**AL FINAL DEL ARCHIVO SE ENCUENTRA EL PARCIAL RESUELTO**

Informática: Es la ciencia que estudia el análisis y resolución de problemas utilizando computadoras.

Memoria estática: La memoria estática es aquella donde se almacenan las variables de las cuales su tamaño no será alterado. *Las variables solo cambian su contenido, no su tamaño.*

Memoria dinámica: La memoria dinámica es aquella donde las variables a almacenar no tienen un tamaño definido, y podrán aumentar o disminuir su tamaño tanto les sea permitido por la misma.

Tipo de dato: El tipo de dato es un atributo, tanto para el ordenador, como para el programador, para saber qué valores manejará una variable declarada en memoria.

Existen varios tipos de datos, y estos se dividen en simples y compuestos.

* SIMPLES: Integer, real, boolean, char y puntero.
* COMPUESTO: String y Estructuras (registró, lista y vector).
* DEFINIDOS POR EL PROGRAMADOR.

*El tipo de dato real no admite div y mod.*

Puntero: Un puntero es un tipo de dato simple, que trabaja con la memoria dinámica, contiene la dirección de memoria, en memoria dinámica, donde se encuentra almacenado el dato real, este dato puede ser un entero, un real, un string, y los demás tipos de datos, inclusive otro puntero.

Tipo de dato definido por el programador: Un tipo de dato definido por el programador es un nuevo tipo de dato que define el programador, para su posterior uso en el programa. Este se puede obtener a partir de tipos de datos simples o compuestos. Sintaxis en pascal:

Type

Identificador = Tipo;

Para su posterior uso, se debe asignar este nuevo tipo en la declaración de una variable.

Var

variable : Identificador;

Estructuras de control: If y else, case, while, for y repeat until.

* If y Else: sentencia que modifica, o no, una sola vez la ejecución del programa.

Estructura:

If (condición) then

{pasos a realizar}

else

{pasos a realizar en caso contrario}

Else es dependiente de If, no puede estar esta instrucción si anteriormente no hubo un if.

* Case: Estructura de control que simula ser un if anidado, y analizará los diferentes casos dados, de esta forma realizará una acción dada al encontrar un caso verdadero.

Estructura:

case (condicion) of

{instrucción 1}

{instrucción 2}

.

.

.

{instrucción n}

* while: estructura de control que se ejecuta en base a una condición, cero, una o n cantidad de veces realizando instrucciones. La condición debe ser finita, ya que podría producir un bucle infinito. La condición deberá ser verdadera para su correcto uso.

Estructura:

while (condición) do

{paso a realizar}

* For: estructura de control que se ejecuta en base a una condición, una o n cantidad de veces. La condición de esta estructura tiende a ser finita, ya que se le da un valor de comienzo y uno de final.

Estructura:

for (valor de comienzo) to (valor final) do

{pasos a realizar}

* repeat until: estructura de control que se ejecuta antes de verificar la condición, una o más veces. El código se ejecuta tantas veces como la condición sea falsa. Se utiliza mayormente cuando quiere procesar el dato que cambia la condición a verdadera. Cabe recalcar, que la condición debe ser finita.

Estructura:

repeat

{pasos a realizar}

until (condición)

* \*

Diferencias entre if y case:

* If y case son lo mismo, simplemente, el case compara un dato con otros tantos, y en caso de cumplirse uno de estos, realizará una acción. En cambio, el if se utiliza para obtener una respuesta en base a un verdadero o falso, no obstante, se pueden anidar varios if y copiar la estructura de un case, pero esto hace menos legible al código.
* En el case, al aceptar valores de tipo ordinal, se pueden especificar rangos dentro de las condiciones, en cambio, el if no tiene esta opción, y se debe realizar una condición que tape todas las posibilidades.

ejemplos de estructuras de control en pascal:

*while*:

var

i:integer;

begin

i:= 0;

while (i < 10) do

begin

i+= 1;

write(i);

end;

end.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*for*:

var

i:integer;

begin

for i:= 1 to 10 do

write(i);

end.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*repeat until*:

var

i:integer;

begin

i:=0;

repeat

i+= 1;

write(i);

until (i=10);

end.

Modularización:

* La modularización sirve para separar el código, de forma lógica, en diferentes partes, encapsulando código relacionado.
* Ayuda a encontrar errores de forma más eficiente.
* Hace más legible el código.
* La reutilización es primordial cuando se habla de modularizar, evitamos programar el doble y reutilizamos el módulo en otra sección del código.

Procedure y Function:

En pascal existe el proceso y la función, que son dos formas de modularizar en un programa, estas están definidas por la siguiente sintaxis:

Procedure nombreProceso (parámetros de E o E/S);

**Características de procedure:**

* Podrá recibir datos, llamados parámetros por valor, y, eventualmente, podrá devolver datos, cuando el parámetro está definido por referencia.
* Puede recibir, pero no necesariamente retornar un valor.
* Puede no recibir ningún parámetro y funcionar igualmente.

Function nombreFuncion (parámetros de entrada): tipoDeFuncion;

**Características de function:**

* Podrá devolver un único valor, de tipo simple, definido en el encabezado de la función, luego de los parámetros.
* Podrá recibir datos, pero no retornar los mismos.
* Sus parámetros no pueden ser por referencia.
* Para retornar un valor, se debe asignar un valor al nombre de la función, que es similar a una variable, guarda su valor en esa posición de memoria, de lo contrario, retornará basura.

**Caracteristicas de procedure y function:**

* Las variables definidas dentro de las mismas sólo serán exclusivas de estas, es decir, su alcance es local.
* Se podrán llamar procesos y funciones dentro de las mismas.
* Cuando se pasa un parámetro por valor, se realiza una copia de este dato, y podrá ser utilizada dentro del módulo sin alterar su valor original.
* Los parámetros, al igual que las variables declaradas dentro del módulo, son locales al módulo.
* Ambas pueden manipular variables globales.
* Se pueden declarar types dentro de las mismas.

**Diferencias:**

* Procedure puede recibir parámetros de entrada, y también existe la opción de ser devueltos, function no tiene esta posibilidad, sólo podrá retornar un único valor.
* Los tipos de datos devueltos por estos son diferentes en cada uno, mientras el procedure puede recibir y devolver cualquier tipo de dato, function solo puede devolver datos de tipo simple.
* El tipo de dato que devolverá la función, al ser único, debe ser definido en el encabezado de la misma, y solo podrá devolver ese único tipo de dato, mientras que el proceso podrá devolver todos los tipos de datos que estén por referencia en la sección de parámetros.

Estructuras de datos:

Permite al programador definir un tipo al que se asocian diferentes datos que tienen valores lógicamente relacionados y asociados bajo un nombre único.

Propiedades de las estructuras de datos:

Elementos: Homogenea y Heterogenea:

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

Acceso: Secuencial y Directo.

* Secuencial: Para ingresar a un valor, se debe tener un orden en particular.
* Directo: Para ingresar a un valor, solo se debe saber dónde está posicionado, o cual es su identificador.

Linealidad: Lineal y no Lineal:

* Lineal: Está formada por ninguno, uno o varios elementos que guardan una relación de adyacencia ordenada donde a cada elemento le sigue uno y le precede uno, solamente
* No lineal: Para un elemento dado pueden existir 0, 1 ó más elementos que le suceden y 0, 1 ó más elementos que le preceden.

Tamaño: Dinamico y estatico:

* Dinámico: El tamaño puede aumentar durante la ejecución del programa.
* Estático: El tamaño no puede variar durante la ejecución del programa.

**Registros:** Un registro es una estructura de datos que contiene elementos en su interior, comúnmente llamados campos, donde cada uno tiene un identificador y un tipo de dato. Estos datos siempre están relacionados, ya que provienen de los atributos de algo en particular. El registro es directo, ya que para ingresar a cada posición se debe saber el identificador del campo, y heterogénea, ya que sus elementos pueden ser de diferentes tipos, como también, pueden ser todos de un mismo. Es estático.

**Vector:** Un vector es una estructura de datos que contiene casillas, de las cuales, cada una contiene un dato, y cada casilla tiene un número que la identifica dentro del vector que va desde 1 hasta el ingresado por el programador, conocida como dimensión física. Es directo, porque se puede ingresar a cualquier casilla sin necesidad de haber recorrido las anteriores. Es homogénea, ya que, el vector solo admite en sus casillas elementos del mismo tipo. Es de tamaño estático, ya que su tamaño será el declarado, y una vez compilado no podrá aumentarse.

**Listas:** Una lista, es una estructura de datos dinámica, que trabaja con punteros y nodos. Un puntero y nodo inicial comienza la lista, y contendrá, en alguno de sus campos, la siguiente posición de memoria donde se encuentre el próximo nodo para continuar la lista. Una lista es de acceso secuencial, ya que para llegar al último nodo, se deberá recorrer cada uno, hasta encontrar uno que, en el campo del siguiente nodo contenga “nil”. Es de tamaño dinámico, ya que su tamaño irá aumentando o disminuyendo a medida que se ejecute el programa, utilizando las instrucciones correspondientes (new o dispose).

Algoritmos de búsqueda: Un algoritmo de búsqueda es un conjunto de instrucciones que están diseñadas para localizar un elemento con ciertas propiedades dentro de una estructura de datos.

**Búsqueda secuencial:** se basa en buscar un elemento dentro de un vector, de principio a fin, o hasta ser encontrado. En caso de que este desordenado, este será el más “eficiente”, en caso de que este ordenado, será ineficiente.

Código en GitHub:

<https://github.com/NicaOB/Algoritmos-de-busqueda/blob/main/busquedaSecuencial.pas>

*A partir del siguiente algoritmo, debe estar ordenado por medio del elemento que se desee buscar.*

**Búsqueda secuencial en vector ordenado:** Se procede a condicionar la búsqueda dentro del algoritmo, dejando un límite de búsqueda.

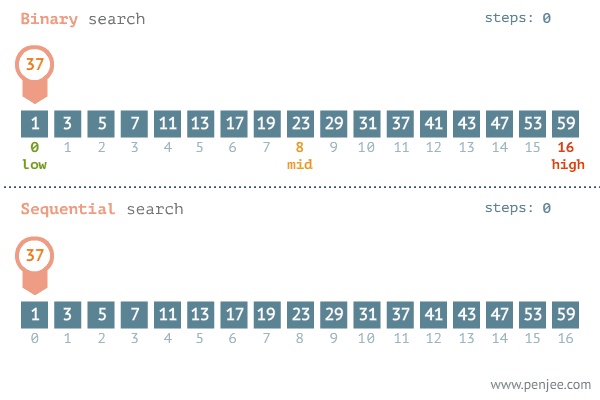
Código en GitHub:

<https://github.com/NicaOB/Algoritmos-de-busqueda/blob/main/BusquedaSecuencialOrdenada.pas>

**Búsqueda binaria o dicotómica:** Este algoritmo, a diferencia de los anteriores, comienza su búsqueda por la mitad del vector, comparando el elemento buscado con esta posición, y en caso de no ser el elemento a buscar, el algoritmo decidirá ir a la primera mitad o a la segunda mitad. Este proceso se repetirá n cantidad de veces, hasta que se encuentre o no el elemento.

Código en GitHub:

<https://github.com/NicaOB/Algoritmos-de-busqueda/blob/main/BusquedaDicotomica.pas>



Algoritmos de eliminación: Un algoritmo de eliminación se basa en buscar un elemento en una estructura de datos adecuada, y eliminar el dato.

**Eliminación en vector:** Para eliminar un elemento de un vector, se debe tener en consideración las siguientes cuestiones:

* La posición que se pida este dentro del rango de la dimensión lógica.
* Realizar el corrimiento desde la posición deseada, hasta el final.
* Decrementar en uno la dimensión lógica.

Código en GitHub:

<https://github.com/NicaOB/Algoritmos-de-eliminacion/blob/main/EliminacionVectosPos.pas>

Variaciones de algoritmo de eliminación en vector:

*Eliminación en vector por valor de elemento (sin ocurrencias):*

<https://github.com/NicaOB/Algoritmos-de-eliminacion/blob/main/EliminacionVectorBasico.pas>

*Eliminación en vector por valor de elemento (con ocurrencias):*

<https://github.com/NicaOB/Algoritmos-de-eliminacion/blob/main/eliminacionVector.pas>

**Eliminación en lista:** Eliminar un elemento de una lista, implica recorrer la lista, hasta encontrar el elemento deseado y eliminarlo con dispose.

Código en GitHub:

https://github.com/NicaOB/Algoritmos-de-eliminacion/blob/main/EliminacionLista.pas

Corrección y eficiencia: Cuando se habla de corrección y eficiencia se habla en base a un algoritmo. Y se divide en dos, corrección y eficiencia del mismo:

**Corrección:** proceso que da a conocer los posibles errores que pueda tener un algoritmo, y consta de 4 etapas, no necesariamente conectadas:

* Testing: Se diseña un plan de pruebas, para comprobar que el algoritmo realiza lo solicitado. Para realizar un plan de pruebas correcto se tiene en cuenta lo siguiente:

1. Saber que se debe testear y buscar datos de prueba para esos segmentos.
2. Determinar el resultado que se espera en cada prueba.
3. Poner atención en los casos límites.
4. Diseñar los casos de prueba e implementarlos al programa.

* Debugging: Se basa en escribir sentencias en diferentes partes del código, que estén relacionadas con el error a encontrar, para ver dónde se encuentra el problema, y solucionarlo posteriormente.

1. Error sintáctico: se muestra en la compilación del programa.
2. Error lógico: se muestra en la ejecución del programa.
3. Error del sistema: son pocos comunes.

* Walkthrough: Recorrer un programa frente a personas, de forma que, se pueda encontrar un error rápidamente, con ayuda de la audiencia.
* Verificación: Significa controlar que se cumplan las pre y post condiciones del mismo.

**Eficiencia:** Una vez el programa es correcto, sigue ver la eficiencia de los algoritmos, analizando el tiempo que tarda el algoritmo en ejecutarse, y la memoria que requiere.

Esta palabra se relaciona con:

* Datos de entrada (tamaño y cantidad).
* Calidad del código generado por el compilador.
* Naturaleza y rapidez en la ejecución de las instrucciones de máquina.
* El tiempo del algoritmo base.

La eficiencia va ligada al tiempo de ejecución, que en pocas palabras, es el tiempo que tarda el algoritmo en finalizar. Este se mide de dos maneras, de forma empírica o teórica:

* Empírica: se ejecuta en un ordenador y se analiza el tiempo de comienzo y final del algoritmo, pero se debe tener en cuenta que cada ordenador es diferente, por ende, el tiempo de ejecución está ligado a este factor, también, depende de los datos que se ingresen.
* Teórica: se busca realizar una medida en UT (unidad de tiempo) de forma general, en cada instrucción. De esta forma, se obtiene un valor más exacto en tiempo de ejecución, que será igual para todas los ordenadores, aunque sigue variando en sus componentes, este valor en UT es una medida que se respetará en todos por igual. Todas las instrucciones elementales miden una unidad de tiempo, es decir **1 UT**.

Ejemplo de analisis teorico:

Program uno;

var

aux,temp,x: integer;

Begin

aux:= 58; ⇐ Asignacion = 1 UT

aux:= aux \* 5; ⇐ Multiplicación y asignación = 2 UT

temp:= aux; ⇐ Asignación = 1 UT

read (x); ⇐ NO SE CALCULA EN CADP.

End.

UT total = 4UT. Ecuación = T (alg) = T(1) + T(2) + T(3)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Program dos;

var

aux,temp,x: integer;

Begin

aux:= 58; ⇐ Asignacion = 1 UT T1

aux:= aux \* 5; ⇐ Multiplicación y asignación = 2 UT T2

if (aux > 45) then ⇐ Ev. condición = 1 UT (todo lo que está acá, hasta el end es uno) T3

begin

temp:= aux – 5; ⇐ Resta y asignación = 2 UT

x:= temp + aux + 2; ⇐ Dos sumas (temp + aux y luego + 2) y asignación = 3 UT

end;

x:= x \* 10; ⇐ Multiplicación y asignación = 2 UT T4

end;

UT total = 11 UT. Ecuación = T(alg) = T1 + T2 + T3 + T4

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Program tres;

var

aux,temp,x: integer;

Begin

aux:= 58; ⇐ Asignación = 1 UT T1

aux:= aux \* 5; ⇐ Multiplicación y asignación = 2 UT T2

if (aux > 45) then ⇐ 1 UT \* cuerpo T3(if, else) T3(5 UT, 2 UT) T3 MAX (5 UT)

begin

temp:= aux – 5;

x:= temp + aux + 2;

end

else

x:= x \* 10; ⇐ Multiplicación y asignación = 2 UT end;

Siempre hay que analizar el peor caso, en este caso, lo que está dentro de la estructura if tiene mayor UT que, lo que hay en el else, por ende, en este caso sería lo siguiente:

UT final: 9UT. Ecuación = T(alg) = T1 + T2 + T3 (MAX)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Program cuatro;

var

aux,temp,x,i: integer;

Begin

aux:= 58; ⇐ Asignación = 1 UT T1

aux:= aux \* 5; ⇐ Mult. y asignación= 2 UT T2

for i:= 1 to 10 do ⇐ **\*\*\*\***

x:= x + 1;

temp:= aux \* 2;

end;

**\*\*\*\*** En este caso tenemos un for, y es un poco diferente, pero se calcula fácil.

Ecuación de For = 3(n) + 2 + n(cuerpo). n es la cantidad de veces que se repite el for, en este ejemplo es 10, reemplazamos, queda algo así For = 3 (10) + 2 + 10(cuerpo). Cuerpo es, literalmente, el cuerpo del for, lo que hay dentro, en este caso una suma y una asignación, y, una multiplicación y asignación, esto equivale a 2UT + 2UT, total 4UT, que va en la variable cuerpo.

Así se vería finalmente: For = 3 (10) + 2 + 10(4)

Resultado final **DEL FOR**: 30 + 2 + 40 = 72UT

Finalmente, el tiempo de ejecución de este código es de 72UT + 1UT + 2UT = 75UT

(1UT por aux:= 58; y 2UT por aux:= aux \* 5;)

Ecuación: T(alg) = T1 + T2 + T3

*Existe el caso donde el límite del For esté dado por un dato ingresado durante la ejecución del programa, en este caso se deja n como variable durante toda la ecuación.*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Program 5;

var

aux,temp,x,i: integer;

Begin

aux:= 5; ⇐ Asignación = 1 UT T1

while (aux > 0) do ⇐  **x**

x:= x + 1;

aux:= aux - 1;

end;

**x** En este caso tenemos un while, que es un poco más complejo que For, pero no difícil exactamente, ecuación de while: (n + 1) . (cond + conectivo) + n(cuerpo)

Comenzamos a reemplazar: n = 5, porque aux = 5, y la condición es, mientras 5 sea mayor que 0, hacer esto. Entonces, 5 + 1 + 5(cuerpo).

Cuerpo = suma y asignación 2UT + resta y asignación 2UT = 4UT.

Resultado final del while = 5 + 1 + 5(4) = 6 + 20 = 26 UT

Resultado final = 26 + 1 = 27 UT.

**Tabla de Unidades de Tiempo**

| Instrucciones | Unidades de tiempo |
| --- | --- |
| Asignación | 1 UT |
| Suma | 1 UT |
| Resta | 1 UT |
| IF | (Ev. de condición = 1 UT) + cuerpo ***\**** |
| FOR | 3(n) + 2 + n(cuerpo) ***\*\**** |
| WHILE | (n + 1) . (cond + conectivo) + n(cuerpo) ***\*\**** |

***\* Tener en cuenta, que cuando exista el else, se deberá calcular el peor de los casos.***

***\*\* n es cantidad de iteraciones. En caso de que no esté dado explícitamente, sea por un read(), se deja como una incógnita en la ecuación.***

Medición de memoria:

La medición de memoria de un programa consta de dos partes, la memoria estática, y la memoria dinámica. La memoria estática es aquella que su espacio va a ser al declarado al inicio, y compilado el programa, y esta no variará durante la ejecución del mismo. Mientras que la memoria dinámica no respeta un espacio exacto, sino que, crecerá tanto el programador lo pida, o el sistema operativo lo permita, aclarando que esta se verá modificada durante la ejecución del programa.

**Tabla de valores:**

| char | 1 byte |
| --- | --- |
| boolean | 1 byte |
| integer | 4 bytes |
| real | 8 bytes |
| string | tamaño + 1 byte |
| subrango | depende el tipo |
| registro | suma de los campos |
| arreglos | dim. Fisica \* tipo de dato |
| puntero | 4 bytes (en mem. estática) |

**Ejemplos en programas:**

Program tres;

Type

puntero = ^real;

puntero2 = ^char;

persona = record

nombre:string[20];

dni:integer;

end;

Var

p:puntero;

q:puntero2;

per:persona;

precio:real;

Begin

new (p);

new (q);

End.

puntero + puntero2 (no importa el tipo al que apunta) : 4 bytes + 4 bytes = 8 bytes;

persona es registro = campo 1: string[20](20 + 1 = 21 bytes), campo 2: integer(4 bytes): Total: 21 + 4 bytes = 25 bytes;

Memoria estática:

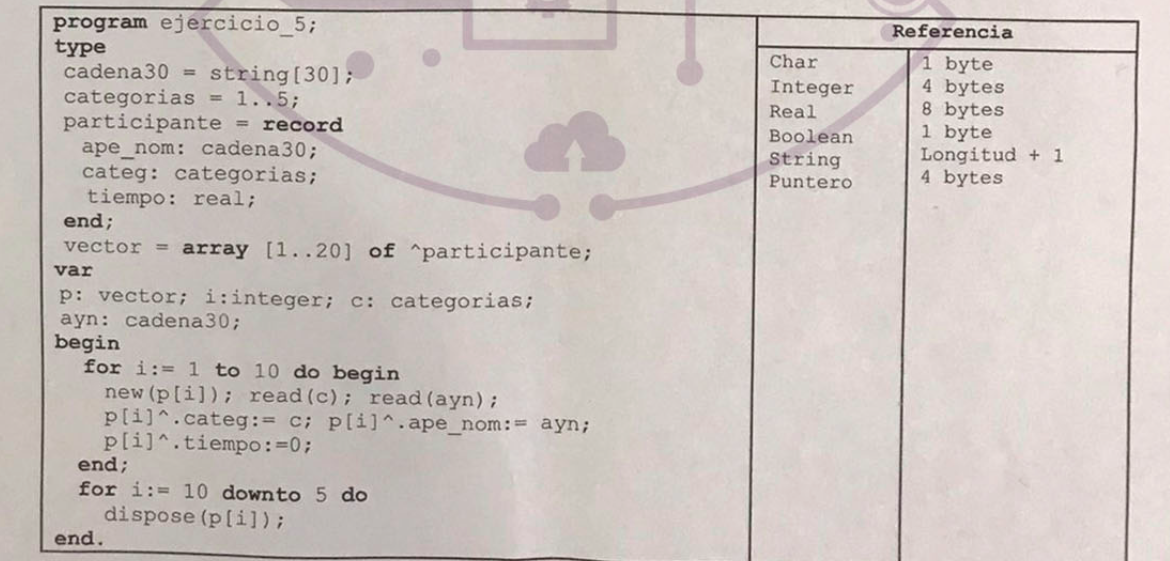
Teniendo en cuenta esto, hay que ver ahora en el programa principal, cual se usa y cual no.

p: puntero, por ende, son 4 bytes; q: puntero2, 4 bytes; per: persona, 25 bytes; precio: real, 8 bytes; en total, el programa ocupa en memoria estática 4 + 4 + 25 + 8 = 41 bytes.

Memoria dinámica:

Este programa le asigna a p y q una posición de memoria a ambas variables, por ende, en memoria dinámica se ocupa 8 bytes + 1 byte = 9 bytes en memoria dinámica.

Otro programa:



cadena30 = string[30] : 30 bytes + 1 byte = 31 bytes.

categorias = 1..5: 4 bytes (integer).

participante = record: (ape\_nom = 31 bytes) + (categ = 4 bytes) + (tiempo = 8 bytes) =

43 bytes;

vector = array de 20 tipo ^participante= 20 \* 4 = 80 bytes.

En el programa principal se declara la variable p, de tipo vector, por ende la memoria estática ocupada es de 80 bytes, se declara una variable i de tipo integer, + 4 bytes, total 84 bytes. C es declarada de tipo categorías, lo cual es un rango de 1 a 5, es decir, 4 bytes. total 88 bytes, ayn es de tipo cadena30, es decir, 31 bytes. total 88 + 31 = 119 bytes en memoria estática, esta no variará durante la ejecución del programa.

Dentro del programa principal, vemos un for de 1 a 10, dentro de esta estructura vemos la instrucción new(p[i]), lo que significa que, se está asignando a p[i], una posición de memoria, que será dinámica, podrá, eventualmente, ser liberada durante la ejecución del mismo programa, y esto sucederá 10 veces, por ende, podemos deducir que, si cada new(p[i]) ocupa 43 bytes en memoria dinámica, ya que hablamos del registro ‘participante’, el cual ocupa 43 bytes por sus campos, entonces, 10 \* 43 = 430 bytes en memoria dinámica, luego de este bucle, nos encontramos con un for que, en vez de crecer, decrece, de 10 a 5, por ende, son 5 repeticiones, y se realiza un dispose de p[i], en las posiciones del 10 al 5, esto significa que, parte de los 430 bytes serán liberados. Calculando seria lo siguiente, 430 / 2, ya que, antes contábamos con 10 posiciones del array con diferentes posiciones de memorias, y en esta estructura de control realizamos 5 dispose(p[i]), la mitad de las que teníamos, con esta conclusión podemos decir que ahora hay 215 bytes en memoria dinámica.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ejercicios final

1. ¿Cuáles son las principales diferencias entre las estructuras de control for, while () do, repeat until ()? Enuncie un problema sencillo para cada estructura de control que justifique su uso e implemente su solución.

Estas 3 son estructuras iterativas, es decir, se repetirán n cantidad de veces. El for se repetirá n cantidad de veces, dependiendo los valores ingresados de comienzo y final. While se repetirá 0, una o n cantidad de veces, siempre y cuando su condición sea verdadera, y repeat until se repetirá al menos una vez, siempre y cuando su condición sea falsa.

For se utilizará cuando se sepa la cantidad de veces que algo debe repetirse, mientras que while y repeat until se utiliza cuando no se sabe el valor de final, pero si la condición que lo detiene. Un inconveniente que presentan while y repeat until son los bucles infinitos, mientras que for, al tener un valor de comienzo y uno de final, nunca tendrá este inconveniente.

2. Defina el concepto de eficiencia de un programa. Describa detalladamente cómo se mide la memoria y el tiempo de ejecución de una solución.

La eficiencia de un programa, va ligada al tiempo de ejecución del programa y el espacio en memoria que ocupa el mismo. La memoria se mide en la compilación del programa cuando se trata de memoria estática, que son las variables locales y globales, mientras que la memoria dinámica, variará durante la ejecución del programa. Para medir la memoria estática, debemos ver las variables declaradas, ya que una vez compilado el programa, estas ocuparan ese espacio en memoria, y cambiar su contenido no varía su tamaño. Los diferentes tipos de datos ocupan distintos tamaños, yendo desde el menos significativo que es el char, hasta el arreglo que ocupará más tamaño según su dimensión física y su tipo. La memoria dinámica, en su lugar, ocupará el tamaño a lo largo de la ejecución del programa, y como este dato no es concreto, dependerá de lo que queramos lograr y así obtener un tamaño aproximado. Un puntero, por ejemplo, ocupa 4 bytes en memoria estática, pero una vez se le asigna una posición en memoria dinámica, depende exclusivamente de a qué tipo apunta este.

El tiempo de ejecución se mide de dos maneras, de manera empírica y de manera teórica, cuando hablamos de empírica, se debe a la ejecución de un programa en una computadora, y de esta forma obtener, en tiempo real, una respuesta de cuánto tiempo tardaría en ejecutarse, aunque no es recomendable, ya que esta prueba está ligada al ordenador en el que se ejecuta. Por otra parte, la teórica toma en cuenta las instrucciones dentro del programa, y las mide en unidad de tiempo, UT. En este caso, obtenemos una ecuación, y las UT que tendrá nuestro algoritmo o programa, consiguiendo finalmente cuantas unidades de tiempo necesita nuestro programa para ejecutarse correctamente, y este valor es universal para todas las computadoras, claro que seguirá dependiendo de los componentes de la misma, pero el valor oficial será en UT.

Defina el tipo de dato Arreglo y describa los pasos necesarios (pseudocódigo) para realizar operaciones de:

1. Buscar un elemento en un arreglo ordenado.
2. Insertar un elemento en una posición determinada del arreglo.

El tipo de dato arreglo, es una estructura de datos, que contiene casillas, donde en cada una se almacena un valor, del tipo de dato especificado. Este es directo, ya que no se necesita recorrer de principio a fin, solo se debe saber la posición del el. a buscar. Es homogénea, ya que, los datos almacenados en el vector tienen que ser del mismo tipo especificado. Y estática, ya que su tamaño es el declarado por el programador, y no variará durante la ejecución.

A) Búsqueda normal con condición de fin.

Declaro una variable booleana, donde guardo si encontre el valor.

Declaro una variable auxiliar para recorrer el vector de manera secuencial, y le asigno el valor 1.

mientras (var aux sea menor a dimL Y NO haya encontrado el el.) y el vector en la posición actual sea menor al valor buscado, hago lo siguiente

Si vector en la posición auxiliar es el valor buscado

encontrado es igual a verdadero

sino

sumo uno a auxiliar

Búsqueda dicotómica

Declaro una variable booleana, donde guardo si encuentro o no el valor.

Declaro una variable mitad, para su posterior uso.

Declaró y asignó el inicio y final del vector.

mientras el inicio sea menor o igual al final, y no haya encontrado el valor, repito lo siguiente

mitad es igual a inicio mas final, dividido 2

Si el arreglo en la posición mitad, es mayor al valor buscado

final es igual a mitad - 1

Sino, si arreglo en la posición mitad, es menor al valor buscado

inicio el igual a mitad + 1

Sino

encontrado es igual a verdadero

**PARCIAL HECHO EN CLASE**

<https://github.com/NicaOB/Parcial-CADP-Resuelto>