Final de CADP.

Definiciones:

Algoritmo: Especificación rigurosa de la secuencia de pasos (instrucciones) a realizar sobre un autómata para alcanzar un resultado deseado en un tiempo finito.

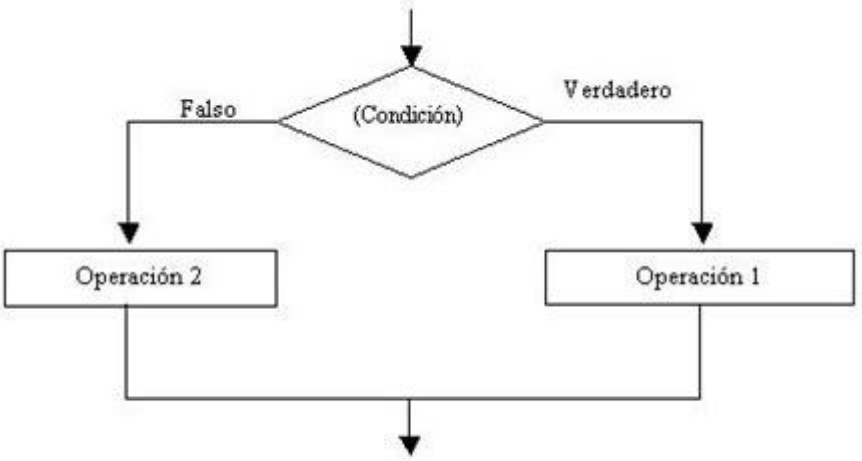
Dato: Es una representación de un objeto del mundo real mediante la cual podemos modelizar aspectos del problema que se quiere resolver con un programa mediante una computadora (*Rango de valores posibles, conjunto de operaciones permitidas, representación interna*).

- **Simple:** Tienen un único valor en un momento determinado.
    - **Definido por el lenguaje:** Son provistos por el lenguaje.
    - **Definido por el programador:** Permiten definir nuevos tipos de datos a partir de los tipos simples.
- **Variables:** Es una zona de memoria cuyo contenido puede cambiar su valor durante el programa.
- **Constante:** Es una zona de memoria cuyo contenido no puede cambiar de valor durante todo el programa.
- **Valores:**
    - **Pre-condición:** se conoce verdadera antes de iniciar el programa (ó módulo).
    - **Post-condición:** Información que debería ser verdadera al concluir el programa (ó módulo).

Estructuras de control: Los lenguajes de programación tienen un conjunto de instrucciones que permiten especificar el control del programa.

- **Secuencia:** Está representada por una sucesión de operaciones (*Ejem: asignaciones*), el orden de la ejecución coincide con el orden físico de las instrucciones.

- **Decisión:** Estructura básica de decisión entre dos alternativas



- **Iteración:** Cuando se desea ejecutar un bloque de código desconociendo el número exacto de veces.
    - **Pre-condicional:** Evalúa la condición y si es verdadera, ejecuta el bloque de acciones (el valor inicial debe ser conocido antes de la evaluación de la condición)(o, 1, N veces).
    - **Post-condicional:** Ejecuta las acciones, luego evalúa y sigue ejecutando mientras la condición es falsa. (1, N veces).
- **Repetición:** Consiste en repetir N veces un bloque de acciones.
- **Selección:** Permite realizar distintas acciones dependiendo el valor de una variable de tipo ordinal.

Ventajas Datos Definidos por el Usuario:

- Aumenta la riqueza expresiva del lenguaje.
- Mayor seguridad respecto de las operaciones realizadas.
- Límites preestablecidos sobre los valores posibles.
- **Flexibilidad:** En caso de modificar la representación del dato solo se debe modificar una declaración en lugar de un conjunto.
- **Documentación:** Se pueden usar identificadores auto-explicativos, facilitando el entendimiento y lectura del programa.

- **Seguridad:** Se reducen los errores por uso de operaciones inadecuadas.

Subrango:

- Es de tipo ordinal, consiste en una sucesión de valores de un tipo ordinal (*predefinido por el usuario*).
    - **Simple.**
    - **Ordinal.**
    - **Existe en mayoría de lenguajes.**

Modularización: Significa dividir el problema en partes funcionalmente independientes, que encapsulan datos y operaciones.

- **Módulo:** Encapsula tareas o funciones. Tarea específica bien definida.
- **Ventajas:**
    - **Mayor productividad:** Al dividir un problema en módulos, un equipo de desarrollo puede trabajar simultáneamente.
    - **Reusabilidad:** Un objetivo fundamental para la ingeniería de software es la reusabilidad, es decir, el reuso de software.
    - **Facilidad de crecimiento:** Permite disminuir y reducir costos de incorporar nuevas prestaciones.
    - **Legibilidad:** Una mayor claridad para leer y comprender el código fuente.

Alcance de variables:

- **Variables local del Programa:** Pueden ser usados por todo el programa.
- **Variable locales al proceso:** Solo pueden ser usados por el proceso en el que son declarados.

En un Proceso: Primero se busca si es una variable local al proceso, o si es un parámetro, o si es global.

En un Programa: Primero se ve si es local al programa o si es global.

Parámetros:

- La solución al problema de las variables globales es el **Data Hiding** (ocultamiento de datos), esto quiere decir que los datos solo son visibles por el módulo.
- **Parámetros por valor:** Un dato de entrada (**IN**), significa que el módulo recibe un valor proveniente de otro módulo y hace una copia del valor (podrá realizar operaciones dentro del módulo, pero no cambiará su valor fuera del módulo)
- **Por referencia:** (**IN/OUT**) Recibe una dirección de memoria (su valor cambia hacia fuera del módulo).

Estructura de datos: Permite al programador definir un tipo al que se asocian diferentes datos que tienen valores lógicamente relacionados.

Elementos

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

Tamaño

- **Estática:** El tamaño de la estructura no varía durante la ejecución del programa.
- **Dinámica:** El tamaño de la estructura varía durante la ejecución del programa.

Acceso

- **Secuencial:** Para acceder a un elemento particular se debe respetar un orden predeterminado.
- **Directo:** Se puede acceder directamente a un elemento particular directamente.

Linealidad

- **Lineal:** Está formada por ninguno, uno o varios elementos que guardan una relación de adyacencia.
- **No-Lineal:** Pueden existir 0, 1 o más elementos.

Arreglo: Es una estructura de datos **compuesta** que permite acceder a cada elemento por medio de un índice.

- **Homogénea:** Los elementos 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 se debe utilizar una variable **índice** que es de tipo ordinal

  Recorridos:

  - **Recorrido total:** Implica analizar todos los elementos del vector, que implica recorrer todo el vector.
  - **Recorrido parcial:** Implica analizar los elementos del vector, hasta encontrar aquel que cumple con lo pedido. Puede ocurrir que se recorra todo el vector.

Dimensión del arreglo:

- **Física:** Se especifica en el momento de la declaración y determina su ocupación máxima de memoria.
- **Lógica:** Se determina cuando se cargan contenidos a los elementos del arreglo. Indica la cantidad de posiciones de memoria ocupadas con contenido real. No puede superar la dimensión física.

Búsqueda en un vector:

- **Vector 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.
- **Vector Ordenado:** Se debe aprovechar el orden, existen al menos dos formas: Búsqueda mejorada o búsqueda dicotómica.

Alocación estática y dinámica:

- **Estática:** Las variables y tipos reservan memoria en su declaración y se mantienen durante todo el programa.
- **Dinámica:** Permite modificar en tiempo de ejecución la memoria utilizada.

Tipo Puntero: (*Simple)*. 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.

- Pueden apuntar solamente a direcciones almacenadas en memoria dinámica (***HEAP***).
- Solo pueden apuntar a un único dato.
- Pueden reservar y liberar memoria durante la ejecución del programa.
- **Dispose:**
    - Libera la conexión que existe entre la variable y la posición de memoria.
    - Libera la posición de memoria.
    - La memoria queda libre para volver a utilizarse.
- **Asignar a NIL (A:=NIL):**
    - Libera la conexión que existe entre la variable y la posición de memoria.
    - La memoria sigue ocupada
    - La memoria no se puede referenciar ni utilizar.

Estructura Lista: (***Estructura dinámica***). Es una 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.

- **Homogénea:** Los elementos son del mismo tipo.
- **Dinámica:** La cantidad de nodos puede variar durante la ejecución.
- **Lineal:** Cada nodo tiene un único antecesor y sucesor.
- **Acceso:** El acceso a cada elemento es de manera secuencial.

Corrección y eficiencia:

- **Testing:** Evidenciar que el programa hace el trabajo esperado.
    - Decidir cuáles aspectos deben ser testeados y encontrar datos de prueba para cada aspecto.
    - Determinar el resultado que se espera que el programa espera.

    - Poner atención en los casos límites.
    - Diseñar casos de prueba sobre la base de lo que hace el programa (hacerlo antes de escribir el programa).
- **Debugging:** Descubrir y reparar la causa del error; el error puede ser porque **el programa no es el adecuado** y porque **el programa no está escrito correctamente**.
    - **Errores sintácticos:** Se detectan en la compilación.
    - **Errores lógicos:** Se detectan durante la ejecución.
    - **Errores de Sistema:** (Son raros).
- **Walkthrough:** Recorrer el programa frente a una audiencia que no tiene preconceptos o ideas sobre de qué va el programa, por lo cual puede descubrir errores.
- **Verificación:** Controlar que cumpla las pre y post condiciones.

Eficiencia: Analiza el tiempo de ejecución y la cantidad de memoria que requiere.

- Datos de entrada (tamaño y cantidad).
- Calidad del código generado.
- Naturaleza y rapidez de la ejecución.
- El tiempo del algoritmo base.

Algoritmos eficientes:

Algoritmo A.

```
m:=a
if b<m then m := b;
if c<m then m := c;
```

Algoritmo B.

```
if(a<=b) and (a<=c)
then m:=a
else if b<=c then m := b
else m := c;
```

- El algoritmo A lo encontramos más eficiente ya que realiza dos comparaciones y al menos una asignación, mientras que el algoritmo B, puede llegar a hacer 3 comparaciones y una asignación

También debemos evitar realizar cálculos innecesarios, como por ejemplo el siguiente algoritmo:

```
y:=0;
for n:=1 to 2000 do
      y:= y+1/(x*x*x-n)
```

Este algoritmo realizaría el cálculo de x*x*x-n 2000 veces, cosas que podemos evitar haciendo el siguiente algoritmo:

```
t:= x*x*x;
y:=0
for n=1 to 2000 do
      y:=y+1/(t-n);
```

A veces los códigos más compactos no son los más eficientes.

Tiempo de ejecución: Algunos algoritmos no depende de la características de los datos sino de la cantidad de datos de entrada o su tamaño.
Otros dependen de su función de “entrada” específica.

- **Análisis empírico:** Se realiza el programa y se mide el tiempo consumido. (Fácil de realizar; requiere ejecutar el algoritmo varias veces).
- **Análisis teórico:** Encontrar una cota máxima para expresar el tiempo de nuestro algoritmo.
  Se consideran: las instrucciones elementales (asignación, aritmético/lógicas).

Una instrucción elemental utiliza un tiempo constante para su ejecución. Cada una de ellas se ejecuta en una **unidad de tiempo** (**1T**).

Ejemplo:

```
Begin
      aux:= 58; //Asignación = 1T.
      aux:= aux * 5; //Multiplicación + asignación = 2T.
      temp:= aux; //Asignación = 1T.
      read (x); //NO SE CONSIDERA.
End.
```

Ejemplo 2:

```
if (aux > 45) //Evaluar condición = 1T.
```

Ejemplo 3:

```
for i:= 1 to 10 do
{
Tiempo del for = (3(N) + 2) + N(cuerpo)
N = 10.
(cuerpo) = (suma + asignación = 2UT) +
(multiplicación + asignación = 2UT)
}
      x:= x + 1;
      temp:= aux * 2;
```

Ejemplo 4:

```
while (aux > 0) do
{
Tiempo del while N+1 (eval. cond.)+N(cuerpo)
}
```

Cálculo de memoria (Estática y dinámica):

```
Program dos;
Type
puntero = ^real;
puntero2 = ^char;
persona = record
nombre:string[20];
dni:integer;
end;
Var
p:puntero; //LOS 4 BYTES DECLARADOS (YA QUE SON DE TIPO PUNTERO).
q:puntero2; //LOS 4 BYTES DECLARADOS (YA QUE SON DE TIPO PUNTERO).
per:persona; //LOS 21 + 4 BYTES DECLARADOS.
precio:real; //8 BYTES DECLARADOS.
Begin
new (p); //8 BYTES DE DINÁMICA, YA QUE SE HACE EL NEW() DE UN TIPO REAL.
End
```

TOTAL MEMORIA ESTÁTICA: 37 BYTES.

TOTAL MEMORIA DINÁMICA: 8 BYTES.

Tabla de ocupación:

- char = (**1 byte**)
- boolean = (**1 byte**)
- integer = (**4 bytes**)
- real = (**8 bytes**)
- string = (**tamaño + 1 byte**)
- subrango = (**depende el tipo**)
- registro = (**suma de sus campos**)
- arreglos = (**dimFísica*tipo elemento**)
- puntero = (**4 bytes**)

Algoritmos eficientes:

Vector:

- **Insertar en vector**

```
Procedure insertar( var vector:vector; var dl:integer;
valor:integer; pos:integer);
Var
i:integer;
Begin
if(((dl+1) <= dimFisica) and (pos>=1) and (pos <=dl) )then begin
//comprobamos que se puede insertar y no puede superar la dimensión
//física y comprobamos que pos sea válido.
for i:= dl down to pos do
//bajamos desde el final hasta la posición a insertar para que haga
el corrimiento de los elementos (es decir vayan aumentado una
posición para dejar la posición deseada libre).
a[i+1] := a[i];
//Asignamos el elemento que está en i en i+1, es decir, lo subimos
una posición.
a[pos]:=valor;
//Asignamos el valor deseado en la posición deseada.
dl := dl +1;
//Aumentamos la dimensión lógica.
end;
end;
```

- **Borrar en vector**

```
Procedure borrar (var a :números; var dim:integer;
var
pos:integer;
Begin
if ((pos>=1) and (pos<=dim))then begin
//verificamos que la posición que se desea eliminar sea válida
for i:= pos to (dim-1) do //desde la posición hasta la dimensión
hacemos el corrimiento.
a[i]:= a[i+1]; // El valor de a[pos] se aplasta con el valor de
//a[i+1], y así se hace el corrimiento.
dim:= dim - 1; //Reducimos la dimensión lógica.
end;
end;

function buscarDesordenado (a:números; dim:integer, valor:integer):
boolean;
Var
pos:integer; esta:boolean;
Begin
esta:= false; pos:=1;
//Usamos pos como índice, y un booleano para cortar cuando
//encuentre.
while ( (pos <= dim) and (not esta) ) do begin
//Hacemos la comprobación de corte
if (a[pos]= valor) then esta:=true //Evaluamos si lo encontramos.
else pos:= pos + 1; //Sino aumentamos la posición para seguir
//buscando.
end;
buscar:= esta;
end.
```

- **Búsqueda binaria en vector**

```
Procedure BusquedaBinaria ( Var vec: números; dimL: integer;
bus: integer; var ok : boolean);
Var
pri, ult, medio : integer;
//tenemos pri, que toma valor 1; ult, que toma valor diml; y medio
//que toma valor de medio := (pri+ult) div 2;
Begin
ok:= false; pri:= 1 ; ult:= dimL; medio := (pri + ult ) div 2 ;
While ( pri < = ult ) and ( bus <> vec[medio]) do
//Sigue hasta que sea igual al valor o pri supera ult.
begin
if ( bus < vec[medio] ) then ult:= medio -1 ;
//Si el valor en el que está el vector es más alto que el medio,
necesitamos nuevo medio.
else pri:= medio+1 ;
//Si es mayor es también necesitamos nuevo medio,
medio := ( pri + ult ) div 2 ;
end;
if (pri <=ult) and (bus = vec[medio]) then ok:=true;
end;
```

Lista:

- **Agregar elemento adelante de la lista.**

```
Procedure agregarAdelante(var p:lista; num:integer);
Var nuevo:lista;
Begin
new(nuevo); nuevo^.dato:=num; nuevo^.sig:=nil;
if(p=nil) then p:=nuevo;
//Si la lista está vacía entonces p es igual al puntero nuevo.
else begin
      nuevo^.sig:=p;
      p:=nuevo;
//Sino enlazamos nuevo delante de p y asignamos p a nuevo para
//completar el enlace.
end;
end;
```

- **Agregar elemento al final de la lista**

```
Procedure agregarFinal(var p:lista; num:integer);
var
nuevo, aux:lista;
Begin
new(nuevo); nuevo^.dato:=num; nuevo^.sig:=nil;
if(p=nil) then p:=nuevo;
//Evaluamos si la lista está vacía.
else begin
aux:=p;
while(aux^.sig <> nil) do
      aux:=aux^.sig;
//Avanzamos hasta llegar a nil
end;
aux^.sig:=nuevo; //Lo asignamos.
```

- **Agregar elemento al final de la lista (eficiente)**

```
Procedure agregarAlFinal2(var p,pUlt:lista; num:integer);
Var nuevo:lista;
Begin
new(nuevo); nuevo^.dato:=num; nuevo^.sig:=nil;
if(p=nil) then begin
      p:=nuevo;
      pUlt:=nuevo;
end
else begin
      pUlt^.sig:=nuevo;
      pUlt:=nuevo;
end;
end;
//Éste agregar nos evita recorrer la lista siempre ya que vamos tener el último nodo disponible para enlazar a otro elemento.
```

- **Busca elemento en la lista**

```
Function buscarElemento(p:lista; valor:integer):boolean;
Var aux:lista; encontre:boolean;
Begin
encontre:=false; aux:p;
while((aux <> nil) and (aux^.dato < valor))do begin
aux:=aux^.sig;
end;
if(aux <> nil) and (aux^.dato = valor) then encontre := true;
buscar:=encontre;
end;
```

- **Insertar elemento en la lista**

```
Procedure insertar(var l:lista; valor:integer);
Var nuevo,actual,ant:lista;
new(nuevo); nuevo^.dato:=valor; nuevo^.sig:=nil;
if(p=nil) then p:=nuevo;
//En caso de que la lista esté vacía.
else begin
      actual:=p; ant:=p;
      while(actual <> nil) and (actual^.dato < nuevo^.dato) do
      begin
            ant:=actual;
            actual:=actual^.sig;
      end;
//Recorremos la lista hasta encontrar la posición indicada para
//insertar.
if(actual = p) then begin
      nuevo^.sig:=p;
      p:=nuevo;
end
//Caso en que debamos insertar al comienzo de la lista.
else if(actual <> nil) then begin
      ant^.sig:=nuevo;
      nuevo^.sig:=actual;
end
//En el caso de que el valor tenga que ir en medio de la lista.
else begin
      ant^.sig:=nuevo;
      nuevo^.sig:=nil;
end;
//En el caso de que vaya al final de la lista.
end;
```

- **Eliminar elemento**

```
Procedure eliminar(var p:lista; valor:integer);
Var
actual,ant:lista;
Begin
actual:=p;
while (actual <> nil) do begin
if (actual^.elem <> valor) then ant:=actual; actual:= actual^.sig;
//Recorremos la lista hasta encontrar el elemento o que no esté.
else begin
if (actual <> nil) then begin
     if (actual = p) then p:= p^.sig
//Si está al principio hacemos que p apunte al siguiente.
     else
          ant^.sig:= actual^.sig;
//Si está en el medio hacemos el re-enlace entre los nodos anterior
//y siguiente al actual, que queremos eliminar.
     end;
      dispose (actual);
//Hacemos el dispose y eliminamos el nodo.
end;
```