debe incluir un archivo en el código. El método Main es el punto de entrada de un programa ejecutable; es donde se inicia y finaliza el control del programa. Mediante la función printf podemos escribir datos en el dispositivo de salida estándar (pantalla). Complementariamente a scanf, printf puede escribir cualquier combinación de valores numéricos, caracteres sueltos y cadenas de caracteres. La función printf transporta datos desde la memoria a la pantalla, a diferencia de scanf, que envía datos desde el teclado para almacenarlos en la memoria. La función printf devuelve el número de caracteres escritos. Si devuelve un valor negativo indica que se ha producido un error.	
Aprendizajes esenciales o Competencias esenciales 3er parcial	Estrategias de Aprendizaje	Productos a Evaluar
3. Utilizando estructuras de repetición	1. Las estructuras de repetición son las llamadas estructuras cíclicas, iterativas o de bucles. Permiten ejecutar un conjunto de instrucciones de manera repetida (o cíclica) mientras que la expresión lógica a evaluar se cumpla (sea verdadera). Apóyate de la siguiente lectura y realiza un resumen de las tres estructuras de repetición más utilizadas en la actualidad, identifica su funcionalidad y características ✓ (WHILE) Mientras ✓ (DO—WHILE) hacer – mientras ✓ (FOR) desde/para Puedes apoyarte en la siguiente información: Los bucles son estructuras que permiten ejecutar partes del código de forma repetida mientras se cumpla una condición. Esta condición puede ser simple o compuesta de otras condiciones unidas por operadores lógicos. **Sentencia / Bucle Whi**le	1. Resumen

	Con esta sentencia se controla la condición antes de entrar en el bucle. Si ésta no se cumple, el programa no entrará en el bucle.
	
	Naturalmente, si en el interior del bucle hay más de una sentencia, éstas deberán ir entre llaves para que se ejecuten como un bloque.
	
	Su sintaxis es:
	while (condición) sentencia;
	
	Sentencia / Bucle DO...WHILE
	Con esta sentencia se controla la condición al final del bucle. Si ésta se cumple, el programa vuelve a ejecutar las sentencias del bucle.
	
	La única diferencia entre las sentencias while y do...while es que con la segunda el cuerpo del bucle se ejecutará por lo menos una vez.
	
	Su sintaxis es:
	do {
	sentencia1;
	sentencia2;
	} while (condición);
	
	Sentencia / Bucle For
	La inicialización indica una variable (variable de control) que condiciona la repetición del bucle. Si hay más, van separadas por comas.
	
	Su sintaxis es:
	for (inicialización;condición;incremento) {
	sentencia1;
	sentencia2;
	}	

	2. Los operadores son símbolos que indican cómo se deben manipular los operandos. Los operadores junto con los operandos forman una expresión, que es una fórmula que define el cálculo de un valor. Los operandos pueden ser constantes, variables o llamadas a funciones, siempre que éstas devuelvan algún valor.	2. Tablas
	3. A continuación, se muestran dos imágenes con los operadores más utilizados en programación. **<u>Operadores Aritméticos</u>**	3. Pseudocódigo y Diagrama de flujo de 3 problemas

OPERADOR	PROPÓSITO
+	adición
-	sustracción
*	multiplicación
/	división
%	resto de división entera

Operadores lógicos y relacionales

OPERADOR	PROPÓSITO
>	mayor que
>=	mayor o igual que
<	menor que
<=	menor o igual que
==	igual
!=	distinto
&&	AND lógico
||	OR lógico
!	NOT lógico

En tu cuaderno copia las 2 tablas anteriores en donde se muestran los operadores aritméticos, lógicos y relacionales.

4. A continuación, se presenta un caso práctico sobre el uso del lenguaje C al ingresar 3 números nos debe arrojar como resultado si están en orden creciente o no.

```
#include <stdlib.h>
int main(void)
{
 int num1,num2,num3;
 printf("Introduzca número 1:");
 scanf("%d",&num1);
 printf("Introduzca número 2:");
 scanf("%d",&num2);
 printf("Introduzca número 3:");
 scanf("%d",&num3);


        if (num1<num2) {
                if (num2<num3) {
```

4. Elaborar el código para los 2 problemas propuestos

	printf("Orden creciente");
	} else {
	printf("No están introducidos en orden creciente ");
	}
	} else {
	printf("No están introducidos en orden creciente ");
	}
	system("PAUSE");
	return 0;
	}
	
	✓ Tomando como muestra el ejemplo anterior en tu libreta elaborar el código para los siguiente programas
	✓ Elaborar un programa para saber si un número es positivo o es negativo
	✓ Elaborar un programa para saber si un número es mayor a 100.	

	5. Utilizando tu libreta elaborar un resumen con todo lo aprendido durante el semestre con un mínimo de 1 hoja y un máximo de 3. ***NOTA IMPORTANTE: TODAS LAS ACTIVIDADES SERÁN REALIZADAS EN EL CUADERNO.***	5. Resumen

Aprendizajes esenciales			
Carrera:	TÉCNICO EN PROGRAMACIÓN	Semestre:	2o.
Módulo/Submódulo:	**Módulo I. Desarrolla software de aplicación con programación estructurada.** **Submódulo 3**. Aplica estructuras de datos con un lenguaje con un lenguaje de programación		

Aprendizajes esenciales o Competencias esenciales 1er parcial	Estrategias de Aprendizaje	Productos a Evaluar
1. Conceptos Generales de Estructura de Datos.	1. Realizar una investigación sobre los conceptos generales de las estructuras de datos, para poder realizar esta investigación favor de leer el ANEXO D. a. Vectores. b. Matrices. c. Listas. d. Pilas. e. Colas. f. Árboles binarios. g. Grafos. h. Montículos.	1. Glosario en el cuaderno 2. Cuadro Comparativa

ANEXO D.

Cuando estamos iniciando en el mundo de la programación, uno de los conceptos más difíciles de entender son las estructuras de datos y las definiciones que usualmente se encuentran un tanto enredadas.

¿Qué son las estructuras de datos?

Piensa en ellas como una forma de representar información. Así como usamos una variable de tipo array para representar un número finito de elementos, podemos representar una lista en una estructura de datos de tipo lista enlazada, esta estructura puede ser creada por nosotros o provista por una librería. Las estructuras de datos no solo representan la información, también tienen un comportamiento interno, y se rige por ciertas reglas/restricciones dadas por la forma en que está construida internamente.

¿Por qué son útiles?

Las estructuras de datos nos permiten resolver un problema de manera más sencilla gracias a que las reglas que las rigen nunca cambian, así que puedes asumir que ciertas cosas son siempre ciertas. Adicionalmente son dinámicas, si usas lenguajes de programación como Java, sabrás que necesitas definir el tamaño de los arrays antes de ser usados. Usando una estructura de datos, puedes hacer “un array” de tamaño indeterminado.

¿Por qué son importantes las estructuras de datos?

Son herramientas que podemos usar para resolver problemas complejos, manteniendo nuestro código relativamente sencillo, y probablemente también hagan nuestro código más rápido, pero hay que entenderlas a fondo para saber cuándo debemos usar una vs. Otra.

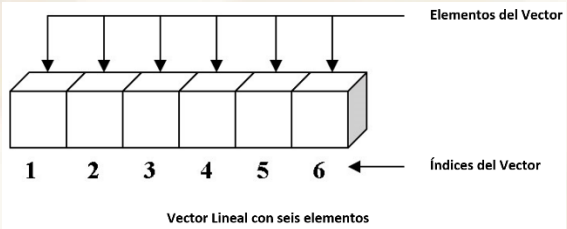
Tipos de estructuras de datos

Al hablar de estructuras de datos debemos pensar en primera instancia en cómo los datos se representan en la memoria, ¿se trata de estructuras contiguas o enlazadas?, al partir de esta pregunta podemos darnos la idea correcta sobre la base de nuestra estructura y cómo es que los datos se van a almacenar.

- Las estructuras **contiguamente asignadas** están compuestas de bloques de memoria únicos, e incluyen a los V*ectores*, M*atrices*, *Montículos*, y Tabla H*ash*.
- Las estructuras **enlazadas** están compuestas de distintos fragmentos de memoria unidos por *pointers* o punteros, e incluyen a las L*istas*, *Árboles*, y *Grafos*.
- Los **contenedores** son estructuras que permiten almacenar y recuperar datos en un orden determinado sin importar su contenido, en esta se incluyen las Pilas y *Colas*.

Array o Vector

Esta estructura es "la" fundamental de las estructuras contiguamente asignadas. *Arrays* ó arreglos son estructuras de datos de **tamaño fijo** de modo que cada elemento puede ser eficientemente ubicado por su *index* (índice) ó dirección. Un *array* tiene un tamaño fijo y una dirección (*índice*) con la cual podemos localizar de forma rápida un elemento en el arreglo, de modo que podemos apuntar a un elemento en el *array de la siguiente forma: vector[índice]*, donde array es el nombre de nuestra estructura, seguida de dos "corchetes" que abren y cierran, donde especificamos el que deseamos apuntar.



Vector Lineal con seis elementos

Los vectores muestran algunas ventajas con respecto a otras estructuras, como, por ejemplo:

- Al tener un espacio contiguo en memoria cada *índice* de cada elemento del *vector* apunta directamente a una dirección de memoria, de esta forma podemos acceder arbitrariamente a los datos de forma instantánea puesto que sabemos la dirección de memoria exacta. Esto deriva en un **acceso de tiempo constante** dado por los *índices.*
- Los *vectores* son puramente datos lo que significa que no es necesario desperdiciar espacio en memoria almacenando información extra que ayude a la localización de sus elementos como es el caso de las estructuras enlazadas, los *vectores tienen* **eficiencia de espacio**.
- **Localidad de memoria**, es común que en la programación los datos de una estructura sean iterados (o recorridos) y los vectores son buenos para esto ya que exhiben excelente localización de memoria, permitiéndonos aprovechar la alta velocidad de la caché en computadoras modernas.

La gran desventaja de los *vectores* es que no podemos ajustar su tamaño a la mitad de la ejecución de un programa, pero ¿y qué tal si creamos uno nuevo con la nueva dimensión deseada?; esto sería bueno si supiéramos el tamaño que deseamos todo el tiempo, pero si no lo sabemos sólo tenemos 2 opciones:

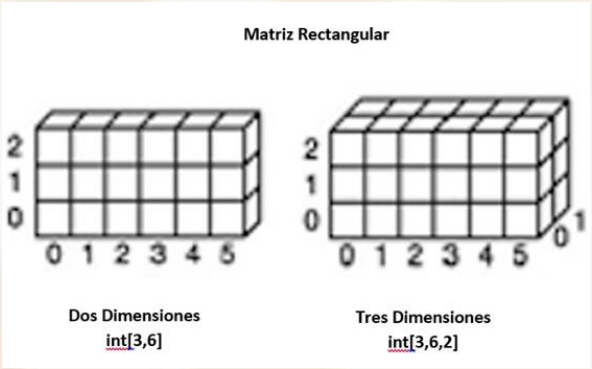
- crear una *vector* lo suficientemente grande para almacenar nuestros datos, pero esto deriva en un desperdicio de memoria totalmente innecesario.
- Podemos crear un nuevo *array*, doblar el tamaño de éste cada vez que se necesite crecer y copiar los datos del *arra*y anterior al nuevo vector, hacer esto tiene el mismo nivel de complejidad que si tuviéramos un vector único suficientemente grande, pero con la ventaja de que sólo va a crecer cuándo sea necesario, evitando así desperdiciar memoria.

Es pues que de esta forma podemos alargar un *array* en tiempo de ejecución, a este concepto se le conoce como ***dynamic arrays*** ó vectores dinámicos.

Los *vectores* son la base de muchos otros tipos de estructuras de datos como acabamos de ver con los arreglos dinámicos.

Matriz.

Este tipo de estructuras no son más que *vectores* de múltiples dimensiones. Pensemos, si un arreglo es una colección consecutiva de valores, se puede decir que son de tipo lineal en una dimensión plana ó unidimensionales, si quisiéramos formar una matriz podríamos crear un *vector* bidimensional, lo que quiere decir que tendremos que localizar a un elemento por su *índice [i , j]*, o bien su ubicación lineal a la derecha y hacia abajo como en un plano cartesiano con dimensiones *x, y* asemejando a una tabla, o podríamos crear un *vector* tridimensional, agregando una dimensión más de "fondo", que haría que localizáramos a un elemento en su *índice [i, j, k],* en plano cartesiano como *x, y, z*.



Estructuras enlazadas (linked structures)

La magia de las estructuras enlazadas es dada por los *pointers* o punteros, que como su nombre lo indica apuntan a una dirección de memoria donde se encuentra ubicado un valor. Los *pointers* son los encargados de mantener los "enlaces" entre valores de modo que es posible tener una secuencia de valores todos enlazados por *pointers*. Si ponemos atención a la definición de los *pointers* podemos deducir que los valores no necesariamente tienen que estar localizados en memoria uno después del otro como en el caso de los *vectores* que son contiguamente asignados, por lo que podemos tener los valores ubicados en distintas localidades de la memoria y aun así representar una colección de valores consecutivos.

Representación de una secuencia de valores enlazada por punteros.

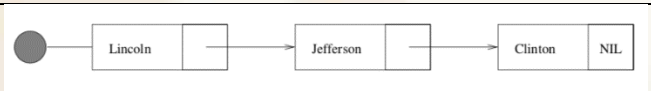
Como se observa en la imagen, tenemos una secuencia de valores enlazados por *pointers* llamados nodos, que están representados por un cuadro vacío y una flecha. El cuadro vacío representa el valor con la dirección de memoria a donde está apuntando el valor que contiene, de modo que: el valor 12 está enlazado por medio de un *pointer* que guarda la ubicación del siguiente valor en la secuencia que es el 19 y este a su vez guarda la ubicación al valor 37. Cuando hablemos de estructuras enlazadas debemos siempre pensar en que además de almacenar el valor que deseamos, debemos tener un espacio extra donde debemos almacenar la dirección de memoria del siguiente valor.

Después de una breve introducción a los *pointers* podemos pasar a la estructura más común de las estructuras enlazadas que son las *linked lists* o listas enlazadas.

Listas Enlazadas

Comencemos por definir una "lista". Una lista es aquella estructura que representa un número contable de valores ordenados donde un mismo valor puede repetirse y considerarse un valor distinto a otro ya existente. Las listas son consideradas secuencias de valores y cómo ya veíamos en la explicación anterior sobre *pointers* estos se pueden aplicar para implementar una lista de tal forma que las características principales de una lista serían:

- Cada nodo en nuestra lista contiene uno o más campos que contienen el valor que deseamos almacenar.
- Cada nodo contiene al menos un campo *pointer* apuntando a otro nodo, lo que significa que podemos tener 2 *pointers*, uno que apunta a un nodo consecuente y otro que apunta a un nodo previo formando así una lista doblemente enlazada (*double linked list*).
- Es necesario tener un *pointer* que apunte a la cabeza de la estructura para así saber por dónde comenzar.



Representación de una lista enlazada con sus respectivos punteros

La mayoría de los lenguajes de programación representan a las "cadenas" (palabras, secuencia de caracteres, etc) como *arrays* de caracteres, es por eso que con otra estructura como las *listas* podemos tener secuencias de palabras pudiendo representar un párrafo de un libro o de un post de medium que habla sobre estructuras de datos: p

Ahora debo mencionar que las 3 operaciones básicas soportadas por una *lista enlazada* son: búsqueda, inserción y eliminación de nodos. Suena obvio ¿no creen?, para explicar esto haremos una comparación sobre ambos tipos de estructuras y quizá así quede más claro.

Vectores vs Listas Enlazadas

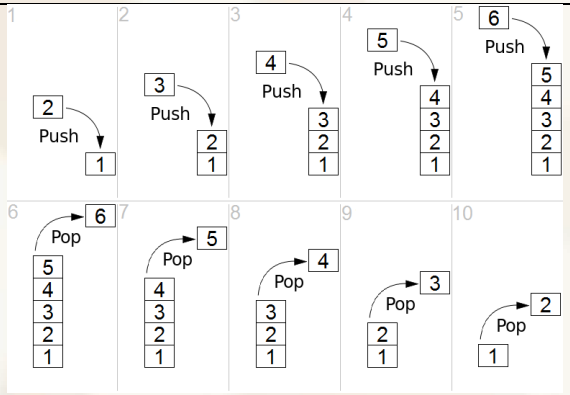
- Una diferencia grande entre los *arrays* vs *linked lists* es que insertar o eliminar de una *linked list* es más fácil ya que no tiene tamaño fijo y lo único que debemos hacer para insertar o eliminar un valor es simplemente apuntar al nuevo nodo creado, ó apuntar al siguiente nodo de la lista si un nodo fue eliminado. En un *array* no existe esta flexibilidad.
- Si tenemos una gran cantidad de valores siempre será más fácil mover *pointers* de un valor a otro que mover los valores en sí.
- En un *array* podemos provocar un desbordamiento de memoria (*memory overflow*) si queremos insertar un valor extra y ya hemos excedido el tamaño del *array*, lo cual no sucedería en una lista enlazada.
- Por otro lado, en una lista enlazada necesitamos más espacio en memoria para almacenar los *pointers.*
- Los *vectores* tienen un mejor manejo del acceso aleatorio a los valores y son mucho mejores en la ubicación de los datos en memoria y aprovechamiento de la caché.

Contenedores

Las estructuras de tipo contenedor se caracterizan principalmente por la forma particular de recuperación ordenada de datos que soportan, y en los dos tipos principales de contenedores (*plas* y *colas*) el orden de recuperación depende del orden de inserción.

Stack o Pila

Un *stack* o pila, soporta la recuperación ordenada de datos *last-in, first-out* (*LIFO*) o bien: el último dato en entrar, el primer dato en salir. De la misma forma en que se hace en una pila de platos limpios, si necesitamos un plato limpio vamos a la pila y tomamos el primero de la pila, que en realidad fue el último plato que agregamos a ella. Las pilas son estructuras que encontramos de muchas formas en el mundo real, siempre que podamos apilar algún objeto como libros, cazuelas, películas o la forma en la que metemos latas de refresco en el refrigerador :)



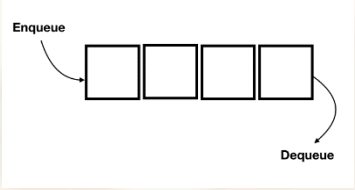
Representación de una pila y sus operaciones push y pop

Tomando en cuenta lo anterior podemos pensar que si usamos un *stack* es porque probablemente el orden de la recuperación de datos no nos importa tanto, simplemente queremos apilarlos y des apilarlos, por lo que las operaciones fundamentales en un *stack* son *push* y *pop* para poner y obtener datos de la pila.

Con *push()* insertamos un elemento en el tope de la pila, y con *pop()* retornamos y removemos el elemento en el tope de la pila, como se ve en la imagen.

Queue o cola

Una *queue* o cola, soporta la recuperación ordenada de datos *first-in*, *first-out* (*FIFO*) o bien: el primer dato en entrar, es el primer dato en salir. Justo como una cola en el banco cuando vamos a realizar alguna operación con nuestra cuenta bancaria, si todos los asistentes están ocupados se genera una cola donde el primero que llegó será el primero en ser atendido y el resto esperamos en la cola. Este tipo de estructuras se utiliza mucho en el control de tiempos de espera de servicios, tiempos de ejecución en un CPU, de conexión de red, o en el mundo real en una cola para las tortillas, para entrar en una avenida rápida, etc.



Representación de una queue y sus operaciones enqueue y dequeue

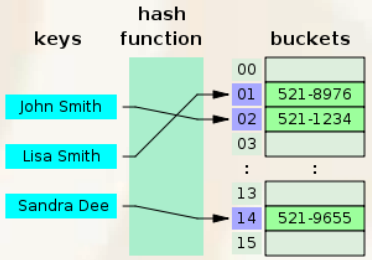
En este tipo de estructura el orden sí importa, pues siempre el primero de la cola debe ser el primero en ser atendido y el resto de forma ordenada esperan su turno. Las operaciones de esta estructura son: *enqueue* y *dequeue* para encolar y desencolar (poner y obtener de la cola).

De modo que *enqueue()* inserta un elemento al final de la cola, y *dequeue()* retorna y remueve el primer elemento de la cola.

Tabla Hash

Una estructura tabla hash es un diccionario que implementa una función ***hash*** a la cual le pasamos una cadena de caracteres (*string*) y esta nos devuelve un valor numérico asociado a esa cadena de caracteres.

Una función ***hash*** hace un mapeo de *cadenas* a números que debe ser **consistente**, lo que significa que cada *cadena* que entre a la función regresará siempre el mismo número. Además, debe poder mapear distintas *cadenas* a distintos números por lo que, para una *cadena* dada en el mejor caso posible debe mapearse a un número distinto.



Una tabla hash se puede implementar con un *array* y una función *hash*, pero de esta forma nuesta tabla *hash* sería muy simple y pronto encontraríamos complicaciones. Nuestro *array* serviría para almacenar los valores que deseamos y la función *hash* debería retornar un número por cada *string* dada que se encuentre dentro del número de casillas en nuestro *array*.

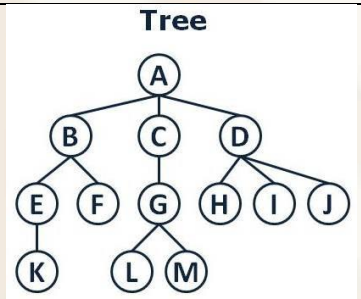
En nuestro ejemplo anterior si nosotros tenemos un *array* de tamaño 10 y nosotros pasamos a la función *hash* "Marcela", entonces el número 3 corresponde a la casilla número 3 del *array*, de modo que `vector[3] = valor a guardar`. Si nosotros fuéramos a guardar el password de un usuario podríamos hacerlo en una tabla *hash* , el valor que se almacenará en el *array* será el password y sólo tenemos que encargarnos de que nuestra función *hash* sea lo suficientemente buena para retornar un número consistente y no repetido por cada *cadena* que le mandemos como llave (*key*), suena fácil, ¿no?.

Trees (árboles)

Ya se han mencionado 5 estructuras de datos hasta ahora, se comentó sobre cuáles por sus características son mejores para búsquedas y actualización de datos, es por eso que ahora veremos una estructura que permita esta flexibilidad para buscar y actualizar datos de forma eficiente.

Binary Search Tree (BST)

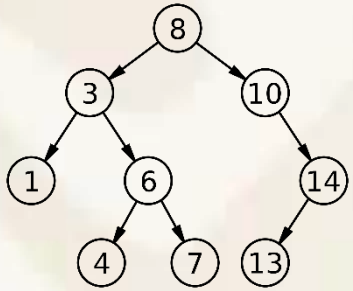
Comencemos por definir un *tree* (árbol). Un árbol es una estructura que consta de nodos y hojas. Cada nodo está conectado a otro nodo de forma jerárquica existiendo nodos "padre" y nodos "hijo" en distintos niveles, de modo que podamos crear una jerarquía desde un nodo raíz (*root*) hasta un último nivel de nodos hijo. Un *tree*, en realidad es un caso específico de un grafo, pero eso lo veremos más delante.



Árbol de letras

Ahora bien, un *binary search tree* (árbol de búsqueda binaria, BST), como su nombre lo dice, es un árbol que nos permitirá hacer búsquedas binarias, pero ¿a qué nos referimos con binarias? Bueno, pues el término binario se define como un compuesto de 2 partes, por lo tanto, nuestro árbol estará seccionado en 2 partes, la parte de la izquierda y la parte de la derecha partiendo de un nodo raíz. En un árbol binario, a la izquierda se encuentran los nodos cuyo valor es menor al del nodo raíz, y a la derecha los de mayor valor, de forma que: un árbol binario sólo acepta tener como máximo 2 nodos hijo (izquierda y derecha). Y ¿por qué esta estructura sería flexible para buscar y para actualizar datos?, al tener sólo 2 partes en dónde buscar es más fácil encontrar un elemento.

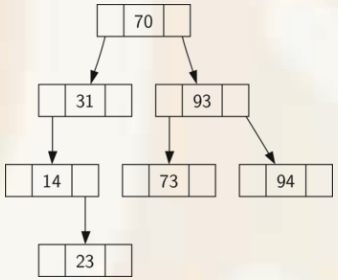
Juguemos un poco a adivinar el número que estoy pensando entre 1 y 20, ¿cómo podrías adivinar?, pues claro, intentando un número en ese rango, si dices 3, y yo digo no, el número que estoy pensando es más grande, ¿qué otro número dirías?, un acercamiento sería continuar mencionando uno por uno hasta que lo atines: ¿es el 4? no, ¿es el 5? no, ¿es el 6? no, ¿es el 7? no, ¿es el 8? no, ¿es el 9? sí, ¡tan tan! adivinaste después de 7 intentos, imagina que hubiera pensado el número 20, hubieran sido demasiados intentos, entonces, ¿podrías mejorar el número de intentos la próxima vez?, si, si hacemos una búsqueda binaria por menores y mayores. Así la próxima vez que pregunte qué número estoy pensando entre 1 y 20 podrías comenzar por dividir a la mitad y preguntar, ¿es el 10? y yo diría no, es más chico, entonces sabes que debes buscar números entre 1 y 9 por lo que, si divides en dos partes de nuevo el siguiente número sería el 5, ese tampoco es, pero es mayor que 5, entonces el siguiente número entre 5 y 9 sería 7, que tampoco es, pero es mayor, continua por la mitad y sería el 8 que no es, pero es mayor así que la última opción que queda es 9. El número de intentos se redujo a 5, que aún son muchos, pero si el número que hubiera pensando hubiera sido el 7 hubieras adivinado sólo en 3 intentos.



Binary search tree

Ahora ya sabes cómo buscar de forma binaria un dato, pero esto realmente podrías hacerlo en un *array* que contenga los 20 números, dividiendo en mitad el *array* hasta encontrar el que buscas, ¡ah! pero estás olvidando la flexibilidad para actualizar un dato, en un *array* actualizar datos es muy costoso pues son estructuras de tamaño fijo y a pesar que el acceso es muy rápido insertar o eliminar "constantemente" no es eficiente, es por eso que los árboles funcionan como lo hacen las *linked lists*, manteniendo un apuntador (*pointer*) a sus nodos hijos y separando siempre entre mayores y menores por lo que la estructura base de un árbol sería un nodo raíz o padre y sus nodos hijos, el menor a la izquierda y el mayor a la derecha (como se ve en la imagen en el tercer nivel en el nodo 6 con sus hijos 4 y 7 correspondientemente). Al mantener punteros hacia los nodos hijos es muy fácil insertar uno nuevo, por ejemplo, si deseamos agregar el número 5 en ese árbol de la imagen, ¿en dónde se intertaría?. El número 5 es menor que el nodo raíz 8, por lo tanto, debe ir a la izquierda del árbol, pero 5 > 3 a la derecha, 5 < 6 a la izquierda y 5 > 4, así que el 5

se insertaría como un nuevo nodo hijo del nodo 4 a su derecha. Y si quisiéramos remover el nodo 6, ¿cómo lo haríamos?, lo primero es buscar el 6 que es menor que 8 entonces vamos a la izquierda, pero 6 > 3 entonces a la derecha y una vez encontrado, como este nodo tiene hijos debemos elegir un nuevo "padre" para esos hijos, en este punto existen varias técnicas que definen las reglas para elegir al nuevo padre, yo elegiré tomar el nodo hijo que no tenga nodos hijos, que en este caso sería el 7, ahora el nodo 3 apunta a la derecha a 7 y 7 tiene a su izquierda al nodo 4, sin romper el esquema de mayores y menores hemos removido el nodo 6 sin mayor problema.



Binary search tree con punteros

En la imagen de la izquierda podemos observar de qué forma podemos representar un árbol con apuntadores de la misma forma en la que lo haría una *linked list*, pero en vez de mantener secuencias de datos mantenemos una jerarquía. Las operaciones fundamentales sobre un BST son: *insert(x)*, *remove(x)*, y *search(x)* como ya hemos visto en los ejemplos anteriores sobre insertar, remover y buscar en un árbol binario. Al igual que las *linked lists*, los árboles requieren de más espacio en memoria si mantenemos tantos punteros, esto es algo a considerar si decidimos usarla.

Los árboles de búsqueda binaria o BST, son muy comúnes debido a su fácil representación binaria para hacer búsquedas y actualización de datos de forma eficiente, ayudan a mantener una gran cantidad de datos de forma ordenada y aseguran una complejidad algorítimica muy baja en estos casos, sin embargo, no debemos olvidar otros conceptos de árboles como el balance. Cuando hablo de **balance** me refiero a que para que un BST sea "eficiente" debe estar balanceado, o que debe mantener (por lo menos) el mismo número de niveles de nodos en ambas partes. Imagina qué pasaría si en el ejemplo de adivinar el número insertamos del 1 al 20 consecutivamente en un árbol, ¿el árbol estaría balanceado? la respuesta es no, porque si comenzamos insertando el 1 como nodo raíz y continuamos con el 2 este se insertaría a su derecha y el 3 a la derecha del 2 y el 4 a la derecha del 3 y así sucesivamente hasta tener una secuencia 1=>2=>3=>4=>5 … 19=>20, esto realmente no representa un árbol binario balanceado, si qusiéramos buscar un número tendríamos que recorrer todos los niveles hasta encontrar el número lo cual no es nada eficiente. La forma correcta de balancear este árbol es hacerlo de la misma forma en la que buscamos, ¿cuál es la mitad de 20? es 10, entonces hacemos al 10 el nodo raíz y dividimos en 2 partes la inserción, de modo que 10 tiene como hijos a 5<=10=>15, 5 tiene como hijos a 2<=5=>7, 15 tiene como hijos a 12<=15=>17 … y así hasta balancear el árbol por completo. Ahora sí, si buscamos un número en este árbol balanceado lo podremos hacer de forma eficiente sin tener que recorrer todos los niveles en el peor de los casos.

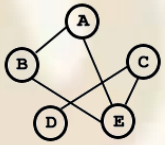
Los árboles son estructuras poderosas y los BST son los más usados, sin embargo, no es la única forma de representar datos sobre un árbol. Puesto que podemos definir jerarquías con distintas reglas, para una representación específica un nodo padre puede llegar a tener 6 hijos o más y así llegar a tener diferentes tipos de árboles como son los *Tries*, o la representación del código morse en un árbol que nos permita traducir mensajes en morse al idioma deseado, ambas representaciones de datos en árboles nos permiten la búsqueda fácil y rápida de elementos.

Grafos

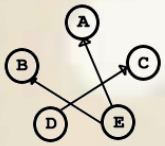
Empezaremos por una definición informal. Los grafos son un conjunto de puntos, de los cuales algún par de ellos está conectado por unas líneas. Si estas líneas son flechas, hablaremos de grafo dirigido (dígrafo), mientras que si son simples líneas estamos ante un grafo no dirigido.

Más formalmente se pueden definir como un conjunto de vértices y un conjunto de aristas. Cada arista es un par (u,v), donde u y v pertenecen al conjunto de vértices. Si este par es ordenado el grafo es dirigido.

Vamos a ver un par de ejemplos:
Grafo no dirigido.



Grafo dirigido.



Los grafos son uno de los temas **unificadores** de la informática y una representación abstracta que describe la organización de los sistemas de transporte, las interacciones humanas y las redes de telecomunicaciones. Que se puedan modelar tantas estructuras diferentes utilizando un solo formalismo es una fuente de gran poder para el programador educado.

Utilización de los grafos

Los campos de utilización de los grafos son muy variados, ya que los vértices pueden representar cualquier elemento (ciudades, aeropuertos, pc's...), y las aristas serían la conexión entre esos elementos (carreteras, pasillos aéreos, redes...). Por lo tanto, los grafos son muy usados en la modelización de sistemas reales.

Aprendizajes esenciales o Competencias esenciales 2º parcial	Estrategias de Aprendizaje	Productos a Evaluar
2. Operaciones básicas con Estructuras de Datos.	1. Leer el anexo E donde se explica las operaciones básicas de las siguientes estructuras de datos: 1. Pilas 2. Colas 3. Listas 4. Listas Ordenadas 5. Tabla Hash 6. Árbol binario de búsqueda 2. En base a la lectura anterior realizar el pseudocódigo de las siguientes estructuras de datos: 1. Pilas 2. Colas	1. Resumen en el cuaderno de la estructura de datos más simple y más complicada de implementar. 2. Pseudocódigos en el cuaderno.

ANEXO E.

Operaciones básicas de las pilas

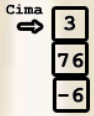
Vamos a estudiar las principales operaciones a realizar sobre una pila, insertar y borrar.

Insertar

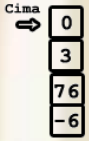
En primer lugar, hay que decir que esta operación es muy comúnmente denominada *push.*

La inserción en una pila se realiza en su cima, considerando la cima como el último elemento insertado. Esta es una de las particularidades de las pilas, mientras el resto de estructuras de datos lineales se representan gráficamente en horizontal, las pilas se representan verticalmente. Por esta razón es por lo que se habla de cima de la pila y no de cola de la cima. Aunque en el fondo sea lo mismo, el último elemento de la estructura de datos.

Las operaciones a realizar para realizar la inserción en la pila son muy simples, hacer que el nuevo nodo apunte a la cima anterior, y definir el nuevo nodo como cima de la pila.
Vamos a ver un ejemplo de una inserción:



Al insertar sobre esta pila el elemento 0, la pila resultante sería:



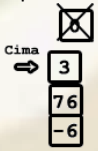
Borrar

Esta operación es normalmente conocida como *pop*.

Cuando se elimina un elemento de la pila, el elemento que se borra es el elemento situado en la cima de la pila, el que menos tiempo lleva en la estructura.

Las operaciones a realizar son muy simples, avanzar el puntero que apunta a la cima y extraer la cima anterior.

Si aplicamos la operación *pop* a la pila de 4 elementos representada arriba el resultado sería:



Operaciones básicas de las colas

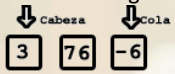
Pues en las colas como en toda estructura de datos las operaciones principales son insertar y eliminar, aunque en varias implementaciones de colas puedan recibir nombres diferentes.

Insertar

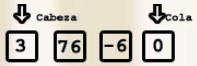
La inserción en las colas se realiza por la cola de las mismas, es decir, se inserta al final de la estructura.

Para llevar a cabo esta operación únicamente hay que reestructurar un par de punteros, el último nodo debe pasar a apuntar al nuevo nodo (que pasará a ser el último) y el nuevo nodo pasa a ser la nueva cola de la cola.

Vamos a verlo gráficamente sobre la siguiente cola:



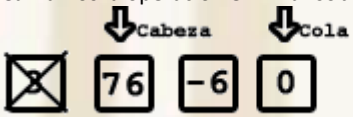
Si a esta cola le añadimos el elemento 0, la cola resultante sería:



Borrar

El borrado es una operación muy simple en las colas. Esta operación supone extraer la cabeza de la cola, ya que es el elemento que más tiempo lleva en la estructura. Para llevar a cabo esta operación únicamente hay que extraer el elemento situado en la cabeza de la cola y avanzar el puntero *cabeza* una posición, para que de esta forma la nueva cabeza sea el segundo elemento que más tiempo lleva en la cola.

Si realizamos la operación eliminar sobre la cola de 4 elementos del último gráfico el resultado sería el siguiente:



Una diferencia importante entre las colas y las listas, es que en las colas no se puede borrar un elemento cualquiera, se borra siempre el que está en la cabeza de la cola.

Operaciones básicas de las listas

En toda estructura de datos hay dos operaciones que sobresalen por encima del resto: Insertar y borrar. Estas dos operaciones aparecerán en toda estructura de datos, puede que, con otro nombre, o con una funcionalidad ligeramente diferente, pero su filosofía será la misma, proporcionar unas operaciones para la construcción de la estructura de datos.

Insertar

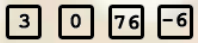
La operación insertar consiste en la introducción de un nuevo elemento en la lista.

En una lista no ordenada no es necesario mantener ningún orden, por lo tanto, la inserción de elementos se puede realizar en cualquier lugar de la lista, al principio, al final, en una posición aleatoria.

Generalmente se realiza la inserción de tal forma que la complejidad temporal sea mínima, es decir, que sea una operación sencilla para que se realice en el menor tiempo posible. La operación más sencilla depende de la implementación de la estructura de datos, en unos casos puede ser la inserción al inicio, en otros la inserción al final y en este caso la inserción la realiza en el segundo nodo de la lista.

Si tenemos la lista...

... e insertamos el elemento 0, la lista quedaría de la siguiente forma:

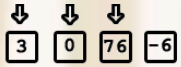


Borrar

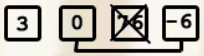
La operación borrar consiste en la eliminación de la lista de un elemento concreto. El elemento a borrar será escogido por el programador.

La eliminación en una lista no conlleva ningún trabajo adicional más que el propio de la eliminación del elemento en sí. Para borrar un elemento cualquiera habría que realizar un recorrido secuencial de la lista hasta encontrar el nodo buscado y una vez localizado reestructurar los punteros para saltarse el nodo a borrar y así poder eliminarlo.

Vamos a verlo con un ejemplo. Borrado del elemento 76 en la lista anterior:
Paso 1: Localizar el elemento a borrar.



Paso 2: Reestructurar punteros y eliminar nodo.



Otras operaciones

A partir de estas dos operaciones básicas cada lista puede presentar muchas operaciones diferentes, vamos a comentar algunas de ellas, dejando claro que las dos básicas que siempre aparecerán son las anteriores.

- Tamaño: Esta operación suele informar sobre el número de elementos que tiene en ese instante la lista.
- Buscar: Comprueba si existe un determinado elemento en la lista.
- Recorrer lista: Recorre toda la lista, realizando una operación en cada nodo. Por ejemplo, mostrar el contenido por pantalla.

Operaciones básicas de las listas ordenadas

Como en el caso de las listas no ordenadas hay dos operaciones fundamentales, insertar y borrar. El borrado de un elemento es idéntico en el caso de una lista ordenada que, en una lista no ordenada, en cambio la operación de inserción sí que es diferente.

Insertar

El procedimiento de añadir un nuevo elemento a una lista ordenada es ligeramente más complejo que en una lista desordenada. Cuando una lista es ordenada hay que mantener el orden siempre, por ello hay que tener especial cuidado al modificar la estructura de datos.

El procedimiento para insertar un elemento consta de dos pasos:

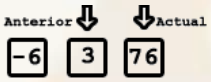
- Buscar el lugar adecuado en la lista para el elemento a insertar.
- Efectuar la inserción en el lugar buscado.

El proceso de búsqueda se suele realizar generalmente mediante una búsqueda secuencial. De esta forma se recorrerá la lista empezando por el inicio hasta encontrar el lugar adecuado para llevar a cabo la inserción.

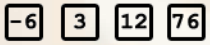
Vamos a verlo con un ejemplo:
Partimos de la siguiente lista

Si sobre esta lista se inserta el elemento 12, la lista sufrirá los siguientes cambios:

1. Búsqueda secuencial en la lista de la posición adecuada para insertar el elemento 12. La búsqueda se para al llegar al elemento 76, ya que es mayor que el elemento a insertar, y por lo tanto debe estar en una posición posterior al nuevo elemento. Para poder realizar posteriormente la inserción hay que recorrer la lista manteniendo dos referencias, una al nodo actual y otra al nodo anterior.



2. Inserción del elemento 12 en la posición entre el elemento 3 y el elemento 76. Para realizar esta operación se reestructuran los punteros del nodo que contiene al elemento 3 y del nuevo nodo. El nuevo nodo pasa a contener una referencia al nodo actual (elemento 76), mientras que el nodo anterior (elemento 3), pasa a apuntar al nuevo nodo.



Borrar

La operación borrar en listas ordenadas es igual que la operación borrar en listas no ordenadas, ya que la eliminación de un elemento nunca violará la condición de orden de esa lista, siempre y cuando la lista estuviese realmente ordenada antes de la eliminación.

Otras operaciones

Como en el caso de listas no ordenadas las operaciones adicionales que pueden incorporar las listas ordenadas es muy variado, pero también como en el caso anterior las dos operaciones principales son las descritas arriba.

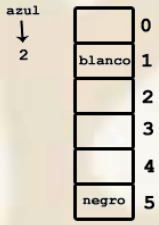
Como apunte se puede comentar el diferente tratamiento que se le da a la operación buscar en las listas ordenadas, ya que en estas estructuras se puede aplicar un algoritmo más rápido que la búsqueda secuencial aplicada en las listas no ordenadas. Este tratamiento es la búsqueda binaria, que consiste en dividir la lista en dos partes, comprobando en cuál de ellas debería estar el elemento (si existiese). Una vez localizada la mitad en la que se encuentra el objeto buscado se realizaría la misma operación recursivamente sobre esa mitad. Este proceso se propagaría hasta que se encuentre el elemento en el caso de que efectivamente exista, o hasta que el número de elementos de la mitad resultante sea 0 en el caso de que no exista.

Operaciones básicas de las tablas hash

Insertar

El proceso de inserción en una tabla hash es muy simple y sencillo. Sobre el elemento que se desea insertar se aplica la función de dispersión. El valor obtenido tras la aplicación de esta función será el índice de la tabla en el que se insertará el nuevo elemento.

Veamos este proceso con un ejemplo. Sobre la siguiente tabla hash se desea introducir un nuevo elemento, la cadena *azul*. Sobre este valor se aplica la función de dispersión, obteniendo el índice 2.



	0
blanco	1
	2
	3
	4
negro	5

El resultado de la inserción sería el siguiente:

	0
blanco	1
azul	2
	3
	4
negro	5

En el caso de que se produzca una colisión al tratar de insertar el nuevo elemento, el procedimiento será distinto en función del tipo de hash con el que se esté tratando.

- Encadenamiento separado: El nuevo elemento se añadirá al final de la lista que se inicia en la posición indicada por el valor que retornó la función de dispersión.
- Direccionamiento abierto: En este caso, se busca una nueva posición en la que almacenar el nuevo valor.

Borrar

El borrado en una tabla hash es muy sencillo y se realiza de forma muy eficiente. Una vez indicada la clave del objeto a borrar, se procederá a eliminar el valor asociado a dicha clave de la tabla.

Esta operación se realiza en tiempo constante, sin importar el tamaño de la tabla o el número de elementos que almacene en ese momento la estructura de datos. Esto es así ya que al ser la tabla una estructura a la que se puede acceder directamente a través de las claves, no es necesario recorrer toda la estructura para localizar un elemento determinado.

Si sobre la tabla resultante de la inserción del elemento *azul* realizamos el borrado del elemento *negro*, la tabla resultante sería la siguiente:

	0
blanco	1
azul	2
	3
	4
	5

Operaciones básicas de los árboles binarios de búsqueda

Como en toda estructura de datos hay dos operaciones básicas, inserción y eliminación.

Inserción

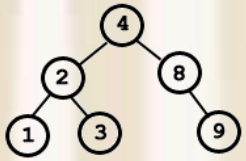
El procedimiento de inserción en un árbol binario de búsqueda es muy sencillo, únicamente hay que tener cuidado de no romper la estructura ni el orden del árbol.

Cuando se inserta un nuevo nodo en el árbol hay que tener en cuenta que cada nodo no puede tener más de dos hijos, por esta razón si un nodo ya tiene 2 hijos, el nuevo nodo nunca se podrá insertar como su hijo. Con esta restricción nos aseguramos mantener la estructura del árbol, pero aún nos falta mantener el orden.

Para localizar el lugar adecuado del árbol donde insertar el nuevo nodo se realizan comparaciones entre los nodos del árbol y el elemento a insertar. El primer nodo que se compara es la raíz, si el nuevo nodo es menor que la raíz, la búsqueda prosigue por el nodo izquierdo de éste. Si el nuevo nodo fuese mayor, la búsqueda seguiría por el hijo derecho de la raíz.

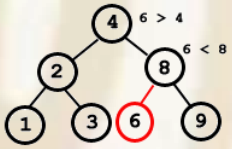
Este procedimiento es recursivo, y su condición de parada es llegar a un nodo que no tenga hijo en la rama por la que la búsqueda debería seguir. En este caso el nuevo nodo se inserta en ese hueco, como su nuevo hijo.

Vamos a verlo con un ejemplo sobre el siguiente árbol:



Se quiere insertar el elemento 6.

Lo primero es comparar el nuevo elemento con la raíz. Como 6 > 4, entonces la búsqueda prosigue por el lado derecho. Ahora el nuevo nodo se compara con el elemento 8. En este caso 6 < 8, por lo que hay que continuar la búsqueda por la rama izquierda. Como la rama izquierda de 8 no tiene ningún nodo, se cumple la condición de parada de la recursividad y se inserta en ese lugar el nuevo nodo.

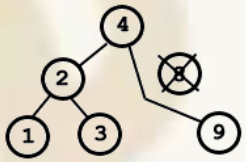


Borrar

El borrado en árboles binarios de búsqueda es otra operación bastante sencilla excepto en un caso. Vamos a ir estudiando los distintos casos.

Tras realizar la búsqueda del nodo a eliminar observamos que el nodo no tiene hijos. Este es el caso más sencillo, únicamente habrá que borrar el elemento y ya habremos concluido la operación.

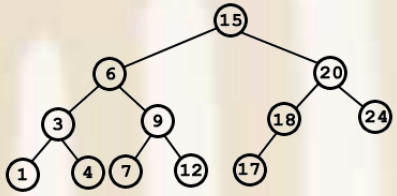
Si tras realizar la búsqueda nos encontramos con que tiene un sólo hijo. Este caso también es sencillo, para borrar el nodo deseado, hacemos una especie de *puente*, el padre del nodo a borrar pasa a apuntar al hijo del nodo borrado.



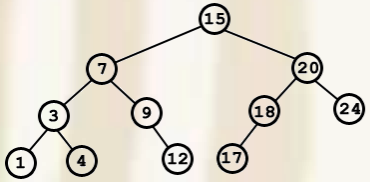
Por último, el caso más complejo, si el nodo a borrar tiene dos hijos. En este caso se debe sustituir el nodo a borrar por el mayor de los nodos menores del nodo borrado, o por el menor de los nodos mayores de dicho nodo. Una vez realizada esta sustitución se borra el nodo que sustituyó al nodo eliminado (operación sencilla ya que este nodo tendrá un hijo a lo sumo).

Sobre el siguiente árbol queremos eliminar el elemento 6. Tenemos dos opciones para sustituirlo:

- El menor de sus mayores: 7.
- El mayor de sus menores: 4.



Vamos a sustituirlo por el 7 (por ejemplo). El árbol resultante sería el siguiente, tras eliminar también el elemento 7 de su ubicación original.



Otras operaciones

En los árboles de búsqueda la operación buscar es muy eficiente. El algoritmo compara el elemento a buscar con la raíz, si es menor continua la búsqueda por la rama izquierda, si es mayor continua por la izquierda. Este procedimiento se realiza recursivamente hasta que se encuentra el nodo o hasta que se llega al final del árbol.

Otra operación importante en el árbol es el recorrido del mismo. El recorrido se puede realizar de tres formas diferentes:

- Preorden: Primero el nodo raíz, luego el subárbol izquierdo y a continuación el subárbol derecho.
- Inorden: Primero el subárbol izquierdo, luego la raíz y a continuación el subárbol derecho.
- Postorden: Primero el subárbol izquierdo, luego el subárbol derecho y a continuación la raíz.

Aprendizajes esenciales o Competencias esenciales 3er parcial	Estrategias de Aprendizaje	Productos a Evaluar
3. Métodos de Ordenamiento	3. Leer el anexo F donde se explica los diferentes métodos de ordenamiento: 1. inserción 2. Selección 3. Burbuja 4. Shell 5. Quick Sort 6. Fusión 4. En base a la lectura anterior realizar el pseudocódigo de los siguientes métodos de ordenamiento: 1. Burbuja 2. Selección	1. Realizar un cuadro comparativo sobre los diferentes métodos de ordenación 2. Pseudocódigo terminado en el cuaderno sobre los métodos de ordenación burbuja y selección.

ANEXO F.

Introducción.

Es la operación de arreglar los registros de una tabla en algún orden secuencial de acuerdo a un criterio de ordenamiento. El ordenamiento se efectúa con base en el valor de algún campo en un registro. El propósito principal de un ordenamiento es el de facilitar las búsquedas de los miembros del conjunto ordenado.

El ordenar un grupo de datos significa mover los datos o sus referencias para que queden en una secuencia tal que represente un orden, el cual puede ser numérico, alfabético o incluso alfanumérico, ascendente o descendente.

✓ ***Métodos de Ordenamiento Elementales:***

1. Inserción
2. Selección
3. Burbuja

✓ ***Métodos de Ordenamiento no Elementales:***

1. Shell
2. Quick Sort
3. Fusión

❖ ***Tipos de Ordenamiento:***

1. Ordenamiento Interno à Ordenamiento de datos en Memoria Principal. (La lectura y grabación se hacen en registros)
2. Ordenamiento Externo à Ordenamiento de datos en Disco.

❖ ***Tipos de Entrada de Datos:***

1. Entrada Ordenada = MEJOR CASO
2. Entrada Orden Inverso = PEOR CASO
3. Entrada Desordenada = CASO AL AZAR

❖ ***Tipos de Algoritmo***

1. Algoritmo Sensible: Modifica su tiempo de ejecución según el tipo de entrada.
2. Algoritmo No Sensible: Su tiempo de ejecución es independiente al tipo de entrada.
3. Algoritmo Estable: Aquellos que teniendo clave repetida, mantiene su posición inicial igual a la final.
4. Algoritmo No Estable: Aquello que no respetan la posición inicial igual que la final teniendo claves repetidas

ORDENAMIENTO POR SELECCIÓN

DESCRIPCIÓN.

- Ø Buscas el elemento más pequeño de la lista.
- Ø Lo intercambias con el elemento ubicado en la primera posición de la lista.
- Ø Buscas el segundo elemento más pequeño de la lista.
- Ø Lo intercambias con el elemento que ocupa la segunda posición en la lista.
- Ø Repites este proceso hasta que hayas ordenado toda la lista.

ANÁLISIS DEL ALGORITMO.

- Ø Requerimientos de Memoria: Al igual que el ordenamiento burbuja, este algoritmo sólo necesita una variable adicional para realizar los intercambios.
- Ø Tiempo de Ejecución: El ciclo externo se ejecuta n veces para una lista de n elementos. Cada búsqueda requiere comparar todos los elementos no clasificados.

Ventajas:

1. Fácil implementación.
2. No requiere memoria adicional.
3. Rendimiento constante: poca diferencia entre el peor y el mejor caso.

Desventajas:

1. Lento.
2. Realiza numerosas comparaciones.

ORDENAMIENTO POR INSERCIÓN DIRECTA

DESCRIPCIÓN.

El algoritmo de ordenación por el método de inserción directa es un algoritmo relativamente sencillo y se comporta razonablemente bien en gran cantidad de situaciones.

Completa la tripleta de los algoritmos de ordenación más básicos y de orden de complejidad cuadrático, junto con SelectionSort y BubbleSort.

Se basa en intentar construir una lista ordenada en el interior del array a ordenar.

De estos tres algoritmos es el que mejor resultado da a efectos prácticos. Realiza una cantidad de comparaciones bastante equilibrada con respecto a los intercambios, y tiene un par de características que lo hacen aventajar a los otros dos en la mayor parte de las situaciones.

Este algoritmo se basa en hacer comparaciones, así que para que realice su trabajo de ordenación son imprescindibles dos cosas: un array o estructura similar de elementos comparables y un criterio claro de comparación, tal que dados dos elementos nos diga si están en orden o no.

En cada iteración del ciclo externo los elementos 0 a i forman una lista ordenada.

ANÁLISIS DEL ALGORITMO.

- Ø Estabilidad: Este algoritmo nunca intercambia registros con claves iguales. Por lo tanto, es estable.
- Ø Requerimientos de Memoria: Una variable adicional para realizar los intercambios.
- Ø Tiempo de Ejecución: Para una lista de n elementos el ciclo externo se ejecuta n1 veces. El ciclo interno se ejecuta como máximo una vez en la primera iteración, 2 veces en la segunda, 3 veces en la tercera, etc.

Ventajas:

1. Fácil implementación.
2. Requerimientos mínimos de memoria.

Desventajas:

1. Lento.
2. Realiza numerosas comparaciones.

Este también es un algoritmo lento, pero puede ser de utilidad para listas que están ordenadas o semiordenadas, porque en ese caso realiza muy pocos desplazamientos.

MÉTODO DE LA BURBUJA

DESCRIPCIÓN

La idea básica del ordenamiento de la burbuja es recorrer el conjunto de elementos en forma secuencial varias veces. Cada paso compara un elemento del conjunto con su sucesor (x[i] con x[i+i]), e intercambia los dos elementos si no están en el orden adecuado.

El algoritmo utiliza una bandera que cambia cuando se realiza algún intercambio de valores, y permanece intacta cuando no se intercambia ningún valor, pudiendo así detener el ciclo y terminar el proceso de ordenamiento cuando no se realicen intercambios, lo que indica que este ya está ordenado.

Este algoritmo es de fácil comprensión y programación, pero es poco eficiente puesto que existen n-1 pasos y n-i comprobaciones en cada paso, aunque es mejor que el algoritmo de ordenamiento por intercambio.

En el peor de los casos cuando los elementos están en el orden inverso, el número máximo de recorridos es n-1 y el número de intercambios o comparaciones está dado por (n-1) * (n-1) = n^2 - 2n + 1. En el mejor de los casos cuando los elementos están en su orden, el número de recorridos es mínimo 1 y el ciclo de comparaciones es n-1.

La complejidad del algoritmo de la burbuja es O(n) en el mejor de los casos y O(n^2) en el peor de los casos, siendo su complejidad total O(n^2).

ORDENAMIENTO POR EL MÉTODO DE SHELL

DESCRIPCIÓN

El método Shell es una versión mejorada del método de inserción directa. Este método también se conoce con el nombre de inserción con incrementos crecientes. En el método de ordenación por inserción directa cada elemento se compara para su ubicación correcta en el arreglo, con los elementos que se encuentran en la parte izquierda del mismo. Si el elemento a insertar es más pequeño que el grupo de elementos que se encuentran a su izquierda, es necesario efectuar entonces varias comparaciones antes de su ubicación.

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

El Shell sort es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:

1. El ordenamiento por inserción es eficiente si la entrada está "casi ordenada".
2. El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.

El algoritmo Shell sort mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso del Shell sort es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.

ORDENAMIENTO QUICK SORT

DESCRIPCIÓN

El ordenamiento por partición (Quick Sort) se puede definir en una forma más conveniente como un procedimiento recursivo.

Tiene aparentemente la propiedad de trabajar mejor para elementos de entrada desordenados completamente, que para elementos semiordenados. Esta situación es precisamente la opuesta al ordenamiento de burbuja.

Este tipo de algoritmos se basa en la técnica "divide y vencerás", o sea es más rápido y fácil ordenar dos arreglos o listas de datos pequeños, que un arreglo o lista grande.

Normalmente al inicio de la ordenación se escoge un elemento aproximadamente en la mitad del arreglo, así al empezar a ordenar, se debe llegar a que el arreglo este ordenado respecto al punto de división o la mitad del arreglo.

Se podrá garantizar que los elementos a la izquierda de la mitad son los menores y los elementos a la derecha son los mayores.

Los siguientes pasos son llamados recursivos con el propósito de efectuar la ordenación por partición al arreglo izquierdo y al arreglo derecho, que se obtienen de la primera fase. El tamaño de esos arreglos en promedio se reduce a la mitad.

Así se continúa hasta que el tamaño de los arreglos a ordenar es 1, es decir, todos los elementos ya están ordenados.

En promedio para todos los elementos de entrada de tamaño n, el método hace O (n log n) comparaciones, el cual es relativamente eficiente.

ORDENAMIENTO POR MEZCLA

DESCRIPCIÓN

Mediante el enfoque de Dividir y conquistar, este algoritmo divide el arreglo inicial en dos arreglos donde cada uno contiene la mitad de los datos (partes iguales más o menos uno), y se ordenan mediante sucesivos llamados recursivos para luego fusionar los resultados en el arreglo inicial.

Este algoritmo se basa en una función que permite mezclar dos vectores ordenados, produciendo como resultado un tercer vector ordenado que contiene los elementos de los dos vectores iniciales, el cual tiene una complejidad de O(n) para mezclar dos arreglos, donde n es la suma de los tamaños de los dos arreglos.

El algoritmo principal que divide el arreglo, realiza O (log2 n) particiones y como llama a la función que mezcla los dos arreglos y que tiene una complejidad de O(n), entonces la complejidad total del algoritmo será de O(n log2 n).

COMPLEJIDAD

Cada algoritmo de ordenamiento por definición tiene operaciones y cálculos mínimos y máximos que realiza (complejidad), a continuación, se muestra una tabla que indica la cantidad de cálculos que corresponden a cada método de ordenamiento:

Algoritmo	Operaciones Básicas
Burbuja	**Ω** (n^2)
Inserción	Ω ($n^2/4$)
Selección	**Ω** (n^2)
Shell	Ω ($n \log^2 n$)
Marge	Ω (n log n)
Quick	Ω (n^2) en peor de los casos y Ω (n l g n) en el promedio de los casos