*Resumen Final CADP*

Un **programa** es un conjunto de órdenes o instrucciones ejecutables sobre una computadora que permiten cumplir con una función específica (dichas instrucciones están escritas en un lenguaje de programación concreto).

**Modularizar:** Significa dividir un problema en partes funcionalmente independientes, que encapsulan operaciones y datos.

Se trata de separar tareas con datos propios y de comunicación perfectamente especificados que se comunican entre sí.

Un **dato** es una representación de un objeto del mundo real. Permiten modelizar los aspectos del problema que se quiere resolver mediante un programa ejecutable en una computadora. Hay datos constantes y datos variables. Cada dato debe tener asociado un *tipo de dato.*

Los **tipos de datos** son una forma de identificar y clasificar los datos utilizados en programación.

Cada uno tiene un rango de valores posibles, una representación interna (es decir, un número máximo y un número mínimo) y están ligados a un conjunto de operaciones permitidas específicamente para ese tipo de dato.

Resultan útiles para la gestión de la información.

Los tipos de datos pueden ser **simples o compuestos**.

Simples: puede tomar un único valor a la vez.

Compuestos: pueden tener varios valores a la vez (guardan alguna relación lógica entre ellos)

Una **constante** representa un dato que no cambia durante la ejecución de un programa. Este tipo de dato puede ser numérico, lógico o carácter.

Un **dato ordinal** es aquel que tiene sus elementos ordenadores discretamente.

Esto significa que para cada elemento (que es parte del tipo) existe un elemento anterior y otro posterior.

Por esta razón los tipos de datos enteros, carácter y lógico se denominan tipos de datos ordinales. Los reales quedan excluidos porque se sabe que entre dos

**Estructuras de control:** permiten cambiar el flujo de ejecución de un programa.

La estructura de control más sencilla es la secuencia, ya que está representada por una sucesión de operaciones en la que el orden de ejecución coincide con el orden físico de aparición de las instrucciones.

Sin embargo, ciertos problemas no requieren algoritmos que sean secuenciales, sino que contemplen condiciones para ejecutar determinadas instrucciones.

* Secuencia: conjunto de instrucciones que se ejecutan una debajo de otra.
* Repetición (**for indice := \_\_ to/downto \_\_ do**): extensión de la secuencia. Consiste en repetir N veces un bloque de acciones. El número de veces que se deben ejecutar es fijo y conocido de antemano.  
  La variable de control debe ser de tipo ordinal y NO puede ser modificada dentro del lazo. Los incrementos o decrementos y testeos son implícitos.
* Decisión (**if \_\_ then - else**): toma decisiones en función de los datos del problema. Hay 2 alternativas: o cumple la condición (TRUE → se ejecutan las instrucciones dentro del “if”) o no cumple la condición (si hay un “else” ejecuta sus instrucciones, sino avanza a la próxima instrucción).

* Selección (**case \_\_ of**): extensión de la decisión; es útil cuando se pueden dar más de dos casos. Puede haber más de un valor en cada una de las entradas, pero cada valor puede ser evaluado en una única entrada (es decir, no se puede poner la misma condición en distintas entradas).  
  La variable debe ser de tipo ordinal y deben incluirse todas las posibilidades.
* Iteración: cuando desea ejecutar un bloque de instrucciones desconociendo el número de veces que se ejecutan se utilizan estructuras de control iterativas condicionales (las acciones se ejecutan dependiendo de la evaluación de la condición definida).  
    
  Estas estructuras pueden ser
* Pre-condicionales (**while \_\_ do**): evalúan la condición antes de ejecutar el bloque de instrucciones. Esto hace que el bloque se pueda ejecutar 0, 1 o más veces.

Post-condicionales (**repeat  until \_\_** ): primero se ejecuta el bloque de instrucciones, luego se evalúa la condición. Si es falsa se ejecuta nuevamente. Esto implica que el bloque se pueda ejecutar 1 o más veces (pero siempre POR LO MENOS 1).

Un **tipo de dato definido por el usuario** es aquel que no existe en la definición del lenguaje, donde el usuario es el encargado de determinar su denominación y el conjunto de valores y operaciones que estarán permitidas para el mismo.

Los tipos de datos definidos por el usuario permiten un lenguaje con:

* Mejores posibilidades de abstracción de datos. Esto permite mayor claridad para la lectura de los programas.
* Mayor seguridad respecto de las operaciones que se llevan a cabo sobre cada clase de datos, permitiendo mayor número de validaciones.
* Límites preestablecidos sobre los valores posibles que pueden tomar las variables que corresponden al elemento del mundo real asociado.

Cada **declaración de tipos** (type) define identificadores que pueden ser utilizados en declaraciones de otros tipos, variables, procedimientos y funciones posteriores. Se deben utilizar nombres descriptivos para identificar los datos de un programa de forma tal que se pueda conocer su dirección real en la memoria y el valor que contiene.

Las ventajas que presentan son:

* Flexibilidad: en caso de que sea necesario modificar la forma en que se representa la información solo se debe modificar una declaración en lugar de una serie de declaraciones de variables.
* Documentación: se pueden asignar nombres de tipo identificadores que representan la manera en que deben ser usados y no como están constituidos, facilitando el entendimiento y la lectura del programa.
* Seguridad: con su declaración se reducen los errores de correspondencia entre el valor que se pretende asignar a una variable y el previamente declarado. Al realizar los chequeos sobre el código será más fácil distinguir errores, de esta manera pueden obtenerse programas más confiables.

Los tipos de datos definidos por el usuario nos permiten crear un nuevo tipo que se corresponda con un grupo de objetos del mundo real.

Este nuevo **tipo de dato** será **enumerativo**.

Un **procedimiento** es un conjunto de instrucciones que realizan una tarea específica y como resultado del cual puede retornar cero, uno o más valores como respuesta.

Una **función** es un módulo que realiza una tarea específica y que como resultado de ella retorna un único valor como respuesta.

Si bien tienen una función similar, existen **diferencias** entre estos dos módulos: Las funciones retornan siempre un único resultado, mientras que los procedimientos pueden retornar 0,1 o más valores, y las funciones indican en su definición el tipo de dato que retorna como resultado.

Se denomina **parámetros** a la serie de datos con los que se comunican los módulos. Los parámetros de un módulo deben definirse en el encabezado del mismo. Cada parámetro debe especificar el tipo de datos con el que se corresponde.

En la invocación del módulo deben definirse también estos parámetros. Aquellos que se definen en el llamado del módulo reciben el nombre de **parámetros actuales**, mientras que los parámetros descritos en el encabezado del módulo invocado (procedimiento o función) se denominan **parámetros formales**.

Una **variable local** es aquella que está declarada y definida dentro de un programa o módulo, en el sentido que está dentro de ese módulo y es distinta a cualquier variable que tenga el mismo nombre y que estuviera declarada en otro lugar del programa.

Una **global** es aquella que está declarada en el programa y que puede ser utilizada por cualquier módulo de este.

Una **estructura de datos** es un conjunto de variables (no necesariamente del mismo tipo) relacionadas entre sí de diversas formas que representa un objeto o elemento de un problema.

Una estructura de datos puede ser homogénea o heterogénea dependiendo de los tipos de datos que componen su estructura, y también se pueden clasificar de acuerdo a la ocupación de memoria estática y dinámica.

Una estructura de datos es **homogénea** si los datos que la componen son todos del mismo tipo, y es **heterogénea** si los datos que la componen son de distinto tipo.

Es **estática** si la cantidad de elementos que contiene es fija, es decir, si la cantidad de memoria que se utiliza no varía durante la ejecución del programa, en tanto es **dinámica** si el número de componentes es variable pues da lugar a que la memoria ocupada cambie a lo largo de la ejecución del programa.

Estructuras estáticas:

Desventaja: el **mal aprovechamiento de la memoria**, ya que si contienen pocos elementos se desperdicia lugar, así como tampoco tienen posibilidad de recibir más elementos de los indicados inicialmente.

Ventaja: su reserva inicial de memoria y **acceso directo a cualquiera de sus elementos**, ya que la posición de cualquiera de ellos puede calcularse como un desplazamiento de la posición inicial de la estructura.

Estructuras dinámicas:

Ventaja: **pueden reservar la memoria necesaria** para cada uno de sus elementos a medida que el algoritmo lo requiera; esto implica un mejor uso de la memoria y la posibilidad de extender las estructuras según las necesidades del problema.

Desventaja: **los datos deben ser accedidos en forma secuencial** dado que sus posiciones no son consecutivas físicamente en la memoria

Los **registros** son uno de los tipos de datos estructurados más utilizados ya que permiten agrupar datos de diferentes clases y con una conexión lógica en una única estructura.

Son un conjunto de valores con tres características básicas:

1. Los valores pueden ser de distinto tipo (ósea, los registros son estructuras heterogéneas).
2. Los valores almacenados en un registro se llaman campos y cada uno tiene un identificador. Pueden ser accedidos de forma directa, sin necesidad de pasar por todos los anteriores para acceder a uno en específico.
3. El almacenamiento ocupado por un registro es fijo (ósea, los registros son estructuras estáticas).

Los **arreglos** son datoscompuestos indexados (ordenados por un índice) que permiten operar sobre cualquier elemento de la estructura especificando su **posición** en la misma

Un **tipo de dato arreglo** (*array*) es una colección ordenada e indexada de elementos, con las siguientes características:

1. Todos los elementos son del mismo tipo (es un tipo homogéneo).

2. Los elementos pueden recuperarse en cualquier orden simplemente indicando la posición que ocupan dentro de la estructura (por este motivo es una estructura indexada).

3. La memoria ocupada a lo largo de la ejecución del programa es fija (por ello es una estructura de datos estática).

Una **lista** es un conjunto de elementos de tipo homogéneo donde los mismos no están almacenados de forma secuencial en la memoria; esto significa que, a pesar de mantener un orden lógico interno, los componentes de una lista pueden aparecer físicamente dispersos en la memoria.

Con las estructuras estáticas dadas durante la cursada (registros y arreglos) se vio que una vez dada la declaración de su estructura, queda definida su forma y ubicación en memoria. Con las listas no pasa esto porque son estructuras dinámicas.

Un **puntero** es un tipo de variable en la cual se almacena la dirección de un dato, permitiendo manejar direcciones "apuntando" a un elemento determinado.

|  |  |
| --- | --- |
| NIL | DISPOSE |
| Libera la conexión que existe entre la variable y la posición de memoria.    La memoria sigue ocupada pero no se puede referenciar ni utilizar. | Libera la posición de memoria de forma que la memoria liberada pueda ser utilizada en otro momento en el programa. |

* Diferencia entre Arreglos y Listas:

La principal diferencia es que los arreglos son estructuras estáticas lo cual nos obliga a tener una dimensión física; mientras que las listas son estructuras de datos dinámicas que nos permiten ir agregando o eliminando elementos a lo largo del programa siempre que sea requerido (y la memoria lo permita).  
Los arreglos son una estructura indexada (necesitan un índice para acceder al contenido del arreglo); mientras que las listas son una estructura lineal.  
Las dos estructuras son homogéneas.

Un programa es correcto si cumple con las especificaciones del problema a resolver. Por esta razón es que la especificación debe ser completa, precisa y exenta de ambigüedades.

Para medir esto el programador cuenta con procesos de **verificación y validación.**

1 .Verificación implica comprobar que el programa cumple con las precondiciones y postcondiciones del mismo y corroborar que el sistema cumple los requerimientos funcionales y no funcionales que se le han especificado. Las tecnicas de correcion se aplican en la etapa de verificacion.

2. Validación (¿Estamos construyendo el producto concreto?): proceso más general, donde se asegura que el software cumple las expectativas del cliente (hace lo que espera el cliente).

TÉCNICAS PARA MEDIR LA CORRECCIÓN: (las tres son complementarias).

Testing (pruebas): proceso mediante el cual se proveen evidencias convincentes respecto a que el programa hace el trabajo esperado.   
Estas evidencias se proveen mediante un plan de pruebas que implica generar distintos casos y analizar si hay errores o no.

Debugging (depuración): proceso mediante el cual se pueden identificar y corregir errores. Puede involucrar el diseño y aplicación de pruebas adicionales para ubicar y conocer la naturaleza del error.

Walkthrough: consiste en recorrer el programa frente a una audiencia que no tenga ideas preestablecidas sobre el y que esté dispuesta a descubrir errores u omisiones.   
También sucede a menudo que cuando no se puede detectar un error el programador trate de probar que no existe, pero mientras lo hace puede detectar el error o bien puede que el otro lo encuentre.

La **eficiencia** es la métrica de calidad de los algoritmos, asociada con una utilización óptima de TODOS los recursos disponibles del sistema donde ejecuta. Estudia principalmente la memoria utilizada y el tiempo de ejecución empleado.

Para calcular el **uso de la memoria** se debe hacer la suma de las variables que utiliza el programa multiplicando la cantidad de cada variable por la memoria que esta ocupa:

|  |  |
| --- | --- |
| TIPO | CANTIDAD DE BYTES (\*) |
| Boolean | 1 byte |
| Entero | 4 bytes |
| Real | 8 bytes |
| Char | 1 byte |
| String | Tantos bytes como indique la longitud del String + 1 |
| Record | La suma de lo que ocupa cada campo que lo compone |
| Arreglo | Dimensión física \* tipo de dato |
| Puntero | 4 bytes |

**Tiempo de ejecución:** depende de distintos factores como lo son el tamaño, contenido y disposición de los datos de entrada del programa, la calidad del código generado por el compilador utilizado, la naturaleza y rapidez de las instrucciones de máquina empleadas en la ejecución del programa y el tiempo del algoritmo base.   
Se consideran más eficientes aquellos algoritmos que cumplan con lo requerido en el menor tiempo posible. A esta clase pertenecen aquellas aplicaciones con tiempo de respuesta finito.

**Reglas generales** para el cálculo del tiempo de ejecución:

1. Sentencias consecutivas.

2. For / For anidados: cuando tengo for anidados no tengo que sumar los de adentro, sino hacer la suma correspondiente y multiplicar el resultado por la cantidad de veces que lleva el índice del for que lo engloba.

3. While / Repeat Until: se debe calcular la cantidad de operaciones elementales que se ejecutan dentro del while y multiplicarla por la cantidad de veces que se ejecuta la estructura. Como no se conoce esa cantidad se considera el peor caso y se escribe como “n”.

If / Else: en el caso de una sentencia IF en su forma completa (then/else), debe calcularse la cantidad de operaciones que se realizan en cada parte y se debe elegir aquella que consuma más tiempo (el peor caso).  
   
EFICIENCIA:

Asignación= 1UT

Operaciones= 1UT

Evaluar condición= 1UT

if/else = se toma el peor de los casos

FOR:

3\*n\*2 + cuerpo \* n

While:

(n+1)\*condiciones + cuerpo\*n

Repeat until:

(n+1)\*condiciones + cuerpo + cuerpo\*n

* OPERACIONES DE ARREGLOS (\*  \*)

**Asignar elementos:** directo utilizando el índice.

**Recorridos:** Completo → for ; Condicional → while + pos

**Lectura/Escritura:** se recorre el vector con la dimensión lógica y se lee/escribe  en la posición del índice que se haya utilizado (cuando hago read(x) uso un while y voy armando la dimL; cuando hago el write(x) uso la dimL).

**Agregar elementos:** Siempre se debe controlar no ingresar más elementos que la dimensión física declarada o en una posición que no corresponda (ej -1 en un índice 1 ... 10).

**Agregar al final:** se debe considerar la dimF y verificar que dimL + 1 <= dimF. Si esto es verdadero, actualizar la dimL; el elemento a agregar va a ocupar la posición de dimL.

**Insertar en una posición determinada:** validar la posición, verificar si hay espacio en el arreglo, hacer corrimiento de los valores hacia la derecha(a partir de la dimL con un for i:= dimL downto pos do v[i+1]:= v[i]), asignar el valor, aumentar la dimL. (!!!)

**Eliminar en una posicion determinada:** validar que la posición sea válida ( (pos >= posInicio) AND (pos <= dimL) ). Hacer corrimiento de derecha a izquierda (for i:= pos to dimL-1 do v[i]:=v[i+1] ). Tener en cuenta que dimL al final se tiene que decrementar ( dimL := dimL-1).

**Búsqueda lineal o secuencial:** Se aplica la función “Buscar” cuando los elementos no tienen orden.  
Este método requiere excesivo consumo de tiempo en la localización del elemento ya que realiza como promedio (dimL+1)/2 comparaciones. Es claro que pierde eficiencia en tanto mayor es el tamaño del arreglo. Lo primero que hay que hacer es *inicializar la variable posición en el primer valor de posición del vector*. El método consiste en buscar la posición del elemento con un while bicondicional que avance de posición en el vector (esto sería aumentar en una unidad la variable de posición) si y sólo si dicha posición es menor igual que la dimensión lógica y el elemento del vector en esa posición no es el que se buscaba.  
Esta bicondición nos obliga a evaluar al salir del mientras por cual condición salió. Si la posición final es mayor que la dimensión lógica, entonces el elemento que se buscaba no se encuentra en el vector. Este método se realiza en una función que devuelve siempre un solo valor; si se diera este último caso, entonces se debe asignar a la variable de posición un valor imposible. La última línea debe ser la asignación de la variable posición a la función.

**Búsqueda secuencial optimizada:** Se aplica cuando los elementos están ordenados, lo que nos permite recorrer el vector hasta encontrar el elemento buscado, hasta encontrar uno mayor/menor que él (dependiendo de si el vector está ordenado de forma creciente o decreciente) o bien hasta que se termina el arreglo.  
Lo primero que hay que hacer es inicializar una variable en la primera posición válida del arreglo para poder avanzar sobre este. El while que permite esto deberá tener dos condiciones: pos <= dimL y que el elemento del arreglo en esa posición no sea mayor al que estoy buscando. Siempre que ninguna de estas condiciones sean verdaderas, la variable pos aumenta un valor.  
Una vez fuera del while es necesario preguntar por que sali de la estructura de control: if (pos <= dimL) AND (arreglo [pos]  = valor) then  
La función toma el valor TRUE cuando no sale del while porque la posición no era válida, sino porque se encontró el elemento que buscaba.

**Búsqueda dicotómica:** se aplica cuando los elementos tienen orden pero no empieza desde la primera posición.

Consiste en comparar el valor buscado con el ubicado en el medio del vector. Si son iguales entonces la búsqueda termina, en caso contrario se debe continuar la búsqueda en la mitad del vector que corresponda: si el valor ubicado en el medio del vector es mayor que el buscado, entonces la búsqueda se concentrará en la mitad izquierda (siempre y cuando el vector esté ordenado de menor a mayor); en caso contrario, la búsqueda se abstrae a la mitad de la derecha.

Este proceso se repite tantas veces hasta que se acaba el vector o se encuentra el elemento.

**Insertar Ordenado:** verificar el espacio disponible (dimL + 1 <= dimF), determinar la posición en donde se va a insertar manteniendo el orden (adaptación de la búsqueda secuencial optimizada) y luego a partir de un for cuyo índice irá desde la dimensión lógica hasta la posición encontrada (downto), el vector en la posición siguiente del índice va a tener el valor del vector en la posición del índice (v[i+1]:= v[i]). Por ultimo insertar el elemento en el la posición determinada.

**Métodos de Ordenación en vectores:** llamamos “algoritmo de ordenación” al proceso por el cual un grupo de elementos puede ser ordenado. Este algoritmo es esencial porque hay problemas que requieren tener los datos en un orden específico para poder resolverlos.

Los diferentes algoritmos de ordenación difieren en dificultad, uso de memoria y tiempo empleado. Estos son selección, intercambio, intercambio con centinela (más eficiente) e inserción:

Eliminar en vector ordenado (repeticiones): Lo primero que hay que hacer es recorrer el vector, guardar la primer posición de las repeticiónes del elemento a eliminar en una variable (pos), obtener la cantidad de repeticiones del valor a eliminar recibido por parámetro (repetir 1 hasta dimL, si (v[i]=elem) entonces sumar 1 a cant (cant:=cant+1)). Hacer un corrimiento de la cantidad de elementos restantes luego de las repeticiones, hacia pos (for i:= 0 to dimL-fin-i) do v[pos+1] := v[fin+i]. Por ultimo decrementar la dimL(dimL-cant).

Operaciones Listas:

Insertar un elemento:

**Se necesita que la estructura tenga un orden e implica agregar el elemento a la lista de manera que la misma siga ordenada.**

Generar un nuevo nodo (nuevo).

Si la lista está vacía

Actualizo la dirección del nodo inicial (PI)

Sino

Preparo los punteros para el recorrido (ant,actual)

Busco la posición

Si va al principio

Asigno como siguiente del nodo nuevo al nodo inicial

Actualizo la dirección del nodo inicial (PI)

Si va en el medio

La dirección del siguiente del puntero ant es la dirección del nodo nuevo

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

sino

La dirección del siguiente del puntero ant es la dirección del nodo nuevo

La dirección del siguiente del nodo nuevo es la dirección nil

Eliminar un elemento:

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

**1.El elemento a eliminar sea el primer nodo de la lista.**

Actualizar la dirección siguiente del puntero inicial pI con la dirección siguiente de actual.

Actualizar la dirección del puntero inicial pI

Hacer el dispose

**2.El elemento a eliminar NO sea el primer nodo de la lista.**

Actualizar la dirección siguiente del puntero ant con la dirección siguiente de actual.

Hacer el dispose

procedure eliminarElemento(var pl:lista; n: integer);

var ant, act: lista;

termino: boolean;

begin

termino:= false;

if (pl<>nil) and (pl^.dato=n) then begin

pl:= pl^.sig; termino := true;

end;

ant:= pl; act:= pl;

while (act<>nil) and (termino = false) do begin

if(act^.dato = n) then begin

ant^.sig:=act^.sig; termino:= true;

end

else begin

ant:=act; act:=act^.sig;

end;

end;

if (termino) then

dispose(act);

end;