RESUMEN LO MAS COMPLETO QUE PUDE HACER PARA CADP PORQUE RENDI EL FINAL 4 VECES MAL Y CON ESTO ESPERO APROBAR O SINO ME ECHAN DE CASA

Este resumen se divide en 3 partes:  
1- Vas a encontrar toda teoría sacada de la catedra con un poco de lo que dijo Laura en sus clases  
2- Vas a encontrar definiciones de ARRAYS REGISTROS Y LISTAS sacadas del libro  
3- Códigos de vectores y listas como por ejemplo Agregar ordenado o eliminar un elemento  
\* Las vas a identificar por colores o también podés buscarlo con el buscador propio de Word con los códigos (11111111,22222222,33333333)

(PARTE 1: 11111111 )

Algoritmo: Especificación rigurosa de la secuencia de instrucciones a realizar sobre un autómata para alcanzar un resultado deseado en un tiempo finito.  
-Alcanzar un resultado en un tiempo finito: Suponemos que un algoritmo comienza y termina. Esta implícito que el numero de instrucciones debe ser también.  
-Especificación rigurosa: Que debemos expresar un algoritmo en forma clara y univoca.  
-Si el autómata es una computadora, tendremos que escribir el algoritmo en un lenguaje entendible y ejecutable por la maquina

Programa = Algoritmo + dato

Dato: Clases de objetos ligados a un conjunto de operaciones para crearlos y manipularlos. Un dato es todo aquello que quiero representar del mundo real en un problema. Puede ser constante o variable.  
Tienen:  
-Rango de valores posibles  
-Conjunto de operaciones permitidas  
-Representación interna

Características que tiene un programa:

\*Para el desarrollador:  
-El programa debe utilizar la función para la cual fue creado  
-El código debe ser fácil de leer y entender.  
-El código de un programa debe estar descompuesto en módulos que cumplan las subfunciones del sistema.  
-Todo el proceso de análisis y diseño del problema y su resolución debe estar documentado mediante texto y/o gráficos (Operatividad, legibilidad, organización, documentación).

Dato numérico: Representa números, enteros o reales  
Enteros: Tipo de dato simple, ordinal (puede variar en la ejecución del programa). Ordinal: tiene un antecesor y un predecesor. Al tener una representación interna, tiene un máximo y un mínimo.

Real: permite representar números con decimales, no es ordinal. Tiene representación interna y un rango.

Orden de precedencia para la resolución:  
1: Operadores \*, /, div, mod

2: Operadores +, -

Dato lógico: Permite tomar dos valores (T,F / V , F)  
Operaciones: And, or ,not

Dato carácter: Representa un conjunto finito y ordenado de caracteres que la computadora reconoce. Un dato de tipo carácter contiene un valor carácter  
-Tipo de dato simple, ordinal.

Variables: Se declaran con un nombre que son guardados en una zona de memoria y puede cambiar su valor.  
Constantes: En una zona de memoria, cuyo contenido va a ser alguno de los tipos mencionados. No puede cambiar su valor durante el programa.  
  
Precondición: Es la información que se conoce como verdadera antes de iniciar el programa.  
Postcondición: Información que debería ser verdadera al concluir el programa o modulo, si se cumplen adecuadamente los pasos especificados.

Read: Operación que contienen la mayoría de lenguajes. Sirve para poder ingresarle un valor a una variable desde un periférico.

Estructuras de control:  
Secuencia: Una instrucción debajo de la otra que sabemos que se van a ir ejecutando en el orden en el que fueron declaradas. (READ,WRITE)  
Decisión: Necesarias para tomar decisiones. Es un algoritmo representativo de un problema real (IF,ELSE)  
Iteración: Las acciones se ejecutan dependiendo de la evaluación de la condición. Estos pueden ser pre o post condicionales.   
Selección: Permite realizar distintas acciones dependiendo del valor de una variable tipo ordinal(Case(variable)/ of)). Ventaja: Puedo aplicar rangos entre valores ordinales

Precondicional: Evalúan la condición y si es verdadera se ejecuta el bloque de acciones. Dicho bloque se puede ejecutar 0/UNA/MAS VECES (WHILE)  
IMPORTANTE: El valor inicial de la condición debe ser conocida o evaluable antes de la evaluación de la condición.  
Postcondicional: Primero ejecuta y luego evalúa, seguro se ejecuta 1 o más veces. (REPEAT/ UNTIL)

Repetición: Consiste en repetir N veces el bloque de acciones (FOR)  
-Índice: Ordinal, el índice no puede modificarse, se incrementa y decrementa automáticamente. Cuando el for termina el índice no tiene valor definido.  
EN RESUMEN:  
Precondicional: Evalúa la condición y en caso de ser verdadera ejecuta.  
-Se repite mientras la condición es verdadera.  
-Puede ejecutarse 0,1 o mas veces.  
Postcondicional: Ejecuta las acciones y luego evalúa la condición  
-Se repite mientras la condición es falsa  
-Se ejecuta 1 o más veces.

¿Cuándo las uso?   
-Pre: Cuando el ultimo dato no se debe procesar  
-Post: Cuando el ultimo dato se debe procesar

Comunicación entre módulos:  
-Variables globales: Se declaran al principio del programa y pueden ser vistas o accedidas por cualquier modulo, por el programa principal y parámetros.  
\*Muchos identificadores   
\*No se especifica la comunicación entre módulos  
\*Conflictos de nombres de identificadores utilizados por diferentes programadores  
\*Probabilidad de perder integridad de los datos, al modificar involuntariamente en un módulo datos de alguna variable que luego deberá utilizar otro modulo.  
-Parámetros: Son la solución al problema de las variables globales porque permite la ocultación de datos y uso de parámetros  
\*Ocultamiento: El modulo solo indica que recibe y que devuelve.  
\*El uso de parámetros: Significa que los datos compartidos se deben especificar como parámetros que se transmiten entre sí.

Parámetro por valor: Es un dato de entrada, el modulo recibe un valor y cualquier modificación que le hagamos no va a hacer que el dato cambie.

Tipos de datos definidos por el usuario, Permiten:  
-Aumentar la expresividad de nuestro programa  
-Mayor seguridad a la hora de realizar operaciones sobre cada clase de datos.  
-Limites prestablecidos sobre los valores posibles que pueden tomar las variables que corresponden al tipo de dato.

Tipos de datos definidos por el usuario: Es aquel que no existe en el lenguaje y el programador es el que los define y los especifica. Se agregan en la zona del type.  
Ventajas:  
-Documentación: Podemos definir tipos de datos con nombres mas agradables o entendibles (no como este resumen). Facilitando el entendimiento y lectura del programa  
-Flexibilidad: Porque podemos hacer que tome otros valores modificando una declaración en lugar de un conjunto de declaraciones de variables  
-Seguridad: Al garantizar el programador que datos quiere que tome puede decir que si no toma uno de esos valores el programa da error porque el usuario esta ingresando un dato incorrecto.

Subrango: Tipo ordinal, simple, y existe en la mayoría de lenguajes.  
-Se basa en valores que ya existen en el programa  
-Operaciones permitidas: Asignación, comparación, las permitidas para el tipo base  
-No permitidas: Depende del tipo base

Modularización: Dividir un problema en partes funcionalmente independientes que encapsulan operaciones y datos.  
Se trata de: Separar en funciones lógicas con datos propios y datos de comunicación perfectamente especificados.  
Modulo: Tarea especifica bien definida que se comunican entre si adecuadamente y cooperan para conseguir un objetivo común. Estos encapsulan acciones o funciones.  
Ventajas:  
-Mayor productividad: Podemos trabajar en conjunto y en simultaneo con más programadores, acortando el tiempo de desarrollo de programa.  
-Reusabilidad: El módulo puede reutilizarse más de una vez  
-Facilidad de crecimiento: Podemos seguir agregando módulos de una manera más sencilla  
-Legibilidad: Es más fácil entender y leer el código

Procedimiento: Conjunto de instrucciones que realizan una tarea especifica y retorna 0,1 o más valores  
Función: Conjunto de tareas especificas que retornan un único valor de tipo simple

Alcance de variables:  
-Variables globales: Pueden ser usadas en todo el programa (incluyendo módulos)  
-Locales al proceso: Usadas solo en el proceso que están declaradas  
-Local de programa: Solo en el cuerpo del programa

Variables en módulos:  
1ro: Busca si es local  
2do: Si es parámetro  
3ero: Si es global

Variables en un programa:  
-Se busca si es variable local al programa  
-Se busca si es global

Estructura de dato: Permite al programador definir un tipo al que se asocian diferentes datos que tienen valores lógicamente relacionados y asociados bajo un nombre único  
Estas estructuras pueden clasificarse en 4 tipos

|  |  |  |  |
| --- | --- | --- | --- |
| Elemento | Acceso | Tamaño | Lineabilidad |
| Homogenea | Secuencial | Dinamica | Lineal |
| Heterogenea | Directo | Estatica | No Lineal |

Homogénea: Los elementos que la componen son del mismo tipo  
Heterogénea: Los elementos que la componen pueden ser de distinto tipo

Estática: Cuando declaramos la variable en el programa el lenguaje ya sabe cuanta memoria va a ocupar. No varia durante la ejecución del programa.  
Dinámica: Puede variar durante la ejecución del programa (El tamaño)  
Secuencial: Para acceder a un quinto elemento no puedo acceder directamente, sino que tengo que pasar por sus anteriores.  
Directo: Si queremos acceder un elemento podemos ir directamente sin necesidad de pasar por los anteriores a él. Referenciando una posición.  
Lineal: Cada elemento puede tener un elemento que le sigue y uno que le precede.  
No lineal: Jerárquicas o tipo árbol: para cada elemento pueden seguirle 0,1, o más elementos.  
Registros: Tipo de datos estructurados, tiene una estructura definida que permite agrupar diferentes datos en una estructura bajo un nombre único (Heterogénea, estática, acceso mediante campos)

Campos: Toda información que queremos representar que podemos almacenar y para nosotros significa un registro.  
Asignación: Es la única operación para una variable de tipo registro

Arreglo (Array): Estructura de datos compuesta que permite acceder a cada componente por una variable índice.  
Vector: Arreglo de una dirección. Los elementos se guardan consecutivamente en la memoria y se pueden referenciar a través de un índice. (Homogénea, estática, indexado). Posee un índice ordinal.  
-Cuando creamos una variable de tipo vector se reserva espacio para todos los elementos.  
Formas de recorrerlo:   
-Total: Se recorre todo el vector  
-Parcial: Se recorre hasta una cierta posición

Punteros característicos:  
-Tipo de dato simple que contiene la dirección donde se encuentra almacenado el dato real  
-Puede apuntar solamente a direcciones almacenadas en memoria dinámica.  
-Cada variable de tipo puntero puede apuntar a un único tipo de dato  
-Se indican con ^ y ocupa 4 bytes en memoria.  
-Pueden reservar y liberar memoria durante la ejecución del programa

Operaciones con punteros:  
-Creación de una variable puntero  
-Destrucción o dispose  
-Asignación entre variables puntero  
-Comparación de una variable puntero

Nil: Corta el enlace, pero la memoria queda ocupada  
Dispose: Libera la conexión que existe entre la variable y la posición de memoria

Con punteros NO SE PUEDE:  
-Leer o escribir punteros  
-Asignar una dirección a un puntero de manera manual  
-Comparar por mayor o por menor direcciones de punteros

Memoria estática: Variables globales, constantes y locales  
Dinámica: Solo cuando se reserva o libera memoria (new / dispose)

Lista: Estructura de datos dinámica, homogénea, lineal, secuencial. Consiste en una sucesión de nodos donde cada uno contiene un elemento y en qué dirección de memoria se encuentra el siguiente nodo. Toda lista tiene un nodo inicial.  
-Los nodos que la componen pueden no ocupar posiciones contiguas de memoria, es decir pueden aparecer dispersos en memoria, pero mantienen un orden lógico.  
\*Homogénea: Los elementos son del mismo tipo  
\*Dinámica: Cantidad de nodos pueden variar durante la ejecución  
\*Lineal: Cada nodo tiene un único antecesor y sucesor

Operaciones de Listas:  
-Creación de una lista  
-Agregar nodos al comienzo  
-Recorrido de la lista  
-Insertar nodos  
-Eliminar nodos de una lista

Tecnicas de corrección:  
\*Un programa es correcto si: Se realiza de acuerdo a sus especificaciones, si cumple su función.

-Testing: Verificamos los casos limite y tratamos de ver la mayoría de casos por los que el programa no funciona. Para esto se plantea un plan de pruebas donde se analizan los casos de prueba si hay errores se corrigen. Estos pasos se realizan hasta que no haya errores.  
-Debuggin: Agregar a un código que no funciona correctamente sentencias que nos permiten ver que esta pasando en el programa (Ej: write/writeln). Se descubren y reparan la causa del error. Los errores pueden ser sintácticos, lógicos o de sistema.   
-Walktrhough: Recorrer el programa frente a una audiencia. Se muestra un código y se pregunta si funciona, por qué si y por qué no.

Eficiencia: Estudia el tiempo que tarda un algoritmo en ejecutarse y la memoria que requiere.  
2 tipos de análisis:  
-Empírico: Para realizarlo es necesario realizar programa y medir el tiempo concurrido. Este análisis es fácil de realizar, obtiene valores exactos pera una maquina y unos datos determinados.  
Contras: Completamente dependiente de la maquina donde se ejecuta. Requiere implementar el algoritmo y ejecutarlo varias veces.  
-Teórico: Implica una cota máxima que expresan el tiempo de nuestro algoritmo, sin necesidad de ejecutarla.

|  |  |  |
| --- | --- | --- |
| Estructura de control | Formula | Formula acortada |
| IF | Evaluar condición + el que pese mas entre el IF o el ELSE | Condición + Máximo C + Max |
| For | 3 \* Veces que itera + 2 + Veces que itera \* Cuerpo del for | 3\*N+2+N\*C |
| While | Veces que itera + Condiciones + Veces que itera \* Cuerpo | N+ Condición + N\* Cuerpo |
| Repeat Until | Veces que itera + 1 + Veces que itera \* Cuerpo + Cuerpo | N+1+N\*C+C |

A PARTIR DE ACA TODO LO QUE SIGUE ES SACADO DEL LIBRO

(PARTE 2: 22222222)

ARRAYS

Un array o arreglo es una secuencia de datos del mismo tipo que se numeran consecutivamente. Puede almacenar diferentes tipos de datos y se utiliza para organizar información. Los elementos se acceden mediante índices, comenzando desde 0 hasta n-1. Cada ítem del array se denomina elemento. Los elementos de un array se numeran, como ya se ha comentado, consecutivamente 0, 1, 2, 3,... Estos números se denominan valores índice o subíndice del array.

Los tipos TipoIndice1, TipoIndice2 ... TipoIndiceL tienen que ser de un tipo simple ordinal, es decir integer, char, boolean, enumerado o subrango.

Además de las operaciones permitidas con un array y la forma de acceder a sus elementos, hay que destacar las siguientes características comunes al tipo array, independientemente de su tamaño o de su dimensión:

• Los arrays son estructuras homogéneas, en el sentido de que sus elementos componentes son todos del mismo tipo.

• El tamaño˜ del array queda fijado en la definición y no puede cambiar durante la ejecución del programa, al igual que su dimensión. Así, el tamaño˜ de cualquier variable del tipo

VECTOR

En terminos generales, un vector es una secuencia, de longitud fija, formada por elementos del mismo tipo. Teniendo en cuenta que un vector es un array de dimensión 1, su definición es sencilla

Un vector es una estructura de datos que representa una secuencia ordenada de elementos del mismo tipo. Se utiliza para almacenar y acceder a múltiples valores relacionados entre sí de manera eficiente. Algunas de las principales características de un vector son:

Tipo de dato: Un vector puede almacenar elementos de cualquier tipo de dato, ya sea datos simples como números enteros, decimales, caracteres, etc., o incluso objetos de clases personalizadas.

Tamaño fijo: Un vector tiene un tamaño fijo y predefinido al momento de su creación. Esto significa que se reserva una cantidad específica de memoria para almacenar los elementos y no se puede modificar su tamaño directamente.

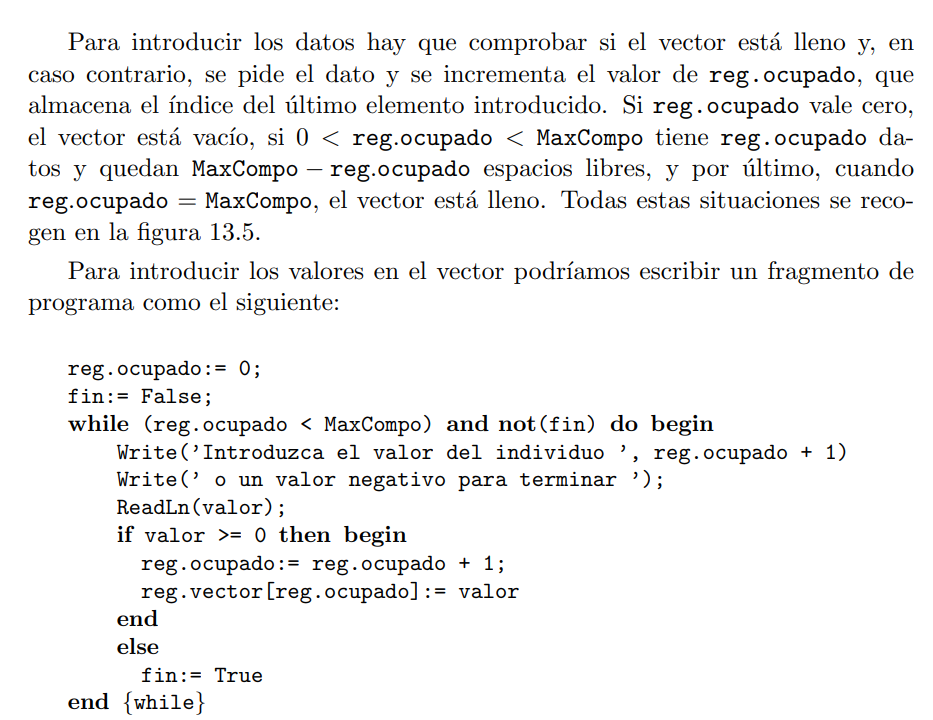
Acceso por índice: Los elementos de un vector se acceden mediante un índice entero. El primer elemento se encuentra en la posición 0, el segundo en la posición 1 y así sucesivamente. Esta característica permite un acceso rápido y directo a los elementos.

Almacenamiento contiguo: Los elementos de un vector se almacenan de manera contigua en la memoria, lo que facilita el acceso secuencial y el cálculo de direcciones de memoria.

Eficiencia en el acceso y búsqueda: Debido a que los elementos se almacenan de manera contigua y se accede mediante índices, el acceso y búsqueda en un vector son eficientes en términos de tiempo de ejecución.

Operaciones de inserción y eliminación costosas: Dado que un vector tiene un tamaño fijo, las operaciones de inserción y eliminación de elementos pueden ser costosas en términos de tiempo y recursos. Esto implica que agregar o eliminar elementos en posiciones intermedias puede requerir el desplazamiento de otros elementos.

En resumen, un vector es una estructura de datos utilizada para almacenar y acceder a una secuencia ordenada de elementos del mismo tipo. Es eficiente en el acceso y búsqueda, pero las operaciones de inserción y eliminación pueden ser costosas.



REGISTROS   
Descripción del tipo de datos registro Los registros1 son otro tipo de datos estructurados muy utilizados en Pascal. Su principal utilidad reside en que pueden almacenar datos de distintos tipos, a diferencia de los demás datos estructurados. Un registro estará formado por varios datos (simples o estructurados) a los que llamaremos campos del registro y que tendrán asociado un identificador al que llamaremos nombre de campo.

Es conveniente destacar que el tipo de datos registro, al igual que el tipo array, es un tipo estructurado de tamaño fijo; sin embargo, se diferencian de ellos principalmente en que los componentes de un array son todos del mismo tipo, mientras que los componentes de un registro pueden ser de tipos distintos.

• Para acceder a los campos de los registros se utilizan construcciones de la forma nomVarRegistro.nomCampo, es decir, el nombre de una variable de tipo registro seguido de un punto y el nombre del campo al que se quiere acceder. En este punto se debe señalar que, a diferencia de los arrays, en los cuales el acceso se realiza por medio de ´índices, en los registros se accede por medio de los identificadores de sus campos, que deben darse explícitamente.

• Los tipos de los campos pueden ser tipos predefinidos o definidos por el programador mediante una definición de tipo previa. Incluso un campo de un registro puede ser de tipo registro

Las operaciones de lectura y escritura de registros han de hacerse campo por campo, empleando procedimientos o funciones especiales si el tipo del campo así lo requiere.

LISTAS  
  
Una lista es una colección lineal de elementos que se llaman nodos. El termino colección lineal debe entenderse de la siguiente manera: tenemos un primer y un último nodo, de tal manera que, a cada nodo, salvo el ultimo, le corresponde un único sucesor, y a cada nodo, salvo el primero, le corresponde un único predecesor. Se trata, pues, de una estructura de datos cuyos elementos están situados secuencialmente.

Una definición del tipo listas se puede realizar usando punteros y registros. Para ello consideramos que cada nodo de la lista es un registro con dos componentes: la primera almacena el contenido del nodo de la lista y, la segunda, un puntero que señala al siguiente elemento de la lista, si ´este existe, o con el valor nil en caso de ser el último. Esta construcción de las listas recibe el nombre de lista enlazada dinámica

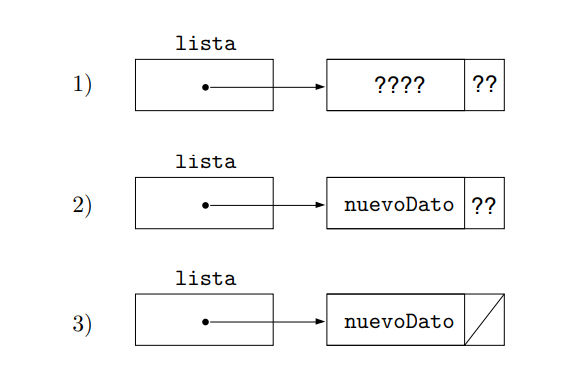
Esencialmente, una lista será representada como un puntero que señala al principio (o cabeza) de la lista.  
  
Inserción de elementos:  
Supóngase que nuestra lista está iniciada con el valor nil. Para introducir un elemento nuevoDato en ella, habrá que completar la siguiente secuencia de pasos:

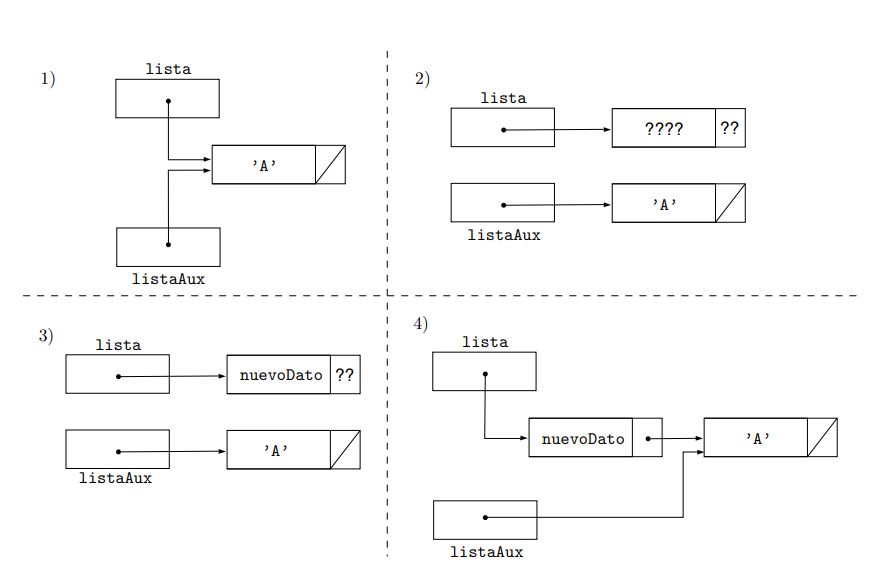
1. En primer lugar, habrá que generar una variable del tipo tNodo, que ha de contener el nuevo eslabón: esto se hace mediante la sentencia New(lista).

2. Posteriormente, se asigna nuevoDato al campo contenido del nodo recién generado. La forma de esta asignación dependerá del tipo de datos de la variable nuevoDato.

3. Y, por último, hay que anular (con el valor nil) el campo siguiente del nodo para indicar que es el último de la lista.

Para empezar, se desarrolla un procedimiento para añadir un elemento al principio de una lista, atendiendo al diseño descrito por los pasos anteriores:





Para insertar un nuevoDato al principio de una lista no vacía, lista, se ha de proceder:

1. Una variable auxiliar listaAux se usa para apuntar al principio de la lista con el fin de no perderla. Esto es:

listaAux:= lista

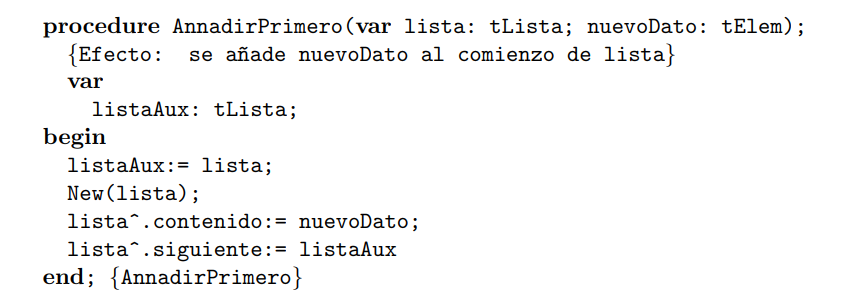
2. Después se asigna memoria a una variable del tipo tNodo (que ha de contener el nuevo elemento): esto se hace mediante la sentencia New(lista).

3. Posteriormente, se asigna nuevoDato al campo contenido del nuevo nodo:

lista^.contenido:= nuevoDato.

4. Y, por último, se asigna la listaAux al campo siguiente del nuevo nodo para indicar los demas elementos de la lista, es decir:

lista^.siguiente:= listaAux



Otras operaciones sobre listas:

1. Determinar el número ´ de elementos de una lista

2. Leer o modificar el k-´esimo elemento de la lista.

3. Insertar o eliminar un elemento en la k-´esima posición.

4. Insertar o eliminar un elemento en una lista ordenada.

5. Combinar dos o más listas en una única lista.

6. Dividir una lista en dos o más listas.

7. Ordenar los elementos de una lista de acuerdo con un criterio dado.

8. Insertar o eliminar un elemento en la k-´esima posición de una lista de acuerdo con un criterio dado.

9. Buscar si aparece un valor dado en algún lugar de una lista.

-HASTA ACA TODO LO TEORICO-

(PARTE 3: 33333333)

**Corte de control (aplicado a registros)**

* Necesito que los datos vengan ordenados.
* Es para poder saber la cantidad de elementos que tengo de cada tipo.
  + Forma:

Program uno;

Type

Perro = record

Raza : string;

Nom : string;

Edad : integer;

End;

//modulos

Var

Ani : perro;

Cant : integer;

Actual : string;

Begin

readln (ani);

While (ani.raza <> ‘XXX’) do begin

Cant := 0;

Actual := ani.raza;

While ((ani.raza <> ‘XXX’) and (ani.raza = actual)) do begin

Cant := cant +1;

readln (ani);

end;

writeln(‘La cantidad de la raza ’, actual,’ es ‘, cant);

end;

end.

**Arreglos**

* Tipo de dato compuesto definido por el programador
* Es una estructura de datos compuesta que permite acceder a cada componente por una variable índice que da la posición de la componente dentro de una estructura de datos.
* Es una colección de elementos que se guardan consecutivamente en la memoria y se pueden referenciar a través de un índice.
  + **Homogénea**: pq todos los datos del vector son del mismo tipo.
  + **Estática**: El tamaño no cambia durante la ejecución (se calcula en el momento de compilación).
  + **Indexada**: para acceder a cada elemento de la estructura se debe utilizar una variable ‘índice’ que es de tipo ordinal.
* Para declararlo se utiliza la palabra ‘array’:
  + **Vector = array [rango] of tipo;**
  + El ‘rango’ debe ser de un tipo ordinal: char, entero, booleano, subrango.
  + El ‘tipo’ debe ser un tipo estático: char, entero, booleano, subrango, real, registro, vector.
* Para cargar un vector se utiliza ‘for i’:
  + For i := 1 to tam do v[i] := valor. (tam = tamaño del vector)

**Recorridos**

* **Recorrido total:** implica analizar todos los elementos del vector, lo que lleva a recorrer completamente la estructura.
  + Se utiliza for i := 1 to tam.
* **Recorrido parcial:** implica analizar los elementos del vector, hasta encontrar aquel que cumple con lo pedido. Puede ocurrir que se recorra todo el vector.
  + Se utiliza un while (pq hay un límite).

**Dimensión Física y Lógica**

* **Dimensión Física:** Se especifica en el momento de la declaración y determina la cantidad máxima de elementos que se pueden guardar en el arreglo.
  + Normalmente se pone como constante ya que es un valor que no puede cambiar con la ejecución del programa, siempre se mantiene igual.
* **Dimensión Lógica:** Se determina cuando se cargan contenidos a los elementos del arreglo. Indica la cantidad de posiciones de memoria ocupadas con contenido real. Nunca puede superar la dimensión Física.

**Agregar/insertar y eliminar elementos**

* **Agregar:** agrega en el vector un elemento detrás del último elemento cargado en el vector. Puede pasar que esta operación no se pueda realizar si el vector está lleno. (dimL < dimF).
  + Pasos:
    1. Verificar si hay espacio (dimL < dimF).
    2. Agregar al final de los elementos ya existentes el elemento nuevo.
    3. Incrementar la cantidad de elementos actuales

(dimL + 1).

* + Forma:

Procedure agregar (var a:numeros; var dL:integer; var pude:boolean; num:integer);

Begin

Pude := false;

If ((dL + 1) <= dF) then begin

pude := true;

dL := dL + 1;

a[dL] := num;

end;

end;

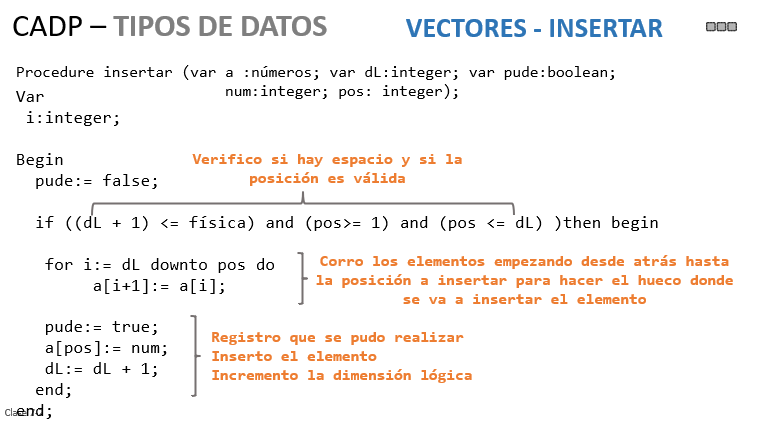
* **Insertar:** agregar en el vector un elemento en una posición determinada. Puede pasar que esta operación no se pueda realizar si el vector esta lleno o si la posición no es válida.
  + La posición en donde queremos insertar el elemento nuevo tiene que ser <= a la dimL y >= que la primera posición del arreglo.
  + Pasos:
    1. Verificar si hay espacio (dimL < dimF).
    2. Verificar que la posición sea válida (dimL >= pos >= primera posición del arreglo).

Solo si se cumplen esas dos

* + 1. Hacer lugar para poder insertar el elemento.
    2. Incrementar la cantidad de elementos actuales

(dimL + 1).

* + Forma:



* **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.
  + **Pasos:**
    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.

**(eliminar)**

* + **Forma:**

Procedure eliminar (var a:números; var dL:integer; var pude:boolean; pos:integer);

Var

i : integer;

begin

pude := false;

if ((pos >= 1) and (pos <= dL)) then begin

for i := pos to (dL - 1) do

a[i] := a[i + 1];

pude := true;

dL := dL – 1;

**Búsqueda**

Recorrer el vector buscando un valor que puede o no estar en el vector.

* **Búsqueda en un vector desordenado:** se recorre sin ningún criterio. Tengo que avanzar por los elementos hasta encontrar el que busco o hasta que se termine de recorrer todos los elementos del vector.
  + Hay que determinar por cual de las dos razones se detuvo.
  + Ejemplo:

function buscar (a:números; dL:integer; valor:integer): boolean;

Var

pos : integer;

esta : boolean;

Begin

esta := false;

pos := 1;

while ((pos <= dL) and (not esta)) do

Begin

If (a[pos] = valor) the esta := true

else

pos := pos + 1;

End;

buscar := esta;

End.

* **Búsqueda en un vector ordenado:** puedo implementar dos tipos de búsqueda:
  + **Búsqueda mejorada:** no recorre en el arreglo las posiciones que no le sirven, o sea, las posiciones en donde el valor es mayor al que estoy buscando.
    1. Inicializar la búsqueda desde la posición 1 (pos).
    2. Mientras ((el elemento buscado sea menor al valor en el arreglo[pos]) y (no se termine el arreglo)).

2.1. Avanzo una posición

* + 1. Determino por qué condición se ha terminado el while y devuelvo el resultado.

Function existe (a:números; dL:integer; valor:integer): boolean;

Var

Pos : integer;

Begin

Pos := 1;

While ((pos <= dL) and (a[pos] < num)) do

Pos := pos + 1;

If ((pos <= dL) and (a[pos] = num)) then buscar := true

Else buscar := false;

End.

* + **Búsqueda binaria/ dicotómica:**
    1. Se calcula la posición media del vector (teniendo en cuenta la cantidad de elementos). Esto lo hago con dos variables auxiliares (inf y sup) que inicialmente las voy a poner en 1 y en la dimensión lógica y el medio se calcula como la suma, dividido 2 (1(pri) + dL(ult) DIV 2) **.**
    2. Luego de calcular el medio me fijo si no es igual al valor que estaba buscando; si no es igual a nuestro valor, veo con qué mitad me conviene quedarme. En ese caso voy a cambiar los valores de inf y sup dependiendo de lo que me convenga.
    3. Repito estos pasos hasta que encuentre el elemento o cuando el límite inferior sea más grande que el límite superior.

Función existe (v:números; dL:integer; x:integer): boolean;

Var

Pri, ult, medio : integer;

Begin

Ok := false;

Pri := 1; ult := dL; medio := (pri + ult) DIV 2;

While (pri < ult) and (x <> v[medio]) do begin

If (x < v[medio]) then

Ult := medio – 1;

Else pri := medio + 1;

Medio := (pri + ult) div 2;

End;

If (pri <= ult) and (x = v[medio]) then ok := true;

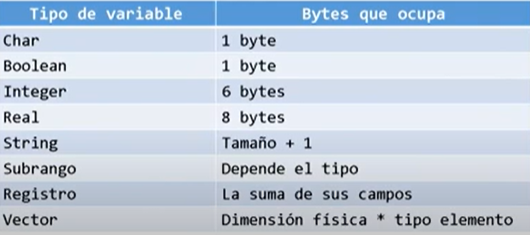
Existe := ok;

End.

**Alocación de memoria**

Nuestra memoria se divide en 3 partes:

* **Memoria estática:** hasta ahora, cualquier variable que se declare en un programa es alojada en la memoria estática de la CPU.
  + Las variables declaradas permanecen en la memoria estática durante toda la ejecución del programa, más allá de que sigan siendo utilizadas o no (al permanecer en la memoria siguen ocupando memoria).

****

* + En el momento de la declaración ya se sabe cuanta memoria es la que van a utilizar, y esa memoria no va a cambiar durante la ejecución del programa.
  + Pero como toda memoria, esta tiene una capacidad y esta a veces se ve superada, por eso existe la:
* **Memoria hip/ dinámica:** en donde uno por un tiempo puede reservar un poco de memoria y cuando no la utiliza más dentro del programa la libera. (el tamaño de esta memoria hip es mucho más pequeño que el de la memoria estática).

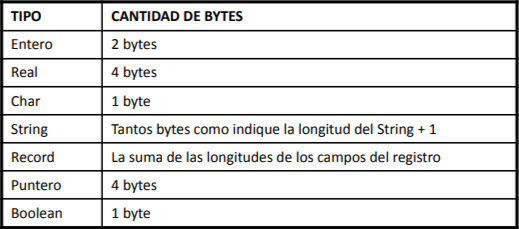
Para solucionar los problemas de espacio mencionados anteriormente, los lenguajes permiten la utilización de tipos de datos que permiten reservar y liberar memoria dinámica durante la ejecución del programa a medida que el programador lo requiera.

Me permite reservar memoria dinámica, utilizarla, asignarle valores, hacer operaciones y cuando no se necesita más se libera ese espacio. A este tipo de datos lo llamamos:

**Puntero**

Es una variable de tipo simple que se aloja en la memoria estática, pero como contenido de esa variable de tipo puntero tiene una dirección y esa dirección pertenece a la memoria dinámica, y en esa dirección de la memoria dinámica es donde realmente se almacena el valor que quiero guardar.

* Siempre ocupa 4 bytes, porque guarda una dirección.
* Cuando uno quiere darle valor a una variable de tipo puntero, primero tiene que pedir que me reserve una dirección de memoria dinámica y recién ahí le puede asignar un valor de algún tipo y cuando ya se que dentro del programa no lo voy a utilizar más voy a tener otra operación que me permite liberar esa memoria dinámica. En la memoria estática donde se declaro esa variable de tipo puntero sigue teniendo ocupado ese espacio durante toda la ejecución del programa (4 bytes).



* **¿Cómo se declara?** Con el símbolo **^** . (alt gr + { , dejo de apretar y { de nuevo)

Type

puntero = ^ tipo de datos;

Var

p : puntero;

(Creación, destrucción, asignación y comparación de variables puntero)

* **Creación:** utilizar una operación que nos permita reservar un pedacito de memoria dinámica para poder guardar efectivamente el valor:
  + **New (variable tipo puntero)**

Program uno;

Type

Puntero = ^integer;

Var

Num : integer;

P : puntero;

Begin

New (p); // el lenguaje elige un pedazo de memoria dinámica disp.

… //no puedo asignarle directamente la dirección con :=

End.

* **Eliminación:** implica liberar la memoria dinámica que contenia la variable de tipo puntero.
  + **Dispose (variable tipo puntero)**

Program uno;

Type

Puntero = ^integer;

Var

Num : integer;

P : puntero;

Begin

New (p);

Dispose (p);

End.

* **Liberación**: Implica cortar el enlace que existe con la memoria dinámica. La misma queda ocupada pero ya no se puede acceder.
  + **Nil**

Program uno;

Type

Puntero = ^integer;

Var

Num: integer;

P: puntero;

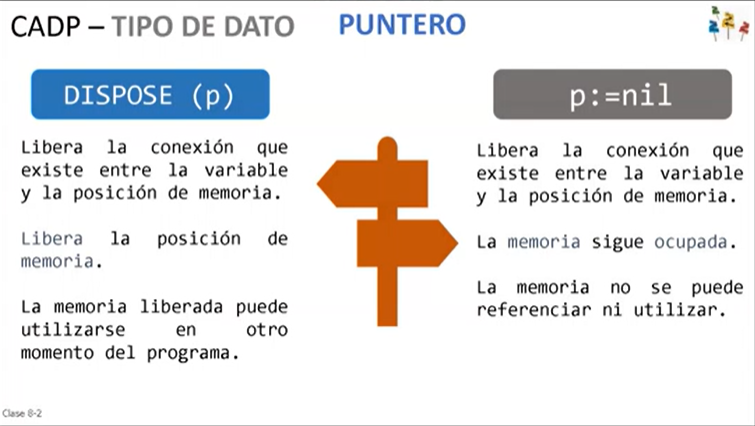
Begin

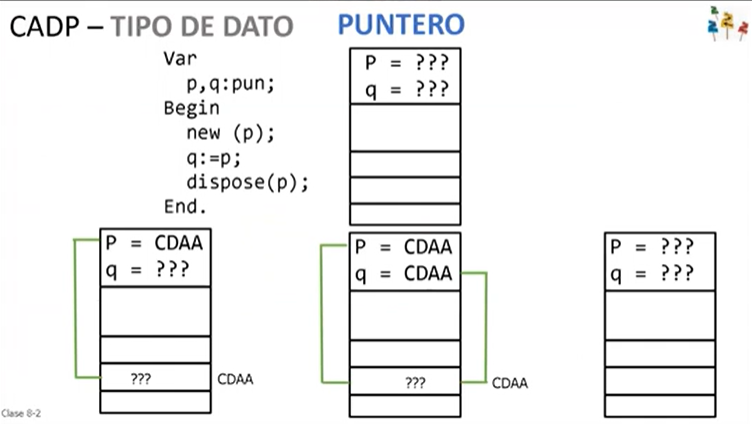
New(p);

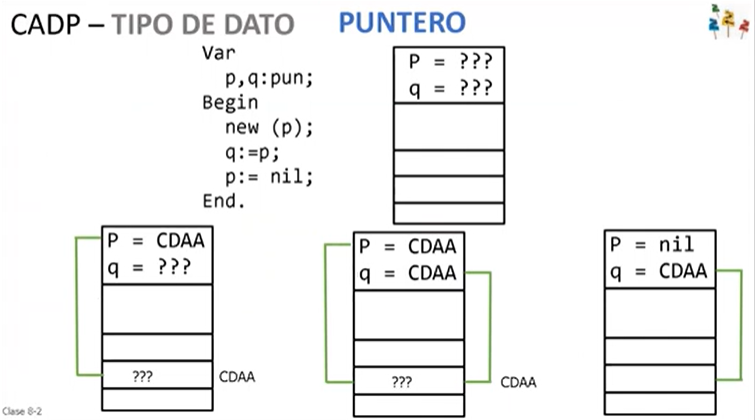
P := nil;

End.

**Diferencia entre dispose (p) y p := nil**

****





* **Asignación entre punteros:** Implica asignar la dirección de un puntero a otra variable del mismo tipo
  + **:=**

Program uno;

Type

Puntero = ^integer;

Var

Q: puntero;

P:puntero;

Begin

New(P);

Q := P; (funciona como los parámetros por referencia)

End;

* **Contenido de un puntero:** Implica poder acceder al contenido que contiene la dirección de memoria que tiene una variable de tipo puntero.
  + **^**

Program uno;

Type

Puntero = ^integer;

Var

P: puntero;

Begin

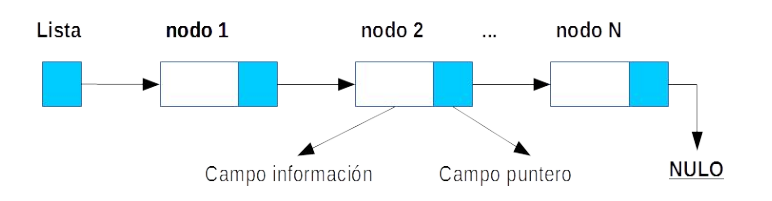
New(p);

P^ := 8;

End;

**Listas**

* Es una **colección de nodos**.
* Cada nodo contiene un registro formado por:
  + Un elemento (valor que se quiere almacenar en la lista).
  + Una dirección de memoria dinámica donde se encuentra el siguiente elemento de la lista (el siguiente nodo de la lista).
* Los elementos que la componen no ocupan posiciones secuenciales o contiguas de memoria, es decir, pueden aparecer dispersos en la memoria, pero mantienen un orden lógico interno.



* + Está compuesta por **punteros (**los punteros son de tipo simple pero como las listas son un conjunto de estos punteros, las listas son de tipo compuesto**).**
  + Son estructuras donde se almacenan datos **sin saber la cantidad de los mismos**.
  + Son estructuras **dinámicas**: se reserva/ libera memoria para datos según sea conveniente.

Siempre debo guardar el puntero inicial de la lista, es decir, el apuntador al primer nodo, para luego poder recorrerla, ya que a partir del primer elemento se puede acceder al siguiente y así sucesivamente.

* + 1. En memoria estática se declara una variable tipo PUNTERO (ya que son las únicas que pueden almacenar direcciones). La dirección almacenada en esa variable representa la dirección donde comienza la lista. Inicialmente ese puntero no contiene ninguna dirección.
    2. A medida que se quiere agregar elementos a la lista (nodo), se reserva una dirección de memoria dinámica y se carga el valor que se quiere guardar. El último nodo de la lista indica que la dirección que le sigue es nil.
* **Características**:
  + **Homogénea**: los elementos son del mismo tipo.
  + **Dinámica**: el tamaño puede cambiar durante la ejecución del programa.
  + **Lineal:** cada nodo de la lista tiene un nodo que lo sigue (salvo el último) y uno que lo antecede (salvo el primero).
  + **Secuencial:** el acceso a cada elemento es de manera secuencial, es decir, para acceder al elemento 5 (por ejemplo) debo pasar por los 4 anteriores.
  + Cada vez que se necesite agregar un nodo se deberá reservar memoria dinámica (new) y cuando se quiere eliminar un nodo se debe liberar la memoria dinámica (dispose).
* **Forma**:

Program ejemplo;

Type

nombreTipo = ^nodoTipo; //elegir un nombre (normalmente es lista) = puntero al registro

nodoTipo = record

elemento : tipoElemento;

punteroSig: nombreTipo;

end;

var

Pri: nombreTipo; // en esta variable se guarda la dirección donde comienza esta lista.

* **Operaciones**:
  + **Crear** lista agregando elementos al inicio.
  + **Crear** lista agregando los elementos al final.
  + **Insertar** un nuevo elemento en una lista ordenada.
  + **Recorrer** una lista.
  + **Agregar** nodos al final de la lista.
  + **Acceder** al k-ésimo elemento de la lista.
  + **Eliminar** un elemento de la lista.
  + **Combinar** dos listas ordenadas formando una sola ordenada (**Merge** de Listas)

--------------------------------------------------------------------------------------------------------------------------

* **Crear una Lista:** Implica marcar que la lista no tiene una dirección inicial de comienzo. Es como cuando inicializamos una variable y le ponemos v := 0; .

Program uno;

Type

listaE = ^datosEnteros;

datosEnteros = recors

elem: integer;

sig: listaE;

end;

var

pri: listaE; (memoria estática reservada)

begin

**pri := nil;**

end.

* **Recorrer una lista:** Implica posicionarse al comienzo de la lista y a partir de allí ir “pasando” por cada elemento de la misma hasta llegar al final (con valor nil).

Program uno;

Type

listaE = ^datosEnteros;

datosEnteros = record

elem: integer;

sig: listaE;

end;

var

pri: listaE;

begin

crear (pri);

cargarLista (pri);

recorrerLista (pri);

end.

* + Inicializo una variable aux con la dirección del puntero inicial de la lista.
  + Mientras (no sea el final de la lista) proceso el elemento (ej: imprimo, sumo, modifico) y avanzo al siguiente elemento aux.

Procedure recorrerLista (pI: listaE); recibo el puntero inicial (pI)

Var

Aux : listaE; tiene que ser del mismo tipo para poder asignarse

Begin

Aux := pI;

While (aux <> nil) do begin

Writeln (aux^.elem);

Aux := aux^.sig;

End;

End;

(como pI esta pasada por valor, ósea que vuelve al programa con el mismo valor que se fue, no se modifica, lo que significa que la variable aux podria no utitlizarse, intercambiando aux por pI).

* **Agregar adelante:** 
  + 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 de **nuevo elemento** es el puntero inicial. Actualizo el puntero inicial de la lista con la dirección del **nuevo elemento**.

Program uno;

Type

listaE = ^datosEnteros;

datosEnteros = record

elem: integer;

sig: listaE;

end;

agregarAdelante (var pI: listaE; num: integer);

var

nuevo: listaE;

begin

new (nuevo);

nuevo^.elem := num;

nuevo^.sig := nil;

if (pI = nil) then pI := nuevo

else begin

nuevo^.sig := pI;

pI := nuevo;

end;

end;

var

pri: listaE;

num: integer;

begin

crear(pri);

readln(num);

agregarAdelante (pri, num);

readln(num);

agregarAdelante (pri, num);

* **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**.

Program uno;

Type

listaE = ^ datosEnteros;

datosEnteros = record

elem : integer;

sig : listaE;

end;

procedure agregarAlFinal (var pI: listaE; num: integer);

var

nuevo, aux: listaE;

begin

new (nuevo);

nuevo^.elem := num;

nuevo^.sig := nil;

if (pI = nil) then pI := nuevo

else begin

aux := pI;

while (aux^.sig <> nil) do

aux := aux^.sig;

aux^.sig := nuevo;

end;

end;

var

pri: listaE;

num: integer;

begin

crear(pri);

readln(num);

agregarAlFinal (pri, num);

readln(num);

agregarAlFinal (pri, num);

end.

(opción 2 para cuando tengo que agregar al final de una lista con muchos elementos)

Procedure agregarAlFinal (var pI, pU: listaE; num: integer);

Var

Nuevo: listaE;

Begin

New(nuevo);

Nuevo^.elem := num;

Nuevo^.sig := nil;

If (pI = nil) then begin

pI := nuevo;

pU := nuevo;

end;

else begin (no es nil, asi que ya tiene elementos cargados)

pU^.sig := nuevo;

pU := nuevo; (actualizo el sig del ult nodo y el ult nodo)

end;

end;

var

pri, ult: listaE;

num: integer;

begin

crear(pri, ult);

readln(num);

agregarAlFinal (pri, ult, num);

readln(num);

agregarAlFinal (pri, ult, num);

end.

* **Buscar un elemento en una lista:** significa recorrer la lista desde el primer nodo buscando un valor que puede o no estar. Se debe tener en cuenta si la lista está ordenada o no.
  + **Desordenada:** se debe recorrer toda la lista (en el peor de los casos), y detener la búsqueda en el momento que se encuentra el dato buscado o en el que la lista terminó.
    1. Comienzo a recorrer la lista desde el nodo inicial.
    2. Mientras ((no sea el final de la lista) y (no encuentre el elemento)) do
       - Si (es el elemento buscado) entonces

Detengo la búsqueda

* + - * Sino

Avanzo al siguiente elemento

Program uno;

Type

listaE = ^datosEnteros;

datosEnteros = record

elem: integer;

sig: listaE;

end;

function buscar (pI: listaE; valor: integer): boolean;

var

aux: listaE;

encontré: boolean;

begin

encontré := false;

aux := pI;

while ((aux <> nil) and (encontré = false)) do begin

if (aux^.elem = valor) then

encontré := true

else

aux := aux^.sig;

end;

buscar := encontré;

end;

var

pri: listaE;

num: integer;

begin

crear(pri);

cargar(pri); //se dispone

readln(num);

if (buscar(pri, num)) then writeln (‘el elemento existe’);

end.

* + **Ordenada:** se debe recorrer la lista teniendo en cuenta el orden. La búsqueda se detiene cuando se termina la lista o el elemento buscado es mayor al elemento actual.

Function buscar (pI: listaE; valor: integer): boolean;

Var

Aux: listaE;

Encontré: boolean;

Begin

Encontré := false;

Aux := pI;

While ((aux <> nil) and (aux^.elem < valor)) do

Aux := aux^.sig;

If (aux <> nil) and (aux^.elem = valor) then

Encontré := true;

Buscar := encontré;

End;

* **Insertar ordenado:** implica agregar un nuevo nodo a una lista ordenada por algún criterio de manera que la lista siga quedando ordenada.
  + **Existen 4 casos:**
    1. Que la lista este vacía.
    2. Que el elemento vaya al comienzo de la lista (es menor al 1er nodo de la lista).
    3. Que el elemento vaya al “medio” de la lista (es menor al último nodo de la lista).
    4. Que el elemento vaya al final de la lista (es mayor al último nodo de la lista).
  + **Caso 1: Lista vacía**
    1. Generar un nuevo nodo (NUEVO).
    2. Asignar a la dirección del puntero inicial (PI) la del nuevo nodo (NUEVO).
  + **Caso 2: lista no vacía, va al principio**
    1. Generar un nuevo nodo (NUEVO).
    2. Preparar punteros para el recorrido.
    3. Asignar a la dirección del puntero siguiente del nuevo la dirección del nodo inicial (PI).
    4. Actualizar con la dirección del nuevo nodo la dirección del puntero inicial (PI). (observar que actual quedó = a pri)
  + **Caso 3: lista no vacía, va en el “medio”**
    1. Generar un nuevo nodo (NUEVO).
    2. Preparo los punteros para el recorrido.
    3. Recorro hasta encontrar la posición.
    4. Reasigno punteros, el sig de anterior es NUEVO y el sig de NUEVO es actual. (observar que actual quedó distinto a nil).
  + **Caso 4: lista no vacía, va al final**
    1. Generar un nuevo nodo (NUEVO). Preparo los punteros para el recorrido.
    2. Recorro hasta encontrar la posición.
    3. Reasigno punteros, el sig al anterior es NUEVO y el sig al NUEVO es nil. (obserar que actual quedó = nil).
* **Forma:**

**Si (la lista esta vacía) (pri = nil)**

Actualizo la dirección del nodo inicial (pri)

**Sino**

Preparo los punteros para el recorrido (anterior, actual)

Recorro hasta encontrar la posición.

**Si (va al principio) (actual = pri)**

Asigno como sig del nodo nuevo al nodo inicial

Actualizo la dirección del nodo inicial (pri)  
 **Si (va en el medio) (actual <> nil)**

La dirección del sig del puntero anterior es la dirección del nodo nuevo

La dirección del sig del nodo nuevo es la dirección del actual

**Sino (al final) (actual = nil)**

La dirección del sig del puntero anterior es la dirección del nodo nuevo

La dirección del sig del nodo nuevo es la dirección nil

Program uno;

Type

listaE = ^datosEnteros;

datosEnteros = record

elem: integer;

sig: listaE;

end;

procedure insertar (var pri: listaE; valor: integer);

var

actual, anterior, nuevo: listaE;

begin

new (nuevo);

nuevo^.elem := valor;

nuevo^.sig := nil;

if (pri = nil) then **(caso 1)**

pri := nuevo;

else begin **(avanzo para buscar el lugar)**

actual := pri;

ant := pri;

while (actual <> nil) and (actual^.elem < nuevo^.elem) do begin

anterior := actual;

actual := actual^.sig;

end;

end;

if (actual = pri) then begin **(caso 2)**

nuevo^.sig := pri;

pri := nuevo;

end

else begin **(caso 3 y 4 pq es lo mismo ya que en el 4 actual = nil)**

anterior^.sig := nuevo;

nuevo^.sig := actual;

end;

var

pri: listaE;

num: integer;

begin

crear (pri);

cargar (pri);

read(num);

insertar(pri, num);

end.

* **Eliminar (dispose):** 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.
  + Si la lista esta **desordenada** seguramente la búsqueda se realizara hasta encontrar el elemento o hasta que se termine la lista.
  + Si la lista esta **ordenada** seguramente la búsqueda se realizara hasta que se termina la lista o no se encuentre un elemento mayor al buscado.
  + Existen 3 casos, que el elemento a eliminar…:

(lista desordenada)

* + 1. No se encuentre en la lista
       - Recorrí toda la lista y el elemento a eliminar no se encuentra. Observar que actual quedó en nil
    2. Sea el primero de la lista
       - ¿Cómo me doy cuenta si es el primero? Pq actual y pri están apuntando a la misma dirección.
       - Empiezo a recorrer la lista

Mientras (no encuentro el elemento a borrar) y (no se termina la lista)

El puntero anterior toma la dirección del puntero actual

Avanzo el puntero actual

Como (el elemento está) y (es el primer elemento)

Actualizo el puntero inicial de la lista

Elimino la dirección del puntero actual

Observar que actual había quedado igual a pri

* + 1. No sea el primero en la lista
       - Empiezo a recorrer la lista

Mientras (no encuentro el elemento a borrar) y (no se termine la lista)

El puntero anterior toma la dirección del puntero actual

Avanzo el puntero actual

Como (el elemento está) y (NO es el primer elemento)

Actualizo el siguiente del puntero anterior con el siguiente del actual

Elimino la dirección del puntero actual

Observar que actual había quedado <> nil y de pri

* + **Forma**:

Program uno;

Type

listaE = ^datosEnteros

datosEnteros = record

elem: integer;

sig: lista;

end;

procedure eliminar (var pri: listaE; valor: integer);

var

actual, ant: listaE;

begin

actual := pri;

while (actual <> nil) and (actual^.elem <> valor) do begin

ant := actual;

actual := actual^.sig;

end;

if (actual <> nil) then

if(actual = pri) then

pri := pri^.sig;

else

ant^.sig := actual^.sig;

dispose(actual);

end;

var

pri: listaE;

num: integer;

begin

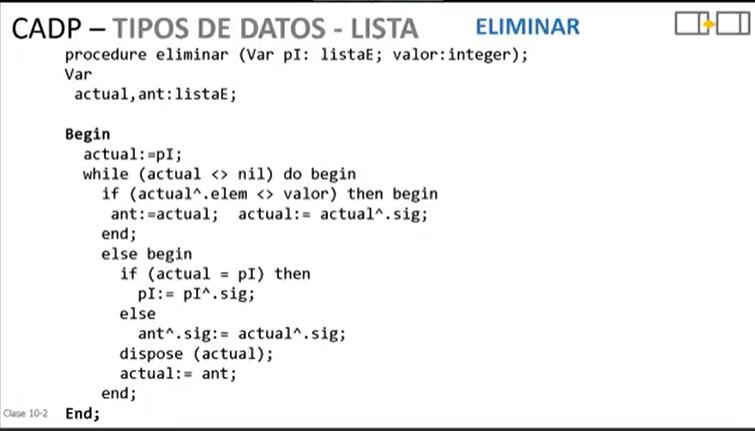
crear(pri);

cargar(pri); //se dispone

read(num);

eliminar(pri, num);

¿Qué modifico si el elemento puede repetirse?



Fijarse que caso más faltaba agregar