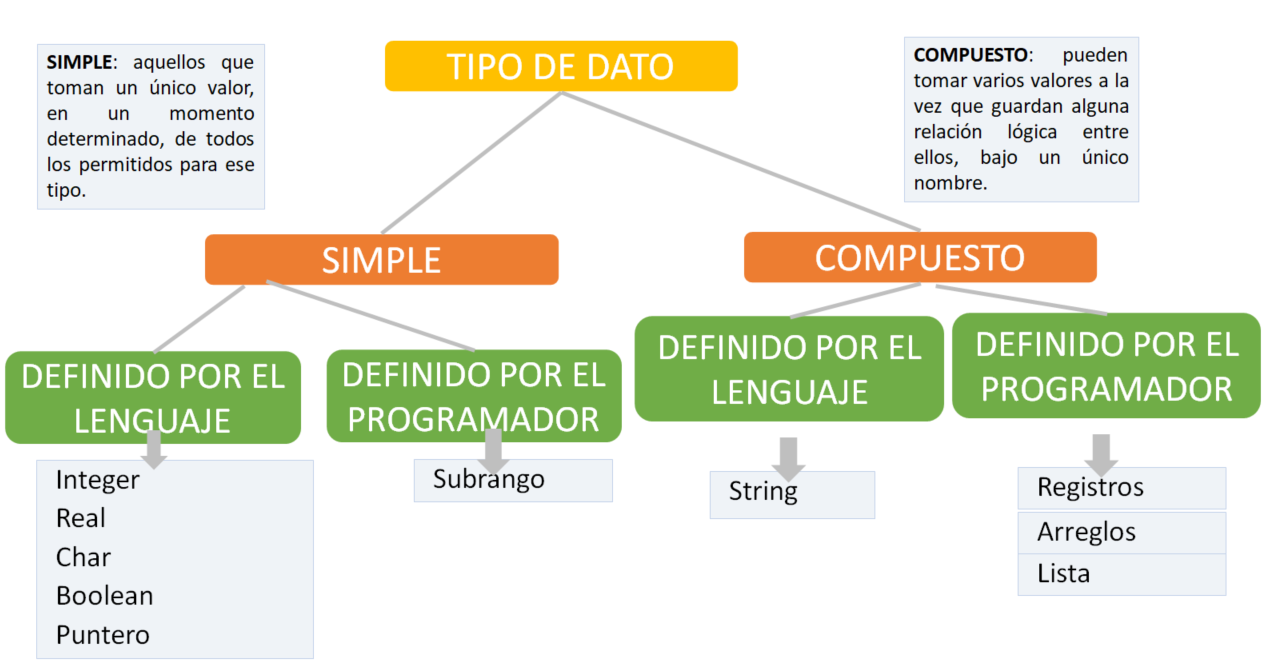
**Resumen CADP**

ALGORITMO: especificación rigurosa de la secuencia de pasos (instrucciones) a realizar sobre una máquina para alcanzar un resultado deseado en un tiempo finito.

Características:

* Alcanzar el resultado en tiempo finito: suponemos que un algoritmo comienza y termina.
* Especificación rigurosa.
* Si el autómata es una computadora, tendremos que escribir el algoritmo en un lenguaje “entendible” y ejecutable por la máquina.

DATOS: es una representación de un objeto del mundo real mediante la cual podemos modelizar aspectos del problema que se quiere resolver con un programa sobre una computadora. Puede ser constante o variable.

Características:

* Tienen un rango de valores posibles.
* Tienen un conjunto de operaciones permitidas.
* Tienen una representación interna.

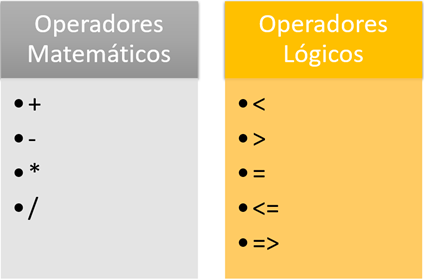
**Simples**: toman 1 UNICO valor, en un momento determinado, de todos los permitidos para ese tipo.

**Compuestos**: toman VARIOS valores a la vez que guardan alguna relación lógica entre ellos, bajo un único nombre.

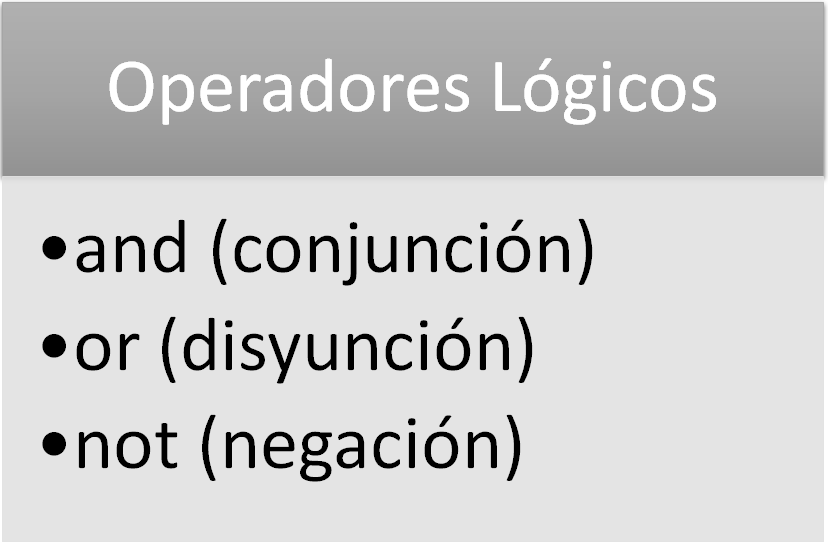


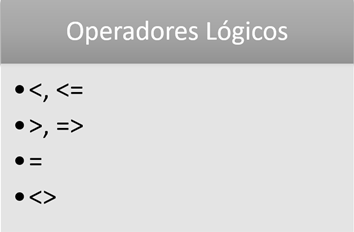
Tipos de datos:

Entero: tipo de dato simple, ordinal.



Real: tipo de dato simple, permiten representar números con decimales. NO funcionan con DIV y MOD.

Lógico: tipo de dato simple, ordinal. Permite representar V o F.



Carácter: tipo de dato simple, ordinal. Representa un conjunto finito y ordenado de caracteres que la computadora reconoce. Un dato de tipo carácter contiene solo un carácter.

Subrango: tipo de dato simple, ordinal que consiste de una sucesión de valores de un tipo ordinal (predefinido o definido por el usuario) tomado como base.

VARIABLES: es una zona de memoria cuyo contenido es un tipo de dato simple o compuesto. La dirección inicial de esta zona se asocia con el nombre de la variable. Puede cambiar su valor durante el programa.

- Índice (i): debe ser de tipo ordinal, no puede modificarse dentro del lazo, se incrementa y decrementa automáticamente, cuando el ‘for’ termina la variable índice no tiene valor definido.

CONSTANTE: es una zona de memoria cuyo contenido va a ser alguno de los tipos mencionados anteriormente. La dirección inicial de esta zona se asocia con el nombre de la variable. NO puede cambiar su valor durante el programa.

READ: es una operación que contienen la mayoría de los lenguajes de programación. Se usa para tomar datos desde un dispositivo de entrada (por defecto desde teclado) y asignarlos a las variables correspondientes. El usuario ingresa un valor, y ese valor se guarda en la variable asociada a la operación read.

WRITE: es una operación que contienen la mayoría de los lenguajes de programación. Se usa para mostrar el contenido de una variable, por defecto en pantalla. El valor almacenado en la variable asociada a la operación write, se muestra en pantalla.

ESTRUCTURAS DE CONTROL: conjunto mínimo de instrucciones que permiten especificar el control del algoritmo que se quiere implementar. Son: COMPLETAR

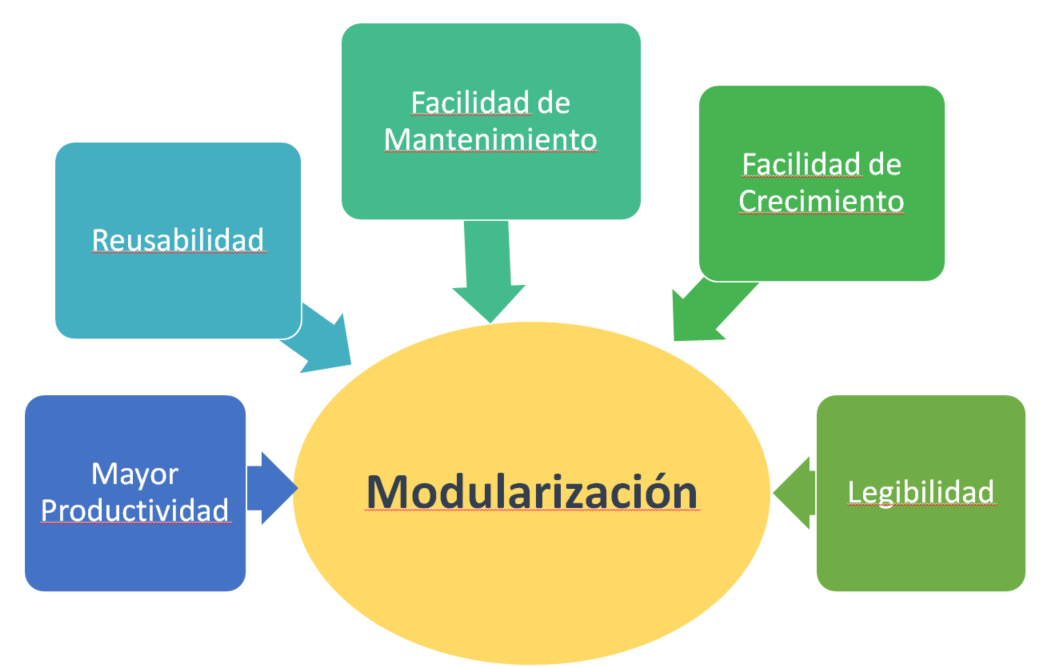
Secuencia

Decisión

Iteración

Repetición

Selección

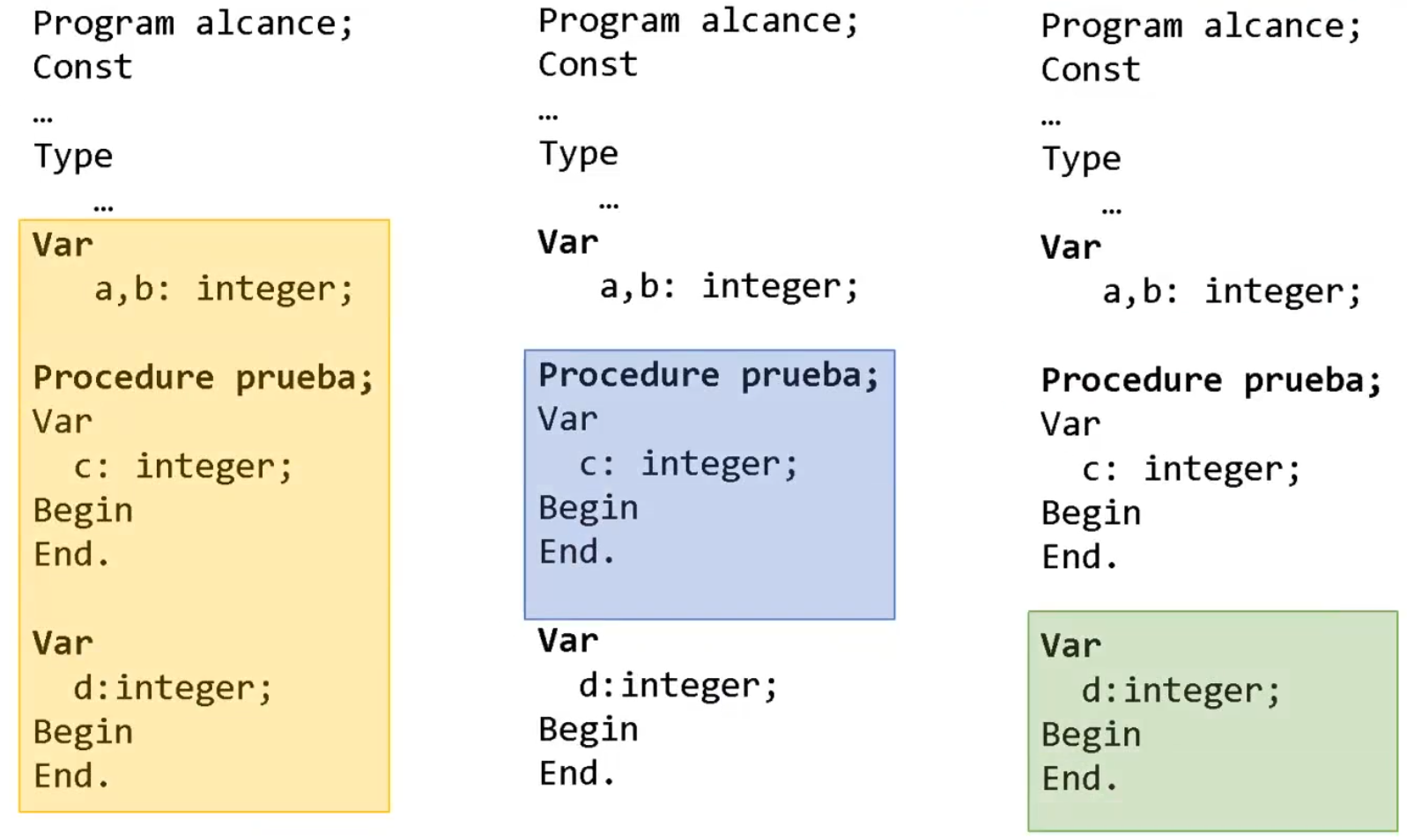


MODULARIZACIÓN: significa dividir un problema en partes funcionalmente independientes, que encapsulen operaciones y datos.

PROCEDIMIENTO (procedure): Conjunto de instrucciones que realizan una tarea específica y retorna 0, 1 o más valores. Estos pueden tener variables locales y recibir parámetros.

FUNCIÓN (function): Conjunto de instrucciones que realizan una tarea específica retorna un único valor. Estos pueden tener variables locales y recibir parámetros.

VARIABLES:

ALCANCE: dónde puedo usar las variables:

- Variables globales: Estas se declaran arriba de los procesos y son conocidas por todo el programa.

- Variables locales al proceso: Estas se declaran al principio de los procesos y pueden ser usadas solo en el proceso que en el que estén declaradas.

- Variables locales al programa: Estas se declaran al principio del programa principal y solo pueden ser utilizadas en el cuerpo del programa principal.

PARÁMETROS: permiten el ocultamiento de datos (los datos exclusivos de un módulo NO deben ser "visibles" o utilizables por los demás módulos).

El uso de parámetros significa que los datos compartidos se deben especificar como parámetros que se trasmiten entre módulos.

Si un módulo necesita solo datos dentro de su ejecución, esos datos deberían ser variables locales. Si el módulo necesita RECIBIR o DEVOLVER algo después de su ejecución, eso debería ser declarado como parámetro, por valor y por referencia respectivamente.

PARÁMETRO POR VALOR (num: integer): cuando el módulo recibe (sobre una variable local) un valor proveniente de otro módulo (o del programa principal). Con él puede realizar operaciones y/o cálculos, pero no producirá ningún cambio ni tampoco tendrá incidencia fuera del módulo.

PARÁMETRO POR REFERENCIA (VAR num: integer): cuando el módulo recibe el nombre de una variable (referencia a una dirección) conocida en otros módulos del sistema. Puede operar con ella y su valor original dentro del módulo, y las modificaciones que se produzcan se reflejan en los demás módulos que conocen la variable.



ESTRUCTURA DE DATOS

**VECTOR:** Se definen como una colección finita, homogénea y ordenada de elementos. Todo vector tiene un límite, es decir, debe determinarse cuál será el número máximo de elementos que podrán formar parte del mismo.

Características:

* Homogénea: los elementos son todos del mismo tipo.
* Estática: La dimensión física del vector no puede cambiar en el programa.
* No lineal: Un elemento del vector puede tener 0, 1 o más elementos antes o después.
* Directa: el acceso a cada elemento es de manera directa.

**OPERACIONES CON VECTORES**

- Agregar: significa poner al final de los elementos que tiene el vector un nuevo elemento. Puede pasar que esta operación no se pueda realizar si el vector está lleno.

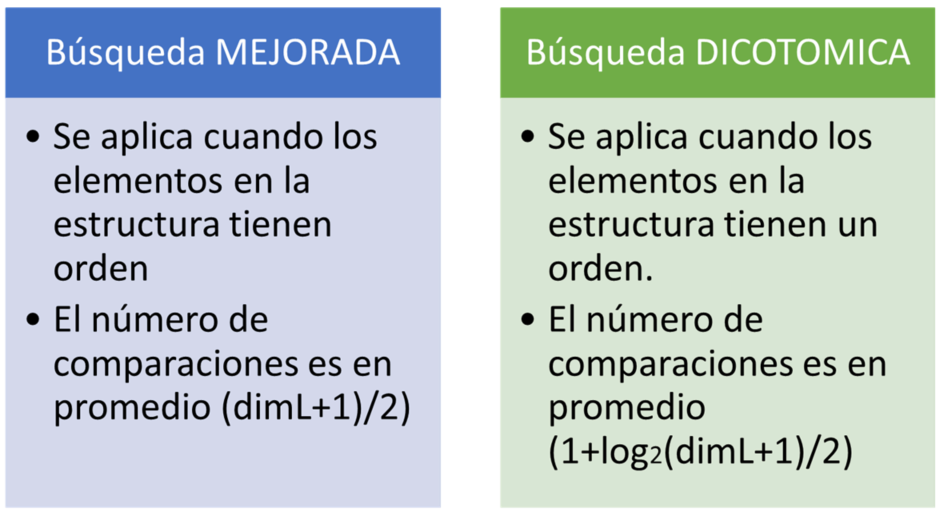
1. Verificar si hay espacio (cantidad de elementos actuales es menor a la cantidad de elementos posibles)
2. Agregar al final de los elementos ya existentes el elemento nuevo.
3. Incrementar la cantidad de elementos actuales.

- Insertar: significa agregar en el vector un elemento en una posición determinada. Puede pasar que esta operación no se pueda realizar si el vector está lleno o si la posición no es válida.

1. Verificar si hay espacio (cantidad de elementos actuales es menor a la cantidad de elementos posibles)
2. Verificar que la posición sea válida (esté entre los valores de dimensión definida del vector y la dimensión lógica).
3. Hacer lugar para poder insertar el elemento.
4. Incrementar la cantidad de elementos actuales.

- Eliminar: significa borrar (lógicamente) en el vector un elemento en una posición determinada, o un valor determinado. Puede pasar que esta operación no se pueda realizar si la posición no es válida, o en el caso de eliminar un elemento si el mismo no está.

1. Verificar que la posición sea válida (esté entre los valores de dimensión definida del vector y la dimensión lógica).
2. Hacer el corrimiento a partir de la posición y hasta el final.
3. Decrementar la cantidad de elementos actuales.

- Buscar: significa recorrer el vector buscando un valor que puede o no estar en el vector.

V desordenado: se debe recorrer todo el vector (en el peor de los casos), y detener la búsqueda en el momento que se encuentra el dato buscado o que se terminó el vector.

V ordenado: se debe aprovechar el orden, existen al menos dos formas: búsqueda mejorada y búsqueda dicotómica.

1. Se calcula el elemento que está en la posición del medio
2. Si es el elemento que busco, entonces la búsqueda termino
3. Si NO es el elemento que busco, entonces: Comparo contra el valor del medio - Elijo del vector la mitad que me convenga.

PUNTEROS: es un tipo de variable usada para almacenar una dirección en memoria. En esa dirección de memoria se encuentra el valor que puede ser de cualquiera de los tipos vistos (char, boolean, integer, real, string, registro, arreglo u otro puntero).

* Tipo de dato simple que contiene la dirección donde se encuentra almacenado el dato real.
* Pueden apuntar solamente a direcciones almacenadas en memoria dinámica (heap).
* Cada variable de tipo puntero puede apuntar a un único tipo de dato.

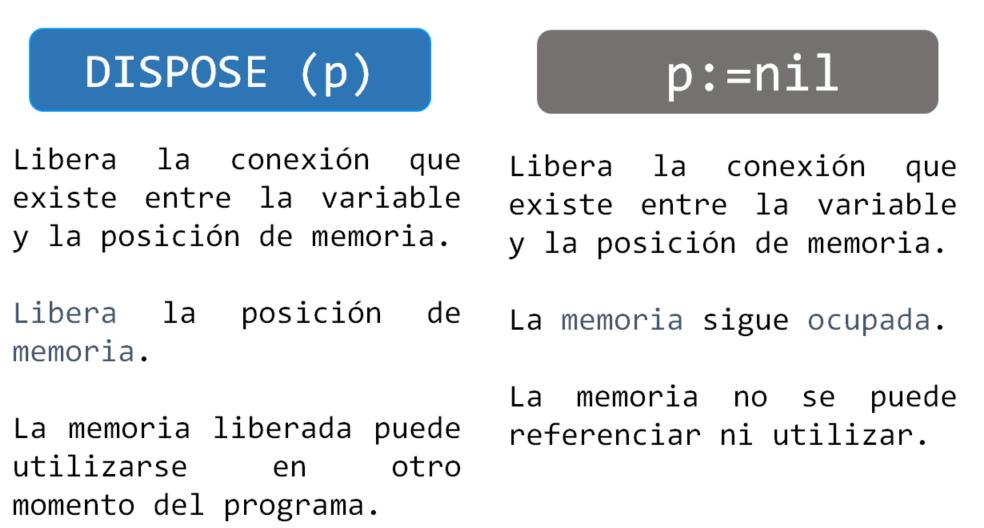
Una variable de tipo puntero (^) ocupa una cantidad de memoria fija, independiente del tipo de dato al que apunta (4 bytes). También puede reservar y liberar memoria durante la ejecución de un programa.

**No** se puede hacer **read (p)**, si p es una variable de tipo puntero.

**No** se puede hacer **write (p)**, si p es una variable de tipo puntero.

**No** se puede asignar una dirección a un puntero de manera manual **(p:= ABCD).**

**No** se puede comparar por mayor o menor direcciones de punteros **(p>q).**

- Creación: new(p); (busca en la memoria una memoria dinámica disponible y se asigna a la variable).

- Eliminación: dispose(p); (libera la memoria dinámica y se corta el enlace con la variable).

- Liberación: p:=nil (se asigna el valor nil al puntero, es decir, cortar el enlace pero no se libera la memoria dinámica)

-Asignación: implica asignar la dirección de un puntero a otro puntero, cualquier cambio que haga en la dirección de p este cambio va a repercutir en q.

-Contenido: implica acceder al contenido de la dirección de memoria con ^, por ej p^:=6;

MEMORIA DE UN PROGRAMA: memoria necesaria para la ejecución de un programa.

MEMORIA ESTATICA: se consideran sólo las variables locales y variables globales de programa.

MEMORIA DINAMICA: se considera sólo cuando en la ejecución de un programa se reserva o libera memoria:

* new(p) -> se reserva espacio para el contenido de p.
* dispose(p) -> se libera ese espacio.

l:= nil -> se corta el enlace pero el contenido anterior sigue ocupando memoria.

**LISTAS**

Colección de nodos, donde cada nodo contiene un elemento y en qué dirección de memoria se encuentra el siguiente nodo.

Cada nodo de la lista se representa con un puntero, que apunta a un dato (elemento de la lista) y a una dirección (donde se ubica el siguiente elemento de la lista).

Toda lista tiene un nodo inicial.

Los nodos que la componen pueden no ocupar posiciones contiguas de memoria. Es decir pueden aparecer dispersos en la memoria, pero mantienen un orden lógico interno.

Características:

* Homogénea: los elementos son todos del mismo tipo.
* Dinámica: la cantidad de nodos puede variar durante la ejecución.
* Lineal: cada nodo tiene un único antecesor y sucesor.
* Secuencial: el acceso a cada elemento es de manera secuencial.

- Crear: implica marcar que la lista no tiene una dirección inicial de comienzo -> l:= nil.

- Recorrer: implica posicionarse al comienzo de la lista y a partir de allí ir “pasando” por cada elemento de la misma hasta llegar al final.

1. Inicializo una variable auxiliar con la dirección del puntero inicial
2. mientras (no sea el final de la lista)

* proceso el elemento (por ejemplo imprimo)
* avanzo al siguiente elemento de auxiliar

- Agregar adelante: implica generar un nuevo nodo y agregarlo como primer elemento de la lista.

1. Reservo espacio en memoria para el nuevo elemento.
2. si (es el primer elemento a agregar)

* asigno al puntero inicial la dirección del nuevo elemento.

1. sino

* indico que el siguiente del nuevo elemento es el puntero inicial.
* actualizo el puntero inicial de la lista con la dirección del nuevo elemento.

- Agregar al final: implica generar un nuevo nodo y agregarlo como último elemento de la lista.

1. Reservo espacio en memoria para el nuevo elemento.
2. si (es el primer elemento a agregar)

* asigno al puntero inicial la dirección del nuevo elemento.

1. sino

* inicializo un puntero auxiliar aux
* mientras (no llegue al último elemento)
* avanzo en la lista.
* actualizo como siguiente del último nodo al nuevo elemento

ES INEFICIENTE (tiene que recorrer toda la lista)

Agregar al final MEJOR:

1. Reservo espacio en memoria para el nuevo elemento.
2. si (es el primer elemento a agregar)

* asigno al puntero inicial la dirección del nuevo elemento.
* asigno al puntero final la dirección del nuevo elemento.

1. sino

* actualizo como siguiente del puntero final al nuevo elemento
* actualizo la dirección del puntero final

- Buscar un elemento: implica recorrer la lista desde el comienzo pasando nodo a nodo hasta encontrar el elemento buscado o que se termine la lista.

1. Inicializo una variable auxiliar con la dirección del puntero inicial
2. mientras (no sea el final de la lista y no encuentre el elemento)

* si es el elemento buscado detengo la búsqueda
* sino
* avanzo al siguiente elemento

- Insertar: se necesita que la estructura tenga un orden e implica agregar el elemento a la lista de manera que la misma siga ordenada.

1. Generar un nuevo nodo (nuevo).

2. Si la lista está vacía

* Actualizo la dirección del nodo inicial (PI)

3. Sino

* Preparo los punteros para el recorrido (ant,actual)
* Busco la posición
* Si va al pricipio
* Asigno como siguiente del nodo nuevo al nodo inicial
* Actualizo la dirección del nodo inicial (PI)
* Si va en el medio
* La dirección del siguiente del puntero ant es la dirección del nodo nuevo
* La dirección del siguiente del nodo nuevo es la dirección del actual
* sino
* La dirección del siguiente del puntero ant es la dirección del nodo nuevo
* La dirección del siguiente del nodo nuevo es la dirección nil

- Eliminar: implica recorrer la lista desde el comienzo pasando nodo a nodo hasta encontrar el elemento y en ese momento eliminarlo (dispose). El elemento puede no estar en la lista.

1. Si el primer nodo contiene el elemento a eliminar

* Actualizar la dirección siguiente del puntero inicial pI con la dirección siguiente de actual.
* Actualizar la dirección del puntero inicial pI
* Hacer el dispose

2. sino

* Actualizar la dirección siguiente del puntero ant con la dirección siguiente de actual.
* Hacer el dispose

CORRECIÓN: ver si el programa funciona para las condiciones solicitadas.

Técnicas para corrección de programas:

- Testing: Provee evidencias convincentes del que el programa cumple el trabajo esperado. Se analiza el programa y si se encuentra un error se arregla y se vuelve a testear hasta que no tenga más errores.

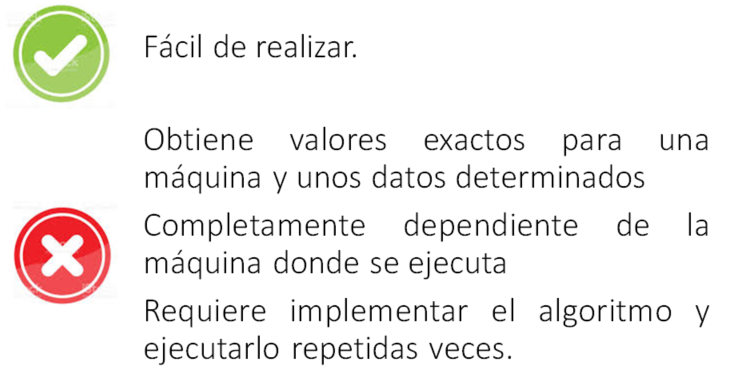
- Debugging: Es el proceso de descubrir y reparar la causa del error. Por ej agregar sentencias en el programa para encontrar el problema.

- Walkthroughs: Significa recorrer y analizar el programa con terceros.

- Verificación: Verificar que un programa se cumplan la pre y post condiciones del mismo.

TIEMPO DE EJECUCIÓN: el tiempo de un algoritmo puede definirse como una función de entrada.

Existen algoritmos donde el tiempo de ejecución no depende de las características de los datos de entrada sino de la cantidad de datos de entrada o su tamaño.

Existen otros algoritmos donde el tiempo de ejecución es una función de la entrada “específica”, en estos casos se habla del tiempo de ejecución del “peor” caso. En estos casos, se obtiene una cota superior del tiempo de ejecución para cualquier entrada.

ANÁLISIS EMPÍRICO: es necesario realizar el programa y medir el tiempo consumido.

ANÁLISIS TEÓRICO: implica encontrar una cota máxima para expresar el tiempo de nuestro algoritmo, sin necesidad de ejecutarlo. Se obtiene el tiempo teórico del algoritmo y a partir de allí obtiene el orden del mismo.

Dado un algoritmo que es correcto se calcula el tiempo de ejecución de cada una de sus instrucciones. Para eso se va a considerar:

* Sólo las instrucciones elementales del algoritmo: asignación, y operaciones aritmético/lógicas.
* Una instrucción elemental utiliza un tiempo constante para su ejecución, independientemente del tipo de dato con el que trabaje -> 1UT.
* Se considera que cada operación elemental se ejecuta en una unidad de tiempo.

SI SE DESEA CALCULAR EL TIEMPO DE EJECUCIÓN:

* Sólo se consideran operaciones aritmético lógicas ( + - / \* DIV, MOV), and, or, not, y asignaciones -> 1 UT c/u.
* Si tenemos un **if** -> 1UT para evaluar c/condición del if (si es if x and x = 3 UT) + MAX (tiempo del then, tiempo del else).
* Si tenemos un **while** -> (N+1) \* (evaluar c/condición) + N \* (el tiempo que requiere el cuerpo).
* Si tenemos un **for i** -> (3N+2) + N \* (el tiempo que requiere el cuerpo).