



final
Cap

Conceptos

Informática: ciencia que estudia el análisis y resolución de problemas mediante una computadora.

Computadora: máquina digital y sincrónica con cierta capacidad de cálculo numérico y lógico. Cuenta con la capacidad de comunicarse con el mundo exterior.

Concepto de programa: conjunto de instrucciones, las cuales son ejecutables sobre una computadora que permiten cumplir con un objetivo específico.

Concepto de Módulo: nos referimos al paso de del modelo general a una descomposición en subproblemas menores (módulos) los cuales deben tener en lo posible, una función bien definida en nuestro sistema.

Concepto de algoritmo: secuencia de pasos bien definidos a realizar sobre un automátor para alcanzar un resultado deseado en un tiempo finito.

Definición de dato: representación de un objeto del mundo real el cual permite darse forma a aspectos de un problema que se desea resolver con una computadora.

concepto de variable: zona de memoria que tiene un contenido y está asociada a un identificador. Puede cambiar su valor en la ejecución del programa.

Concepto de alcance de una variable: una de las características de las variables el alcance. El alcance está determinado como la zona donde la variable es definida y conocida.

Concepto de parámetro: Son variables que principalmente nos permiten transmitir información entre los módulos.

Alcance de un parámetro: dentro de la comunicación entre módulos mediante parámetros, tenemos dos formas →

referencia: se envía la dirección de memoria del elemento, por lo tanto, cualquier cambio quedará hecho al finalizar el módulo.

valor: Se hace una copia del elemento. Cualquier cambio que se haga dentro del módulo no se verá reflejado.

* Se pone la modificación, no sé si es el concepto de alcance de los parámetros.

Modelización de un problema

1. Análisis del problema: dado un problema, cuando lo analizamos se busca interpretar los aspectos esenciales, para luego expresarlo en términos precisos, tratando de encontrar una solución.
2. Modelo de una solución: análisis de la solución como sistema, descomponiendo en módulos y estudiando los datos de nuestro problema.
3. Módulos y diseño de algoritmos: traducir todos nuestros módulos a una solución algorítmica.
4. Escritura del programa en un lenguaje de programación.

→ no es parte de la modelización, sino de la resolución.

5. Verificación: luego de la escritura de nuestro programa y de chequear que este no tenga errores sintácticos, debemos verificar que cumpla con lo pedido.

Algoritmos

Estructuras de control

Secuencial ↗

el orden de ejecución se corresponde con el orden en el que están escritas las instrucciones.

Repetición ↗

consiste en repetir N veces un bloque de acciones.
(For).

Decisión ↗

consiste en evaluar una condición y realizar ciertas tareas si esta es Falsa, o otras si esta es verdadera.
(if)

Selección ↗

una extensión de la estructura de decisión, pero las alternativas son más de dos.
(case)

Iteración ↗

Son utilizadas cuando queremos realizar un bloque de acciones pero se desconoce el número de veces.

Se dividen en dos: pre-condicionales y pos-condicionales

pre-condicionales : evalúan la condición y si es verdadera, ejecutan el bloque de acciones. Ejecutandose 0, 1 o más veces. (while)

pos-condicionales : primero se ejecuta el bloque de acciones y luego se evalúa la condición. Se ejecutar 1 o más veces. (repeat-until).

Modularización

pasando del modelo general de nuestro problema a una descomposición en subproblemas menores. En consecuencia, para comprender un problema complejo del mundo real es necesario dividir y subdividir, o sea modularizar.

¿por qué es importante modularizar?

Trabajo en equipo: para trabajar en un sistema de software complejo se necesita más de una persona para desarrollarlo. Al tener un programa modularizado cada programador puede trabajar una parte del código desconociendo el resto del trabajo.

Mantenimiento del programa: a la hora de realizar modificaciones el costo es muy bajo si el código está bien modularizado, ya que el programador es más fácil de entender y tendremos menos "efectos secundarios" a la hora de realizar cambios.

Reusabilidad del código: cuando creamos un nuevo programa, de ser posible podremos volver a utilizar módulos anteriores que nos resuelvan problemas acordes.

Legibilidad: un programa bien modularizado es más fácil de comprender

Alcance de los datos

Concepto de dato **hidding** → todo dato que es relevante para un módulo, debe ocultarse del resto.

El **alcance** de un identificador es el bloque de programa donde se lo declara.

→ **locales**: se declaran dentro de un módulo y solo es conocido dentro de este.

Datos **locales** y **globales** → **globales**: son conocidas por todo el programa.

Parámetros

Los **parámetros** son variables que tienen como principal característica la comunicación entre módulos.

Existen dos tipos de parámetros:

por **valor**

Se realiza una copia del dato que estamos enviando.
El módulo que recibe la copia no puede efectuar ningún cambio sobre el dato original.

por **referencia**

se envía la dirección donde se encuentra el dato, por lo tanto podemos acceder y modificarlo.

Cuando tenemos variables **locales** al módulo solo son conocidos dentro de este, y dado que las variables **globales** son conocidas por todo el programa (nadie puede modificarlas). La solución son los **parámetros**.
Aporta **integridad** de los datos.

Procedimientos

conjunto de instrucciones que realizan una tarea específica y que pueden retornar o no, uno o más valores.

funciones

conjunto de instrucciones que realizan una única tarea y pueden retornar un solo valor.

- Debe retornar un único valor y de tipo simple.
- NO recibe parámetros por referencia porque estaríamos retornandolo más de un valor.

Alcance de variables

Locales al módulo

1. Busca si es local al módulo.
2. Busca si es parámetro.
3. Busca si es global.

Locales al principal

1. Busca si es local al programa principal.
2. Busca si es global.

Estructuras de datos

Las estructuras de datos son un conjunto de variables que no necesariamente son del mismo tipo pero sí están relacionadas entre si.

Simples → representan un único valor (integer, char).

se clasifican por
-elementos
- acceso
- tamaño
- linealidad

Compuestas → pueden contener más de un valor.

Homogéneas → todos los elementos son del mismo tipo. (arreglos)

Heterogéneas → los elementos pueden ser de distintos tipos. (registros).

Estráicos → cantidad de elementos fija.

Dinámico → la cantidad de elementos puede variar durante el tiempo de ejecución.

Acceso Secuencial → debemos respetar el orden para acceder a un elemento.

Acceso directo → podemos acceder directamente a un elemento.

lineal → a cada elemento le sigue uno y le precede uno.

no lineal → para un elemento dado pueden haber 0, 1 o más elementos anteriores o siguientes.

Registros

permiten agrupar datos de diferentes tipos pero con una relación lógica.

- Heterogénea
- compuesta por campos.
- Estrática.

Arreglos

colección ordenada e indexada de elementos.

- Homogénea
- indexada
- Estrática

Listas

colección homogénea de elementos con relación lineal que los vinculan.

- Homogénea
- dinámica
- lineal

Punteros → como los elementos de una lista no ocupan posiciones consecutivas

de memoria, necesitamos poder direccionarlos entre ellos.

por eso, cada elemento de la lista tiene un indicador que apunta al próximo elemento. De ahí nace el tipo puntero.

El puntero es un tipo de variable que almacena la dirección de memoria de un dato. Nos permite manejar direcciones apuntando a un elemento determinado.

operaciones permitidas para el tipo puntero

new(): reserva memoria,

dispose(): libera memoria

p:=nil: deja el puntero sin dirección asignada.

Corrección

Un programa es correcto si es realizado respecto a sus especificaciones.

Testing → se tratará sobre diseñar un conjunto de casos de prueba para corroborar que el programa funciona correctamente.

Debugging → descubrir y reparar las causas del error en nuestro programa agregando sentencias extras que nos permitan monitorizar el funcionamiento.

Walkthrough → recorrer el programa frente a una audiencia. Una persona ajena a nuestro programa no posee preconceptos, lo cual hace que se analice de una manera mucho más objetiva, descubriendo errores más fácilmente.

Verificación → controlar que se cumplen las pre y pos condiciones de del programa.

Eficiencia

Un algoritmo es eficiente si administra de manera correcta los recursos del sistema en donde se está ejecutando.

Debemos garantizar que el programa es correcto antes de cualquier cálculo.
(tiempo + memoria).

tiempo

Memoria

Se consideran más eficientes aquellos algoritmos que cumplen con lo pedido en el menor tiempo posible.



Algunos algoritmos no dependen de las características de los datos de entrada, sino de la cantidad y tamaño.

Se habla del "peor caso" en donde se toma en cuenta la cota de tiempo máxima.

Análisis Empírico → realizar el programa y medir el de manera manual.

* **Análisis Teórico** → encontrar una cota de tiempo máxima para expresar el tiempo sin tener que ejecutar nuestro programa.

Cálculo de tiempo → dado un algoritmo que es correcto, calcula el tiempo en base a cada una de nuestras instrucciones.

Consideramos

- asignación
- operaciones aritmética/lógicas
- evaluar condiciones

$$= 1 UT$$

* Recordar que siempre consideramos el peor caso y cuando tenemos un else, debemos evaluar ambos y quedarnos con el peor caso.

Se consideran más eficientes aquellos algoritmos que utilizan las estructuras de datos adecuadas para minimizar el uso de memoria ocupada.



Memoria Estática → está ocupada durante toda la ejecución del programa.

.CHAR

.INTEGER

.REAL

.STRING → si está acotado ($N+1$), suma 255.

.SUBRANGO → ocupa el tamaño del tipo de

data.

.REGISTRO → ocupa la suma de todas sus campos.

.ARREGLOS → dimensión física * tamaño del data

.PUNTERO → 4 bytes

*Operaciones read o write NO.

- Cuando no tengamos el N calcularemos el cuerpo y dejaremos expresada la fórmula.

Cálculo

.FOR → $3 \cdot (N) + 2 + N \cdot (\text{cuerpo})$

.WHILE → $(N+1) \cdot (\text{condiciones y operadores lógicos}) + N \cdot (\text{cuerpo})$

.IF → condición + UT del cuerpo

Orden de un algoritmo

$T(\text{alg}) = 36 \text{ UT} \rightarrow \text{constante}$

$T(\text{alg}) = 2N + 4\text{UT} \rightarrow N$

$T(\text{alg}) = 2N^2 + 4N \text{ UT} \rightarrow N^2$

Memoria Dinámica: puede cambiar su tamaño en tiempo de ejecución.

Puntero → tipo de dato definido por el programador.

Permite almacenar una dirección de memoria (en memoria dinámica) donde se encuentra nuestra data realmente.

Puntero = $\hat{\circ}$ integer

variable de tipo puntero que en su dirección de memoria hay un integer.

Cálculo → para memoria estática tendremos en cuenta variables locales y globales del programa.

Para el cálculo de memoria dinámica, sólo cuadra se libera memoria. Contabilizaremos cuando haya un NEW() o un DISPOSE().

Cuando se un dato de tipo puntero su valor será el del dato apuntado.

Pseudocódigos

Arreglos

Agregar un elemento:

1. debemos verificar que hay espacio.
2. agregar al final
3. incrementar la cantidad de la dimensión lógica.

Insertar:

1. verificar si hay lugar
2. verificar que la posición es válida.
3. hacer lugar para poder insertar el elemento.
4. incrementar la dimensión lógica

eliminar un elemento:

1. verificar que exista el elemento y sea una posición válida.
2. Hacer los corrimientos.
3. Decrementar dimensión lógica.

búsqueda lineal: comenzar desde el inicio de la estructura comparando cada uno de los elementos hasta encontrarlo o llegar al final.

vector ordenado



búsqueda mejorada

Se busca el elemento aprovechando el criterio por el cual está ordenado

búsqueda dicrómica

se calcula el elemento que está en el medio, si es el elemento que busco, entonces la búsqueda terminó. Sino, comparo con el medio y elijo la mitad que me convenga. Se repite este proceso.

ordenar vector → selección. intercambio. inserción.

selección: algoritmo de $O(n^2)$ pasadas.

por cada pasada de i , se elige el mínimo de la posición $(i+1)$ y se intercambia, si corresponde.

intercambio: parecido al de selección, pero no recorre grandes distancias, sino que realiza correcciones locales. compara ítems y adyacentes y los intercambia si están desordenados.

Al final, el elemento más grande será arrastrado como una burbuja hacia el final.

inserción: se comienza a partir del segundo elemento, suponiendo que el primero ya está ordenado. Si los dos primeros elementos están desordenados, los intercambia. Luego, toma el tercer elemento y busca su posición correcta respecto a los dos primeros.

En general, para el elemento i , busca su posición con respecto a los $i-1$ elementos anteriores.

Listas

Agregar adelante → por cada elemento a agregar, debemos modificar el puntero inicial.

1. reservo espacio en memoria.
2. asigno el elemento
3. el siguiente del nuevo elemento es el puntero inicial.
4. ahora el puntero inicial es el recientemente agregado.

Agregar al final → llevaremos dos punteros auxiliares, uno al inicio y el otro al final.

1. reservamos memoria y asignamos datos.
2. si la lista está vacía, asignamos ambos punteros.
3. actualizamos el siguiente del último como el nuevo.
4. el nuevo elemento pasa a ser último.

Insertar ordenado → llevaremos dos punteros auxiliares, un actual y un anterior para recorrer la lista.

1. reservamos memoria, asignamos los datos y ambas variables auxiliares apuntan al inicio.
2. recorremos la estructura mientras esta no se termine o hasta encontrar el lugar para insertar el elemento.
3. si la lista está vacía → asignamos el nuevo elemento como primero.
SINO
insertarmos el elemento entre el anterior y el actual.
4. realizamos el enganche.