UNIDAD I MODELOS DE REPRESENTACIÓN DE ESTRUCTURA DE DATOS.

1. LA ABSTRACCIÓN

Es una de las herramientas que nos ayuda solucionar un problema, es un mecanismo fundamental para la comprensión de problemas y fenómenos que poseen una gran cantidad de detalles, su idea principal consiste en manejar un problema, fenómeno, objeto, tema o idea como un concepto general, sin considerar la gran cantidad de detalles que estos puedan tener. El proceso de abstracción presenta dos aspectos complementarios.

1. Destacar los aspectos relevantes del objeto.

2. Ignorar los aspectos irrelevantes del mismo (la irrelevancia depende del nivel de abstracción, ya que, si se pasa a niveles más concretos, es posible que ciertos aspectos pasen a ser relevantes).

Veamos los diferentes tipos de abstracción que podemos encontrar en un programa:

1. Abstracción funcional: son unidades, librerías paquetes que determinan el comportamiento y uso de determinados módulos.

2. Abstracción de datos:

- **Tipo de datos:** proporcionado por los leguajes de alto nivel. La representación usada es invisible al programador, al cual solo se le permite ver las operaciones predefinidas para cada tipo.

- **Tipos definidos:** por el programador que posibilitan la definición de valores de datos más cercanos al problema que se pretende resolver.

- ***TDA:*** para la definición y representación de tipos de datos (valores + operaciones), junto con sus propiedades.

- **Objetos:** Son TDA a los que se añade propiedades de reutilización y de compartición de código.

3. Abstracción de control: Son los procedimientos que al invocarlos mediante un nombre donde se destaca qué hace la función y se ignora cómo lo hace. El usuario sólo necesita conocer la especificación de la abstracción (el qué) y puede ignorar el resto de los detalles (el cómo).

Si profundizamos más al mundo de la programación y sus conceptos, existen dos de estos conceptos que no se deben confundir, ellos son: tipo de datos y estructura de datos.

Un tipo de dato, en un lenguaje de programación, define un conjunto de valores que una determinada variable puede tomar, así como las operaciones básicas sobre dicho conjunto. Ahora veamos cómo se van relacionando estos conceptos. Los tipos de datos constituyen un primer nivel de abstracción, ya que no se tiene en cuenta cómo se implementan o se representan realmente la información sobre la memoria de la máquina. Para el usuario, el proceso de implementación o representación es invisible.

2. ESTRUCTURA DE DATOS.

Comencemos por establecer que estructura de datos es la forma de organizar un conjunto de datos con el objetivo de facilitar su manipulación. Un dato es la mínima información que se tiene en un sistema.

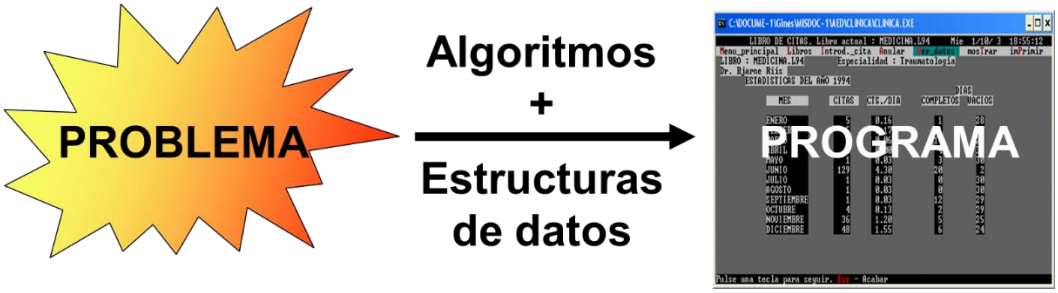
Por lo que cuando se defina una estructura de datos estaremos definiendo la organización e interrelación de éstos y un conjunto de operaciones mínimas que se pueden realizar sobre ellos como ser:

- Alta, adicionar un nuevo valor a la estructura.
- Baja, borrar un valor de la estructura.
- Búsqueda, encontrar un determinado valor en la estructura para realizar una operación con este valor

Otras operaciones que se pueden realizar son:

- Ordenamiento, de los elementos pertenecientes a la estructura.
- Apareo, dadas dos estructuras originar una nueva ordenada y que contenga a las apareadas.
- Otras, definidas de acuerdo a la característica de la estructura.

De tal forma que para resolver un problema intervienen los siguientes elementos Programa, Algoritmo y Estructura de Datos.



Problema: Conjunto de hechos/circunstancias que dificultan la consecución de algún fin.
Algoritmo: Conjunto de reglas finito e inambiguo.
Estructura de datos: Disposición en memoria de la información.
Programa: Algoritmos + Estructuras de datos.

Como ya es conocido por ustedes por lo general tenemos que tratar con datos simples (enteros, reales, booleanos, etc.) que por sí solos no nos dicen nada, ni nos sirven de mucho, es necesario tratar con estructuras de datos adecuadas a cada necesidad.

Las estructuras de datos son una colección de datos cuya organización se caracteriza por las funciones de acceso que se usan para almacenar y acceder a elementos individuales de datos.

Cada estructura ofrece ventajas y desventajas en relación a la simplicidad y eficiencia para la realización de cada operación. De esta forma, la elección de la estructura de datos apropiada

para cada problema depende de factores como la frecuencia y el orden en que se realiza cada operación sobre los datos.

Ejemplo: Partiendo de la lógica de que cuando aprendemos a programar nos piden guardar el nombre, sueldo y carnet de una persona normalmente en el código declararíamos las variables nom de tipo cadena, sueldo de tipo real y carnet de tipo cadena, sin embargo el problema se pone complejo si queremos guardar la información de 150 personas lo cual nos lleva rápidamente a la imposibilidad de declarar 450 variables en nuestro código por lo que tendremos que recurrir al uso de estructuras que permitan manejar esa cantidad de información adecuadamente.

CLASIFICACIÓN DE LAS ESTRUCTURAS DE DATOS

Las estructuras de datos se clasifican en dos considerandos la existencia de tipos de datos básicos ya establecidos y los nuevos por crear.

- **Las estructuras de datos primitivas**: son los tipos de datos fundamentales que son compatibles con una programación. Algunos tipos de datos básicos son enteros, reales, caracteres y booleanos. Etc.
- **Las estructuras de datos no primitivas**: son aquellas estructuras de datos que se crean utilizando datos primitivos. Los ejemplos de tales estructuras de datos incluyen listas, pilas, conjunto, etc. se pueden clasificar en dos categorías: lineal y no lineal
    - Lineal: Si los elementos de una estructura de datos se almacenan en un orden lineal o secuencial, entonces pertenece a la categoría lineal
    - No Lineal: Si los elementos de una estructura de datos no se almacenan en un orden secuencial, entonces es una estructura de datos no lineal

3. TIPO DE DATO ABSTRACTO

El concepto de tipo de dato abstracto (TDA, Abstract Data Types), fue propuesto por primera vez hacia 1974 por John Guttag y otros, pero no fue hasta 1975 que por primera vez Bárbara Liskov lo propuso para el lenguaje CLU.

Primera mujer en obtener un doctorado. en Ciencias de la Computación en los Estados Unidos (Stanford 1968)

DEFINICIÓN

Con mucha frecuencia se utilizan los términos TDA y Abstracción de Datos de manera equivalente, y esto es debido a la similitud e interdependencia de ambos. Sin embargo, es importante definir por separado los dos conceptos.

Como ya se mencionó, los Lenguajes de Programación Orientados a Objetos son lenguajes formados por diferentes métodos o funciones y que son llamados en el orden en que el programa lo requiere, o el usuario lo desea. La abstracción de datos consiste en ocultar las características de un objeto y obviarlas, de manera que solamente utilizamos el nombre del objeto en nuestro programa. Esto es similar a una situación de la vida cotidiana. Cuando yo digo la palabra “perro”, usted no necesita que yo le diga lo que hace el perro. Usted ya sabe la forma que tiene un perro y también sabe que los perros ladran. De manera que abstraemos todas las características de todos los perros en un solo término, al cual llamo “perro”. A esto se le llama ‘Abstracción’ y es un concepto muy útil en la programación, ya que un usuario no necesita

mencionar todas las características y funciones de un objeto cada vez que éste se utiliza, sino que son declaradas por separado en el programa y simplemente se utiliza el término abstracto (“perro”) para mencionarlo.

En el ejemplo anterior, “perro” es un Tipo de Dato Abstracto y todo el proceso de definirlo, implementarlo y mencionarlo es a lo que llamamos Abstracción de Datos.

Vamos a poner un ejemplo real de la programación. Supongamos que en algún Lenguaje de Programación Orientado a Objetos un pequeño programa saca el área de un rectángulo de las dimensiones que un usuario decida. Pensemos también que el usuario probablemente quiera saber el área de varios rectángulos. Sería muy tedioso para el programador definir la multiplicación de ‘base’ por ‘altura’ varias veces en el programa, además que limitaría al usuario a sacar un número determinado de áreas. Por ello, el programador puede crear una función denominada ‘Área’, la cual va a ser llamada el número de veces que sean necesitadas por el usuario y así el programador se evita mucho trabajo, el programa resulta más rápido, más eficiente y de menor longitud. Para lograr esto, se crea el método Área de una manera separada de la interfaz gráfica presentada al usuario y se estipula ahí la operación a realizar, devolviendo el valor de la multiplicación. En el método principal solamente se llama a la función Área y el programa hace el resto.

Al hecho de guardar todas las características y habilidades de un objeto por separado se le llama Encapsulamiento y es también un concepto importante para entender la estructuración de datos. Es frecuente que el Encapsulamiento sea usado como un sinónimo del Ocultación de información, aunque algunos creen que no es así.

SEPARACIÓN DE LA INTERFAZ E IMPLEMENTACIÓN

Cuando se usa en un programa de computación, un TDA es representado por su interfaz, la cual sirve como cubierta a la correspondiente implementación. Los usuarios de un TDA tienen que preocuparse por la interfaz, pero no con la implementación, ya que esta puede cambiar en el tiempo y afectar a los programas que usan el TDA. Esto se basa en el concepto de Ocultación de información, una protección para el programa de decisiones de diseño que son objeto de cambio.

La solidez de un TDA reposa en la idea de que la implementación está escondida al usuario. Solo la interfaz es pública. Esto significa que el TDA puede ser implementado de diferentes formas, pero mientras se mantenga consistente con la interfaz, los programas que lo usan no se ven afectados.

En la terminología de Lenguaje Orientado a Objeto, un TDA es una clase; una instancia de un TDA o clase, es un objeto

CARACTERIZACIÓN

Un TDA está caracterizado por un conjunto de operaciones (funciones) al cual le denominaron usualmente como su interfaz pública y representan el comportamiento del TDA; mientras que la implementación como la parte privada del TDA está oculta al programa cliente que lo usa. Todos los lenguajes de alto nivel tienen predefinidos TDA; que son los tipos denominados simples y las estructuras predefinidas, y estos tienen sus interfaces públicas que incluyen las operaciones como la +, -, *, etc. no se necesita conocer como actúan tales operadores sobre la

representación interna de los tipos definidos, que además, suele ser una implementación bastante dependiente de la máquina sobre la que trabaje el compilador. Lo interesante es que los lenguajes actuales nos van a permitir ampliar los TDA predefinidos con otros que serán definidos por el propio programador para adecuar así los tipos de datos a las necesidades de los programas.

Los TDA que nos van a interesar de ahora en adelante son aquellos que reflejen cierto comportamiento organizando cierta variedad de datos estructuradamente. A esta forma estructurada de almacenar los datos será a la que nos refiramos para caracterizar cada TDA.

Nótese que cuando hablemos de un TDA no haremos ninguna alusión al tipo de los elementos sino tan sólo a la forma en que están dispuestos estos elementos. Sólo nos interesa la estructura que soporta la información y sus operaciones. Para determinar el comportamiento estructural basta con observar la conducta que seguirán los datos.

Caractericemos entonces los TDA. Un TDA tendrá una parte que será invisible al usuario la cual hay que proteger y que se puede decir que es irrelevante para el uso del usuario y está constituida tanto por la maquinaria algorítmica que implemente la semántica de las operaciones como por los datos que sirvan de enlace entre los elementos del TDA, es decir, información interna necesaria para la implementación que se esté haciendo para ese comportamiento del TDA. **Resumiendo, podemos decir, que tanto la implementación de las operaciones como los elementos internos del TDA serán privados al acceso externo y ocultos a cualquier otro nivel**.

Un TDA representa una abstracción:

- Se destacan los detalles (normalmente pocos) de la especificación (el qué).
- Se ocultan los detalles (casi siempre numerosos) de la implementación (el cómo).

ESPECIFICACIÓN DE UN TDA.

La especificación de un TDA se puede representar de dos formas las cuales son:

- Especificación no formal/Informal
- Especificación formal.

Especificación no formal (Usando lenguaje natural) Para la especificación de un tipo de dato abstracto informa se debe seguir el siguiente esquema:

1. Elementos que conformarán la estructura de datos.
   - ▸ Tipo de datos: Nombre del tipo.
   - ▸ Valores: Descripción de los posibles valores
   - ▸ Operaciones:Cita de cada operación

2. ***Descripción de las operaciones*** de la estructura: Se describe cada una de las operaciones sobre el TDA.

   Nombre de la operación.
   Descripción breve de su utilidad.
   Datos de entrada a la operación.
   Datos que genera como *salida* la operación.
   Precondición: Condición que deberá cumplirse antes de utilizar la operación
   Poscondición: Condición en que queda el TDA después de ejecutar la operación

Especificación formal (Usando pseudocodigo o un lenguaje de programacion) y esta constituido por los siguientes puntos.

1. Tipo: Nombre del TDA
2. Sintaxis: Forma de las operaciones
   Nombre de la operación (argumentos) →resultado
3. Semántica: Significado de las operaciones
   Nombre de la operación(valores particulares) → expresión resultante

Para mayor comprensión se presenta parte del TDA Cadena

NO FORMAL	FORMAL/ SEUDOCÓDIGO O LENGUAJE.

I. Elementos que conformarán la estructura de datos.
TDA Cadena (VALORES Todos los caracteres alfabéticos visibles, OPERACIONES Crear, AdicionarCaracter, Llena, InsertaCarácter, EliminaCaracter, NumeroCaracters)
2. Descripción de las operaciones de la estructura.
<u>Crear</u>
Utilidad: Sirve para inicializar la cadena en vacío
Entrada: Cadena S que será inicializada
Salida: Cadena S inicializada.
Precondición: Ninguna.
Poscondición: La cadena S tiene 0 caracteres
 	
<u>AdicionarCaracter</u>
Utilidad: Sirve para agregar un carácter al final de una cadena.
Entrada: Cadena S y carácter L que se añade a la cadena S.
Salida: Cadena S modificada.
Precondición: La cantidad de caracteres en S < 80.
Poscondición: La cadena S tiene el carácter L que queda al extremo derecho de la cadena
 	
<u>Llena</u>
Utilidad: Sirve para verificar si una cadena está llena o no.
Entrada: .Cadena S que será verificada.
Salida: Si longitud de S>=80 VERDAD, sino FALSO
Precondición: Ninguna
Poscondición: Ninguna (pues la cadena S no se modifica).
……
	Opción 1
	Tipo: Cadena
	Sintaxis:
	Crear(S)
	AdicionarCaracter (S,L)
	Llena(S)
	InsertarCaracter(S,P, L)
	EliminarCaracter(S, P)
	NumeroCarcateres(S)
	…
	Semantica
	Para todo L que pertenece a cadena S
	Llena(crea()) → falso
	Numero caracteres(crea())→0
	…
	
	Opcion 2
	class Cadena {
	private:
	// Datos miembro de la clase
	char S[79];
	public:
	// Funciones miembro
	Void AdicionarCaracter(char L);
	Void EliminarCaracter(int P)
	Void insertarCaracter(int P, char L)
	Bool llena()
	}

4. FORMAS DE IMPLEMENTACIÓN DE LOS TDA.

Como se pudo apreciar en la lectura del presente documento se ha evidenciado que la implementación de los TDA es independiente de la especificación de los mismos lo cual ratifica

lo indicado anteriormente “Un TDA puede ser implementado de varias formas sin que esto afecte a las aplicaciones que usen el mismo” es así que las cuatro formas de implementación que se detallaran a continuación no son las únicas.

4.1 Modelo Estático

Esta forma de implementación se refiere a que el uso del recurso memoria en la implementación de un TDA será simplemente definido antes de su ejecución lo cual necesariamente no lleva a usar tipos de datos básicos como ser Enteros, Reales, Cadenas, Vectores, Matrices.

4.2 Modelo Dinámico

En esta forma de implementación se prioriza el uso racional de la memoria lo cual significa que en la medida que sea necesario se solicitara espacio de memoria, para lo cual se sugiere repasar el manejo de punteros.

4.3 Modelo Simulado

A diferencia de las tres formas de implementación anteriormente citadas cabe aclarar que lo planteado en este punto es única y exclusivamente académico para tener una mejor comprensión sobre los punteros por lo que se propone crear un TDA denominado CSMemoria cuyo objetivo será simular el comportamiento de la memoria con el uso de vectores.

4.3 Modelo Persistente

El modelo persistente en el tiempo es la forma más usual que se tiene para hacer que la implementación de un TDA sea persistente en el tiempo, esto significa que su comportamiento podrá estar exento del uso exclusivo de la memoria trasladando las acciones de sus métodos a guardar la información en archivos.

https://youtu.be/CNvHXS0BPvQ

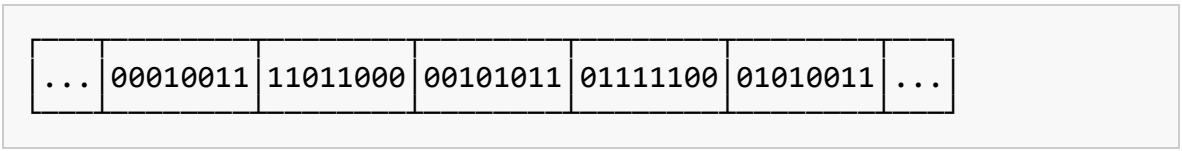
Considerando que existe bibliografía sobre cómo utilizar los modelos de implementación (persistente y estático y Dinámico) a continuación desarrollaremos el modelo simulado.

CONCEPTOS BÁSICOS

COMO FUNCIONA LA MEMORIA

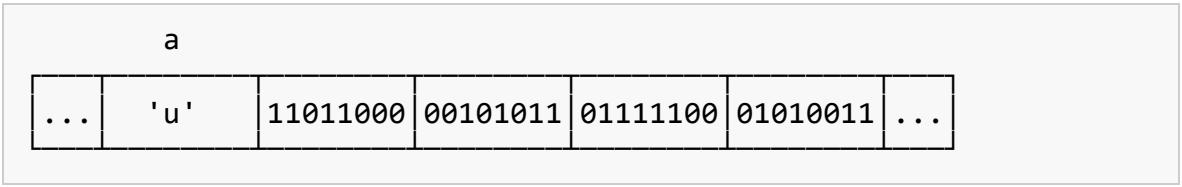
Todas las variables de un programa tienen asociado un lugar en la memoria del computador.

La memoria puede ser vista como un gran arreglo de bits. Un bit es la unidad básica de información que se puede representar en un computador, y puede tener los valores 0 o 1:

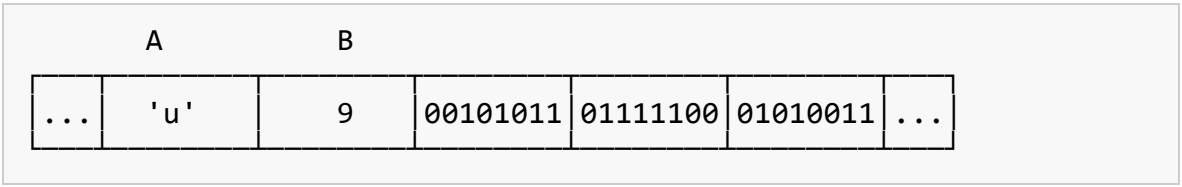


Tipos de datos

El tamaño de un valor de tipo char es de 1 byte, y el significado de los bits está determinado usando la codificación ASCII, es asi que si definimos la variable a de tipo char y luego a='u' la memoria tendría en binario el carácter 'u' según el siguiente gráfico.



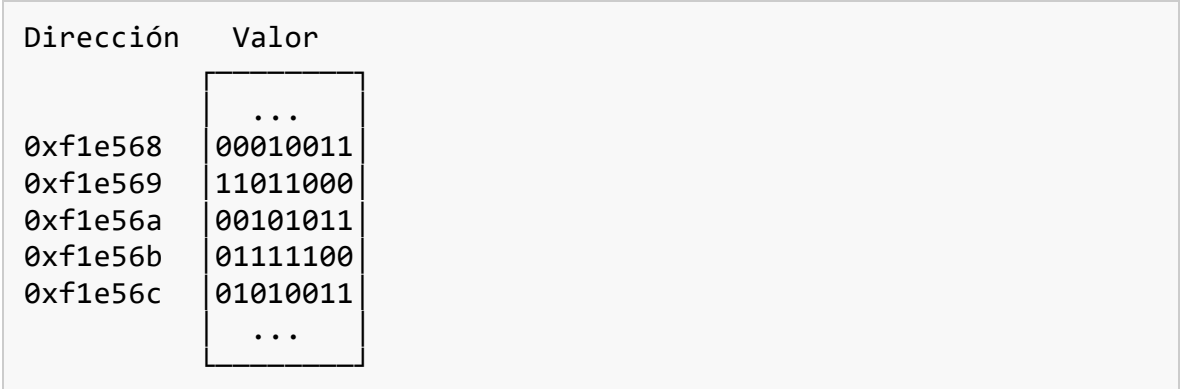
Los enteros son almacenados en su representación binaria. La manera más común de representar los enteros negativos es el complemento a dos, siendo asi y considerando que definimos un espacio de memoria llamado b de tipo entero y decimos b=9 tendríamos lo siguiente considerando que los valores de 'u' y 9 están en binario.



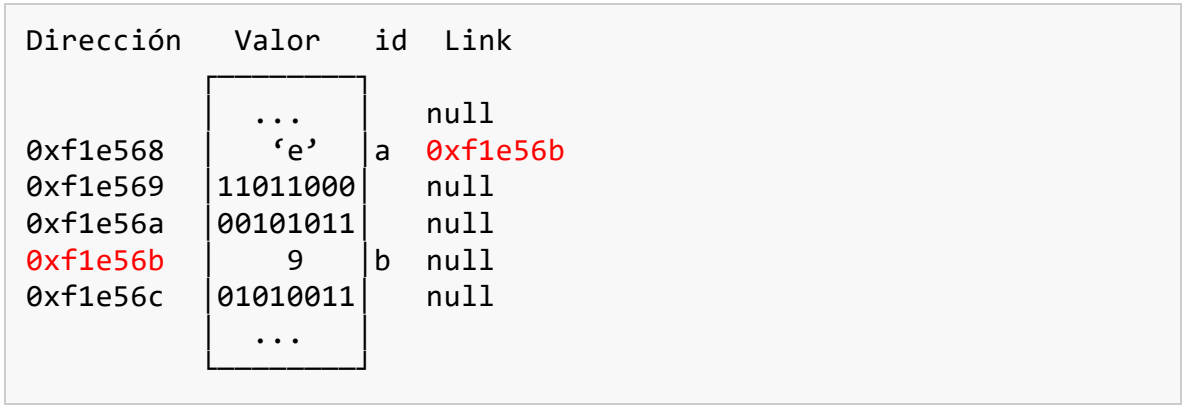
Direcciones de memoria

Todos los bytes en la memoria tienen una dirección, que no es más que un índice correlativo.

Por conveniencia, las direcciones de memoria suelen escribirse en notación hexadecimal, pero no hay que espantarse: se trata simplemente de un número entero.



Sin embargo si tenemos que guardar los datos de la variable Z que tiene dos campos, A de tipo carácter y B de tipo numérico, entonces nuestra grafica anterior debería quedar de la siguiente manera si decimos z.a='e' y z.b=9 solo para efectos únicos de implementar el TDA Smemoria.



Es asi que para la implementación de nuestro TDA Smemoria utilizaremos un vector donde en cada casilla(dirección) guarde Dato, id , link para los fines siguientes:

Dato : Sera el dato a guardar en la memoria
Id : Sera el nombre del espacio de memoria
Link : Sera la dirección de otro espacio de memoria.

(se establece que el simulador planteado solo manipulara datos enteros en valor, por lo que no se incrementara un campo para determinar el tipo de dato guardado)

DESCRIPCIÓN NO FORMAL DEL TDA SMEMORIA.

I. Elementos que conformarán estructura.

TDA SMemoria (VALORES Todos los números enteros , OPERACIONES, Crear, New_espacio, Delete_Espacio, Poner_dato, Obtener_dato, dir_libre, Espacio_Ocupado, Espacio_Disponible)

2. Descripción de las operaciones de la estructura.

Crear

Utilidad: Sirve para inicializar los espacios de memoria
Entrada: Espacios de memoria sin relación.
Salida: Espacios de memoria relacionados entre si.
Precondición: Ninguna.
Poscondición: Memoria lista para usarse.

New_Espacio

Utilidad: Solicitar N espacio de memoria con su debida identificación.
Entrada: Cadena con los identificadores requeridos.
Salida: Dirección de memoria a partir de donde se podrá guardar información para los identificadores referidos.
Precondición: Memoria Inicializada
Poscondición: Memoria disminuida en su capacidad de direcciones libres para trabajar

Delete_espacio

Utilidad: Liberar un espacio de memoria ocupado a fin de que retorne o forme parte de la memoria y pueda ser usado en otra actividad.
Entrada: Dirección inicial del espacio de memoria a ser liberado.
Salida: Ninguna

Precondición: La dirección de memoria a liberar debe existir y pertenecer a las direcciones en uso.
Poscondición: Memoria incrementada en su capacidad de direcciones libres para trabajar.

Poner_Dato
Utilidad: Asignar un valor a un espacio considerando su identificador a partir una dirección de memoria.
Entrada: Direccion inicial de memoria, Identificador de memoria, valor
Salida: Ninguna
Precondición: Dirección inicial de memoria e indicador valido.
Poscondición: Memoria modificada con el valor asignado a lugar que corresponde.

Obtener_Dato
Utilidad: Obtener un valor de un espacio de memoria considerando su identificador y partir una dirección de memoria.
Entrada: Direccion inicial de memoria, Identificador de memoria
Salida: Valor que se encuentra en el lugar especificado según el identificador
Precondición: Dirección inicial de memoria e indicador valido.
Poscondición: Ninguna

Espacio_Disponible
Utilidad: Determina cuantos espacios de memoria estan libres para poder trabajar.
Entrada: Ninguna.
Salida: Numero de espacios de memoria libres.
Precondición: Ninguna.
Poscondición: Ninguna.

Espacio_Ocupado
Utilidad: Determina cuantos espacios de memoria estan ocupados.
Entrada: Ninguna.
Salida: Numero de espacios de memoria ocupada.
Precondición: Ninguna.
Poscondición: Ninguna
.
dir_libre
Utilidad: Verifica si una dirección de memoria está libre.
Entrada: Direccion de Memoria
Salida: Valor booleano / verdadero o falso.
Precondición: La dirección debe ser válida.
Poscondición: Ninguna

IMPLEMENTACIÓN DEL TDA SMEMORIA

Constantes MAX = 20
NULO =-1
Definiendo Tipos de Datos.
NodoM
Dato TipoDato (Entero)
id cadena(12)
Link Direccion de Memoria(Entero)
Fin tipo
clase CSMemoria
Atributos
MEM arreglo(MAX) de tipo NodoM
libre Direccion de memoria (entero)
Métodos
Constructor // crear
direccion new_espacio(cadena)
Delete_espacio(dir)
poner_dato(dir , cadena_id, valor)
TipoDato obtenerDato(dir, lugar)
entero Espacio_Disponible()
entero Espacio_ocupado()
booleano dir_libre(dir)
fin def clase

CONSTRUCTOR CSMEMORIA::CREAR() INICIO PARA CADA **I** DESDE **0** HASTA **max** MEM[I].LINK = I+1 FIN PARA MEM[max].LINK = **-1** **LIBRE = 0** FIN https://youtu.be/Y-xP9ql8phQ	**CSMEMORIA::DELETE_ESPACIO(DIR)** INICIO X = DIR MIENTRAS **MEM[X].LINK<>-1** **X= MEM [X].LINK** FIN MIENTRAS **MEM [X].LINK=LIBRE** **LIBRE=DIR** FIN https://youtu.be/M6pG8zwvrNs

DIRECCION CSMEMORIA::NEW_ESPACIO (cadena)
// cadena 'a,b,c'
INICIO
cant = **Numero_Ids(Cadena)**
DIR = LIBRE
D = LIBRE
PARA CADA I = 1 HASTA CANT -1
MEM[D].**ID=Obtener_Id(cadena,i)**
D = MEM [D].LINK
FIN PARA
LIBRE=MEM **[D].LINK**
MEM **[D].LINK = -1**
MEM[D].ID=Obtener_Id(cadena, cant)
// **RETORNAR DIR**
FIN
https://youtu.be/nYtw5yHD-60
	NUMERO CSMEMORIA::ESPACIO_DISPONIBLE( )
	// Cantidad de memoria disponible
	INICIO
	X = LIBRE
	C = 0 // CONTADOR
	MIENTRAS X <> -1
	C=C+1
	X=MEM[X].LINK
	FIN MIENTRAS
	// RETORNAR C
	FIN
	
	https://youtu.be/Njrcpg-WntM

BOOLEANO CSMEMORIA::dir_libre(DIR)
// si DIR libre Verdadero sino Falso

```
INICIO
   X = LIBRE    C = FALSO    // BANDERA
   MIENTRAS  ( X <> -1)  Y (C = FALSO)
        SI X = DIR ENTONCES C = VERDADERO
        FIN SI
        X=MEM[ X ].LINK
   FIN MIENTRAS
  RETORNAR C
FIN
```

https://youtu.be/GTZIvpCCWDk

TipoDato SMEMORIA::OBTENER_DATO
(DIR , cadena_id)

```
INICIO
    Z = DIR   EX=FALSO
    Eliminar_flecha(cadena_id)
    MIENTRAS (Z<>NULO )
         SI MEM[Z].ID = cadena_id ENTONCES
               // RETORNAR MEM[Z].DATO
        FIN SI
       Z=MEM[Z].LINK
    FIN MIENTRAS
 FIN
```

https://youtu.be/x9G61F1HZYA

CSMEMORIA::PONER_DATO
(DIR, cadena_id, VALOR)
// formato de cadena_id –'→nom_id'

```
INICIO
    Z = DIR
    Eliminar_flecha(cadena_id)
    MIENTRAS Z<>NULO
        SI MEM[Z].ID = cadena_id ENTONCES
            MEM[Z].DATO=VALOR
        FIN SI
       Z=MEM[Z].LINK
    FIN MIENTRAS
FIN
```

https://youtu.be/MIRYZliizdY

NUMERO CSMEMORIA::ESPACIO_OCUPADO()

```
// Cantidad de Memoria Ocupada
INICIO
   C= (max+1) – Espacio_Disponible
   RETORNAR C
FIN
```

https://youtu.be/FpMYLb6IZlo