Universidad de las Ciencias Informáticas

Facultad # 9

# Documentación de la Biblioteca de Estructuras de Datos Avanzadas (Listas, Pilas, Colas, Tablas Hash y DCEL).

**Autor:**

- Yulaine Arias Guerra.

**Tutor:**

- Lic. José Ángel Lago Graverán.

**Co-Tutor(es):**

- Ing. Alexey Díaz Domínguez.
- Yancy Martínez Pérez.
- Lic. Yusnier Valle Martínez.

Ciudad de la Habana, Julio del 2008.

"Año 50 de la Revolución".

## "Nunca consideres el estudio como una obligación sino como una oportunidad para penetrar en el bello y maravilloso mundo del saber…"

**Albert Einstein**

# ÍNDICE

INTRODUCCION ............................................................ 4

CAPÍTULO 1 Estructura de dato: Lista. .......... 7
1.1 Lista utilizando arreglos. .......... 9
1.2 Lista Enlazada. .......... 12
1.3 Lista Simplemente Enlazada. .......... 13
1.3.1 Lista Circular Simplemente Enlazada. .......... 16
1.4 Lista Doblemente Enlazada. .......... 18
1.4.1 Lista Circular Doblemente Enlazada. .......... 22
1.5 Observaciones Generales sobre el TDA Lista. .......... 23
CAPÍTULO 2 Estructura de datos: Pila. .......... 26
2.1 Pila utilizando arreglos. .......... 28
2.2 Pila utilizando listas enlazadas. .......... 30
2.3 Observaciones Generales sobre el TDA Pila. .......... 32
CAPÍTULO 3 Estructura de dato: Cola. .......... 34
3.1 Cola de prioridad. .......... 36
3.1.1 Cola de prioridad utilizando arreglos. .......... 39
3.1.2 Cola de prioridad utilizando listas enlazadas. .......... 41
3.2 Cola de amigos. .......... 43
3.2.1 Cola de Amigos utilizando arreglos. .......... 44
3.2.2 Cola de Amigos utilizando listas enlazadas. .......... 44
3.3 Observaciones Generales sobre los TDA Colas. .......... 45
CAPÍTULO 4 Estructura de dato: Tabla Hash. .......... 47
4.1 Resolución de colisiones por dispersión abierta. .......... 50
4.2 Resolución de colisiones por dispersión cerrada. .......... 52
4.3 Observaciones Generales sobre la estructura de datos: Tabla Hash. .......... 55
4.3.1 Función de dispersión. .......... 55
4.3.2 Desbordamiento de la tabla. .......... 60
CAPÍTULO 5 Estructura de dato: DCEL. .......... 63
5.1- Observaciones Generales sobre la estructura de datos DCEL. .......... 66
COCLUSIONES. .......... 68
GLOSARIO DE Términos Y SIGLAS .......... 69
REFERENCIAS BIBLIOGRAFICAS. .......... 71

# INDICE DE FIGURAS

**Figura 1.1** Representación gráfica de un Nodo............................................................12
**Figura 1.2** Representación gráfica de una Lista Simplemente Enlazada. ....................13
**Figura 1.3** Representación gráfica de una Lista Circular Simplemente Enlazada. ........17
**Figura 1.4** Representación gráfica de una Lista Doblemente Enlazada. .......................18
**Figura 1.5** Representación gráfica de una Lista Circular Doblemente Enlazada. ..........22

**Figura 2.1** Representación gráfica de una Pila. ..........................................................27
**Figura 2.2** Representación gráfica de una Pila utilizando arreglos..............................28
**Figura 2.3** Representación gráfica de una Pila con enlazables....................................30

**Figura 3.1** Representación gráfica de una Cola...........................................................35
**Figura 3.2** Representación gráfica de una Cola de Prioridad.......................................36
**Figura 3.3** Representación gráfica de varias Colas con prioridades. ............................37

**Figura 4.1** Representación gráfica de una Tabla Hash. ...............................................48
**Figura 4.2** Representación gráfica de una colisión. .....................................................49
**Figura 4.3** Representación gráfica de la resolución de colisiones por dispersión abierta en una tabla hash.............................................................................................51
**Figura 4.4** Representación gráfica de la resolución de colisiones por dispersión cerrada en una tabla hash............................................................................................53
**Figura 4.5** Diagrama de la estructura de datos utilizada en la dispersión extensible....62

**Figura 5.1** Representación gráfica de DCEL...............................................................64

# INTRODUCCION.

Cuando se implementa una clase es necesario preocuparse por elegir una representación para los objetos de esa clase. Esta representación consiste en un conjunto de variables de instancias que almacenarán todos los datos que se requiere memorizar para poder realizar las operaciones, además es necesario preocuparse por implementar las operaciones de las clases.

Existen patrones de organización para las variables de instancia, estos patrones se denominan *Estructuras de Datos* y una de las aplicaciones más interesantes y potentes de la memoria dinámica y los punteros son precisamente estas.

No existe una estructura de datos que sirva para todos los propósitos, por tal motivo es importante conocer las ventajas y desventajas de cada una de ellas para en un momento dado conocer cual es la más óptima para darle solución a un determinado problema.

Antes de explorar las diferentes estructuras de datos y sus algoritmos específicos, es necesario examinar una cuestión básica: ¿Qué es una estructura de datos? El conocimiento de este concepto ayudará a entender mejor este documento.

En programación, una *estructura de datos* es una forma de organizar un conjunto de datos elementales[1] con el objetivo de facilitar la manipulación de estos datos como un todo o individualmente.

Los datos de tipo estándar pueden ser organizados en diferentes estructuras de datos: estáticas y dinámicas.

- *Estructura de Datos Estáticas:*

  Son aquellas en las que el espacio ocupado en la memoria de la computadora se define en tiempo de compilación y no puede ser modificado durante la ejecución del programa. Corresponden a este tipo los arreglos y los registros.

[1] un dato elemental es la mínima información que se tiene en el sistema.

- *Estructuras de Datos Dinámicas:*

Son aquellas en las que el espacio ocupado en la memoria de la computadora puede ser modificado en tiempo de ejecución. Corresponden a este tipo las listas, las pilas, las colas, etc.

Cada estructura de datos dinámica posee un comportamiento específico y como tal una lógica diferente para operarla, tanto es así que no es lo mismo eliminar un nodo de una lista simplemente enlazada que uno en una de doble referencia, debido a que la lógica cambia aunque la operación sea la misma.

Las estructuras de datos tienen en común que un identificador y un nombre pueden representar a múltiples datos individuales.

Una estructura de datos define además la organización e interrelación de los elementos y un conjunto de operaciones que se pueden realizar sobre ellos. Las operaciones básicas son:

- **Adiciona**r, adiciona un nuevo valor a la estructura.
- **Eliminar**, borra un valor de la estructura.
- **Buscar**, encuentra un determinado valor en la estructura para realizar una operación con este valor, en forma *secuencial* o *binaria* (siempre y cuando los datos estén ordenados).

Otras operaciones que se pueden realizar son:

- **Ordenar**, ordena a los elementos pertenecientes a la estructura.
- **Concatenar**, dadas dos estructuras originar una nueva ordenada y que contenga a las concatenadas.

Para la realización de las operaciones cada estructura ofrece ventajas y desventajas en relación a su simplicidad y eficiencia. 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, así como la naturaleza de los datos que se almacenarán.

Este documento de ayuda explora cinco de esas estructuras de datos en algunas de sus variantes de implementación:

- Listas:
  - Lista con Arreglo.
  - Lista Simplemente Enlazada.
  - Lista Circular Simplemente Enlazada.
  - Lista Doblemente Enlazada.
  - Lista Circular Doblemente Enlazada.
- Pilas:
  - Pila con Arreglo.
  - Pila con Listas Enlazables.
- Colas:
  - Cola de Prioridad:
    - Cola de Prioridad con Arreglo.
    - Cola de Prioridad con Listas Enlazables.
  - Cola de Amigos:
    - Cola de Amigos con Arreglo.
    - Cola de Amigos con Listas Enlazables.
- Tablas Hash:
  - Tabla Hash Abierta.
  - Tabla Hash Cerrada.
- DCEL.

Seleccionar un tipo de estructura de datos idónea para una aplicación dependerá esencialmente de la naturaleza del problema a resolver y en menor medida del lenguaje de programación.

La elección adecuada de las estructuras de datos y el algoritmo empleado permite obtener un diseño eficiente, tanto en recursos ocupados en la memoria de la computadora como en el tiempo de ejecución.

# CAPÍTULO 1

# ESTRUCTURA DE DATOS: LISTA.

En este capítulo se da a conocer el concepto de la estructura de datos Lista y de algunas de sus formas de implementación, conjuntamente se reflejan las ventajas y las desventajas de algunas, así como las aplicaciones que tienen en la vida práctica, además de mostrar el seudocódigo. Lo mencionado anteriormente permite tener un mejor conocimiento de este tipo de estructura de datos y saber en un problema determinado cual es la más óptima para darle solución al mismo.

**¿Qué es una lista?**

Una lista se define como una n-tupla de elementos (donde ***li*** es el i-ésimo elemento de la lista) ordenados de forma consecutiva, o sea, el elemento ***li*** precede al elemento ***li +1***: ***L= (l1, l2,…, ln).*** Si la lista contiene cero elementos se denomina lista vacía. (1)

Operaciones básicas del TDA Lista:

a. **Vacía ( ),** devuelve verdadero si la longitud de la lista es cero. No se modifica la lista.
b. **Longitud ( )**, devuelve **|*L*|*,*** la longitud de la lista (cantidad de elementos). No se modifica la lista.
c. **Obtener (i)**, devuelve el i-ésimo elemento de la lista (el que se encuentra en la posición ***i***), si la posición ***i*** no existe se dispara una excepción. No se modifica la lista.

d. **Adicionar(x)**, adiciona el elemento ***x*** en la cola de la lista, haciendo que la longitud de la lista se incremente en uno. Si la operación no tiene éxito, se dispara una excepción.
e. **Insertar(x, i)**, inserta el elemento *x* en la posición ***i*,** haciendo que los elementos ***li, li+1,…, ln*** pasen a ser los elementos ***li+1, li+2,…, ln+1*** y se incrementa en uno la longitud de la lista. Si la operación no tiene éxito, se dispara una excepción.
f. **Eliminar(i),** elimina el elemento almacenado en la posición ***i*** de la lista, haciendo que los elementos ***li+1, li+2,…,ln*** pasen a ser los elementos ***li, li+1, …, ln-1***, esta operación disminuye en uno la longitud de la lista. Si la posición no existe se dispara una excepción.

**Ventajas:**

» Las listas son dinámicas, es decir, se pueden almacenar en ellas tantos elementos como se necesiten, siempre y cuando haya espacio suficiente en la memoria de la computadora.

» Al insertar un elemento en la lista, la operación tiene un tiempo constante independientemente de la posición en la que se inserte, solo se debe crear el nodo y modificar los enlaces de los mismos.

» Al eliminar un elemento sucede lo mismo que se mencionó en el punto anterior.

**Desventajas:**

» El acceso a un elemento es más lento, debido a que la información no está en posiciones contiguas en la memoria de la computadora, por lo que no se puede acceder a un elemento con base en su posición como se hace en los arreglos.

**Aplicaciones:**

» Las listas son comunes en la vida diaria: listas de alumnos, listas de clientes, listas de espera, listas de distribución de correo, etc.

- Las listas son estructuras de datos muy útiles para los casos en los que se quiere almacenar información de la que no se conoce su tamaño con antelación.
- También son valiosas para las situaciones en las que el volumen de datos se puede incrementar o decrementar dinámicamente durante la ejecución del programa.
- Cuando se aplican restricciones de acceso a las listas, se tienen a las pilas y a las colas, que son listas especiales.
- Puede usarse para implementar una amplia variedad de TDA, es decir, el TDA Lista, sirve a menudo como una pieza básica en la construcción de los TDA más complicados, como es el caso de las tablas hash, los árboles, los grafos, etc.

## 1.1 Lista utilizando arreglos.

**Definición 1.1:**

Un TDA Lista puede utilizar como estructura de datos a los arreglos lineales, esta clase se deriva de la clase Lista. Los arreglos lineales son convenientes fundamentalmente por la simplicidad de su uso, además el tiempo de acceso a los elementos individuales de un arreglo es fijo para todas las operaciones, lo que resulta muy eficiente. (1)

Sin embargo, las operaciones de inserción y borrado de los elementos en los arreglos son ineficientes, puesto que para insertar un elemento en la parte media del arreglo es necesario mover todos los elementos que se encuentren detrás de él para hacer espacio y al borrar un elemento es preciso mover todos los elementos para ocupar el espacio desocupado.

**Ventajas:**

- Todos los elementos de la lista se almacenan en posiciones de memoria consecutivas, pero se habla de *disposición secuencial* en la memoria de la computadora. Con esta disposición se accede a cualquier elemento de la

estructura de datos en tiempo constante, o sea, permiten un acceso aleatorio.

» Son convenientes por una razón fundamental, la simplicidad de su uso.

» Además el tiempo de acceso a los elementos individuales de un arreglo es fijo para todas las operaciones, lo cual resulta muy eficiente.

**Desventajas:**

» Al asignar el arreglo en tiempo de compilación debe establecerse un límite a *priori* sobre el número de elementos que pueden ser almacenados en las listas.

» Para inserciones y eliminaciones frecuentes hay que hacer corrimientos costosos.

» Otra limitación de estas listas que emplean arreglos es que se llenarán ó necesitarán ser redimensionadas, lo cual es una costosa operación que incluso puede no ser posible si la memoria de la computadora se encuentra fragmentada.

» Al insertar un elemento en el arreglo, la operación tiene un tiempo constante independientemente de la posición en la que se inserte, solo se debe crear el nodo y modificar los enlaces. Esto no es así en los arreglos, ya que si el elemento se inserta al inicio o en el medio, se posee un tiempo lineal debido a que se tienen que mover todos los elementos que se encuentran a la derecha de la posición donde se va a insertar y después insertar el elemento en dicha posición; solo al insertar al final del arreglo se obtienen tiempos constantes.

» Al eliminar un elemento sucede lo mismo que se mencionó en el punto anterior.

**Seudocódigo:**

```
*****************************************************
*           Lista implementada utilizando un Arreglo            *
*****************************************************
```

```
Obtener( entero pos)
Inicio
    si ( (pos >= 0 ) y ( pos < longitud) ) entonces
        retornar elementos[pos]
    sino
        lanzar excepción
Fin
.......................................................................................................
Adicionar(Tipo x)
Inicio
    si (longitud < MAX_ELEMENTOS)  entonces
        elementos[ longitud + 1] ← x
        longitud ← longitud + 1
    sino
        lanzar excepción
Fin
.......................................................................................................
Insertar(Tipo x, entero pos)
Inicio
    si ( (pos < 0) o (pos ≥ longitud) ) entonces
        Error("POSICION NO VALIDA")
    sino
        inicio
           mientras ( (x > 0) y (x< pos-1) ):
               auxiliar.Adicionar (arreglo[x])
           auxiliar.Adicionar(elemento)
           mientras ( (y > pos – 1)  y  (x < longitud) )
               auxiliar.Adicionar(arreglo[y])
            arreglo ← auxiliar.arreglo
        fin
Fin
.......................................................................................................
Eliminar (entero pos)
Inicio
```

```
    para  i ← pos hasta  i ← longitud -1
        arreglo[i] ← arreglo[i + 1]
Fin
```

## 1.2 Lista Enlazada.

**Definición 1.2:**

Una lista enlazada es una secuencia de nodos enlazados donde el orden de los elementos está determinado por un campo de enlace (referencia) explícito en cada elemento. Esta definición remite al concepto de nodo.

Un nodo se compone de un campo que contiene el tipo de dato de los elementos de la lista y por uno o más campos que son referencias a otros nodos. (2)

**Figura 1.1** Representación gráfica de un Nodo.

**Ventajas:**

- Una lista enlazada es un TDA dinámico, su tamaño puede cambiar durante la ejecución del programa y los elementos se pueden insertar indefinidamente.
- No es preciso conocer la cantidad de elementos en tiempo de compilación.
- Ni las inserciones ni las eliminaciones implican realizar corrimientos de los elementos de la lista. Al insertar un elemento en la lista, la operación tiene un tiempo constante independientemente de la posición en la que se inserte, solo se debe crear el nodo y modificar los enlaces.

**Desventajas:**

- No permite el acceso directo a un elemento arbitrario de la lista. Para acceder al *i-ésimo* elemento, se debe recorrer la lista, comenzando por el

primer nodo, hasta llegar al elemento deseado, es decir, permiten solamente un acceso secuencial a los elementos.

### 1.3 Lista Simplemente Enlazada.

**Definición 1.3:**

Una lista simplemente enlazada se compone por nodos de enlace simple, es decir, por nodos que tienen solamente un campo *Dirección*. Un nodo sólo esta enlazado con el nodo que le sigue. (2)

Cabeza

datos | datos | datos

**Figura 1.2** Representación gráfica de una Lista Simplemente Enlazada.

**Desventajas:**

- Las listas simplemente enlazadas solo pueden ser recorridas en una dirección. Esto hace que las listas sean inadecuadas para aquellos casos en los que es útil buscar un elemento por su índice rápidamente, como el *heapsort.*
- Otra desventaja de las listas simplemente enlazadas es el almacenamiento extra necesario para las referencias, que a menudo las hacen poco prácticas para listas de pequeños datos como caracteres o valores booleanos.

**Seudocódigo:**

```
***************************************************
*                Lista Simplemente Enlazada                *
***************************************************
Longitud() : Entero
```

```
Inicio
    longitud ← 0
    cursor ← _cabeza
    mientras ( cursor ≠ Nulo )
    inicio
        longitud ← longitud +1
        cursor ← cursor. siguiente
    fin
    retornar longitud
Fin
..........................................................................................................
Obtener(entero pos)
Inicio
    si ( (pos < 1) o (pos > longitud) ):
        Error ("POSICION FUERA DE RANGO")
    sino
        inicio
            cursor  ←  NodoSE()
            cursor  ← primero
            para i  ← 1, hasta i  ←  pos:
                cursor ← cursor. siguiente
            retornar cursor.GetDato()
        fin
Fin
..........................................................................................................
Adicionar (Tipo x)
Inicio
    nodo ← nuevo NodoSE(x, Nulo)
    si (cabeza == Nulo)
        cabeza ← nodo
    sino
    inicio
        cursor ← cabeza
        mientras (cursor. siguiente ≠ Nulo):
```

```
            cursor ← cursor. siguiente
        cursor. siguiente ← nodo
    fin
Fin
.................................................................................................
Insertar (Tipo x, entero pos)
Inicio
    si ( (pos < 0) o (pos ≥ longitud) ) entonces
        Error(“POSICION NO VALIDA”)
    nodo ← nuevo NodoSE(x, Nulo)
    si (pos ← 0) entonces
    inicio
        nodo. siguiente ← cabeza
        cabeza ← nodo
    fin
    sino entonces
        cursor ← cabeza
        pos_cursor ← 0
        mientras ( (cursor ≠ Nulo) y (pos_cursor+1  ≠ pos) )
        inicio
            pos_cursor ← pos_cursor + 1
            cursor ← cursor. siguiente
        fin
        nodo. siguiente ← cursor. siguiente
        cursor. siguiente ← nodo
    fin
Fin
.................................................................................................
Eliminar (entero pos)
Inicio
    si (Vacía()) entonces
        Error(“LISTA VACIA”)
    si ( (pos < 0) o (pos ≥ longitud) ) entonces
        Error(“POSICION NO VALIDA”)
```

```
        cursor ← cabeza
    si (pos ← 0) entonces
    inicio
        cabeza ← cursor. siguiente
        destruir cursor
    fin
    sino entonces
    inicio
        anterior ← cabeza
        pos_cursor ← 0
        mientras ( (cursor ≠ Nulo) y (pos_cursor ≠ pos) )
        inicio
            anterior ← cursor
            cursor ← cursor. siguiente
            pos_cursor ← pos_cursor + 1
        fin
        anterior. siguiente ← cursor. siguiente
        cursor. siguiente ← Nulo
        destruir cursor
    fin
Fin
```

### 1.3.1 Lista Circular Simplemente Enlazada.

**Definición 1.4:**

Una lista circular simplemente enlazada es una lista lineal en la que el último elemento se enlaza con el primero. Cada nodo tiene un enlace similar al de las listas simplemente enlazadas, excepto que el siguiente nodo del último apunta al primero.

En las listas circulares no puede hablarse ni de "primero" ni de "último", porque cualquier nodo puede ser el nodo de entrada y de salida. Al igual que en una lista simplemente enlazada, los nuevos nodos pueden ser solo eficientemente insertados

después de uno que se tenga referenciado, por esta razón, es usual quedarse con una referencia solamente al último elemento en una lista circular simplemente enlazada, esto permite rápidas inserciones al principio y accesos al primer nodo desde el puntero del último nodo. Además es posible acceder a cualquier elemento de la lista desde cualquier punto dado. La figura 1.3 muestra la representación gráfica de una Lista Circular Simplemente Enlazada.

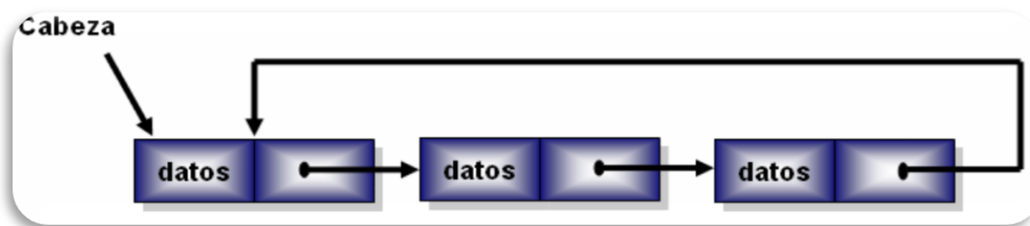


**Figura 1.3** Representación gráfica de una Lista Circular Simplemente Enlazada.

**Ventajas:**

- Son útiles para describir estructuras circulares.
- Pueden recorrer la lista desde cualquier punto.
- También permiten el acceso rápido al primer y último elemento por medio de un puntero simple.
- No existe ningún elemento que apunte a Nulo.
- Se integra una estructura de tipo anillo.
- Solo hay una cabeza. La cabeza siempre será el siguiente enlace para algún nodo.

**Desventaja:**

- Se pueden llegar a crear recorridos en bucles infinitos.

**Aplicaciones:**

- Este tipo de listas es el más usado para dirigir buffers, para "ingerir" datos y para visitar todos los nodos de una lista a partir de uno dado.

### 1.4 Lista Doblemente Enlazada.

**Definición 1.5:**

Una lista doblemente enlazada es una lista lineal que se compone por nodos de enlace doble, es decir, por nodos que tienen dos campos: *Dirección1* y *Dirección2*. Un nodo está enlazado con el nodo que le sigue y con el nodo que inmediatamente le antecede. Esto permite recorrer la lista en cualquier dirección. (2)

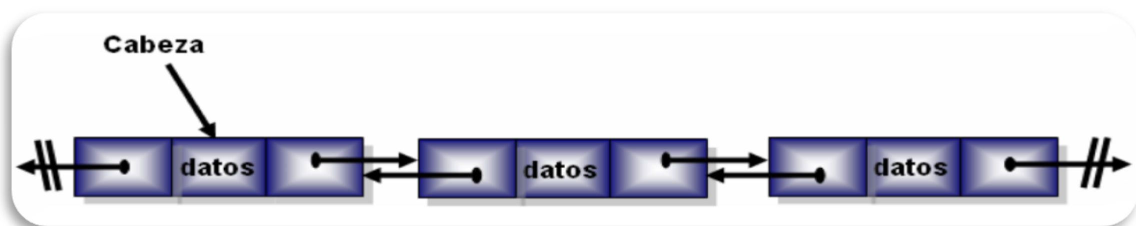


**Figura 1.4** Representación gráfica de una Lista Doblemente Enlazada.

**Ventajas:**

- » Una ventaja que tienen es que pueden recorrerse en ambos sentidos, ya sea para efectuar la operación de insertar, actualizar o borrar cualquier elemento.
- » Otra ventaja de las listas doblemente enlazadas es que se puede usar un puntero a la celda que contiene el *i-ésimo* elemento de una lista para representar la posición ***i***, mejor que usar el puntero a la celda anterior aunque lógicamente, también es posible la implementación similar a la expuesta en las listas simples haciendo uso de la cabecera.
- » La otra ventaja es que las búsquedas son algo más rápidas puesto que no hace falta hacer referencia al elemento anterior.(3)

**Desventajas:**

- » El único precio que se paga por estas características es la presencia de un puntero adicional en cada celda y consecuentemente procedimientos algo más largos para algunas de las operaciones básicas de las listas, es decir, ocupan más espacio en memoria por nodo que una lista simple y sus operaciones básicas resultan más costosas.(3)

**Seudocódigo:**

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                Lista Doblemente Enlazada                                     *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Longitud() : Entero
Inicio
    longitud ← 0
    cursor ← _cabeza
    mientras ( cursor ≠ Nulo )
    inicio
        longitud ← longitud +1
        cursor ← cursor. siguiente
    fin
retornar longitud
Fin
..........................................................................................................
Obtener(entero pos)
Inicio
    si ( (pos < 1) o (pos > longitud) )
        Error("POSICION NO VALIDA")
    sino:
        inicio
            cursor ← NodoDE
            cursor ← primero
            para i ← 1, hasta i ← pos:
                cursor ← cursor. siguiente
            retornar cursor.GetDato()
        fin
Fin
..........................................................................................................
Adicionar(Tipo x)
Inicio
    nodo ← nuevo NodoDE(x, Nulo, Nulo)
```

```
    si (cabeza ← Nulo)
        cabeza ← nodo
    sino
    inicio
        cursor ← cabeza
        mientras ( cursor. siguiente ≠ Nulo )
            cursor ← cursor. siguiente
        cursor. siguiente ← nodo
        nodo. anterior ← cursor
    fin
Fin
.............................................................................................................
Insertar(Tipo x, entero pos)
Inicio
    si ( (pos < 0) o (pos ≥ longitud) ) entonces
        Error("POSICION NO VALIDA")
        nodo ← nuevo NodoDE(x, Nulo, Nulo)
    si (pos ← 0) entonces
    inicio
        si (cabeza ≠ Nulo) entonces
        inicio
            cabeza. anterior ← nodo
            nodo. siguiente ← cabeza
        fin
        cabeza ← nodo
    fin
    sino
    inicio
        cursor ← cabeza
        pos_cursor ← 0
        mientras ( (cursor. siguiente ≠ Nulo) y (pos > pos_cursor) )
        inicio
            pos_cursor ← pos_cursor + 1
            cursor ← cursor. siguiente
```

```
        fin
        nodo. siguiente ← cursor
        nodo. anterior ← cursor. anterior
        cursor.anterior.siguiente ← nodo
        cursor. anterior ← nodo
Fin
.......................................................................................................
Eliminar (entero pos)
inicio
    si (Vacía()) entonces
        Error("LISTA VACIA")
    si ( (pos < 0) o (pos ≥ longitud) ) entonces
        Error("POSICION NO VALIDA")
        cursor ← cabeza
    si (pos ← 0) entonces
    inicio
        cabeza ← cursor. siguiente
        destruir cursor
    fin
    sino entonces
    inicio
        anterior ← cabeza
        pos_cursor ← 0
        mientras ( (cursor ≠ Nulo) y (pos_cursor ≠ pos) )
        inicio
            anterior ← cursor
            cursor ← cursor. siguiente
            pos_cursor ← pos_cursor + 1
        fin
        anterior. siguiente ← cursor. siguiente
        cursor. siguiente ← Nulo
        destruir cursor
    fin
fin
```

### 1.4.1 Lista Circular Doblemente Enlazada.

**Definición 1.6:**

Una lista circular doblemente enlazada es una lista lineal en la que cada elemento tiene dos enlaces, similares a los de la lista doblemente enlazada, excepto que el enlace anterior del primer nodo apunta al último y el enlace siguiente del último nodo, apunta al primero.

Al igual que una lista doblemente enlazada, en la circular doble las inserciones y eliminaciones pueden realizarse desde cualquier punto con acceso a algún nodo cercano. Aunque estructuralmente una lista circular doblemente enlazada no tiene ni principio ni final. Un puntero de acceso externo puede establecer el nodo apuntado que está en la cabeza o al nodo cola y así mantener el orden como en una lista doblemente enlazada. La figura 1.5 muestra la representación gráfica de una Lista Circular Doblemente Enlazada.

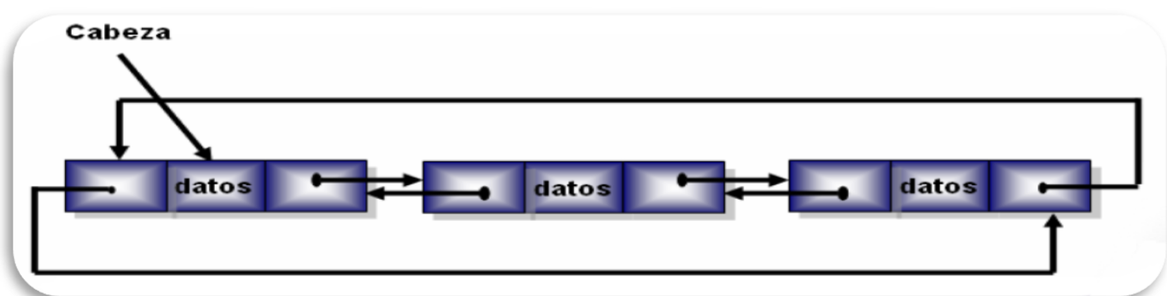


**Figura 1.5** Representación gráfica de una Lista Circular Doblemente Enlazada.

**Ventajas:**

- Son útiles para describir estructuras circulares.
- Pueden recorrer la lista desde cualquier punto.
- También permiten el acceso rápido al primer y último elemento por medio de un puntero simple.
- No existe ningún elemento que apunte a Nulo.
- Se integra una estructura de tipo anillo.
- Solo hay una cabeza. La cabeza siempre será el siguiente enlace para algún nodo.

**Desventaja:**

- Se pueden llegar a crear recorridos en bucles infinitos.

## 1.5 Observaciones Generales sobre el TDA Lista.

**Listas Enlazadas vs. Arreglos.**

En las implementaciones de las listas utilizando arreglos los métodos son sencillos, la manipulación en sí del arreglo lo es, pero una limitante importante es que aún siendo dinámico el arreglo y pueda crecer según se le necesite, está compuesto por un número definido a *priori*, que impone restricciones sobre el tamaño de la lista.

Otra limitante relacionada con esto, es que al hacer crecer el arreglo no se puede hacer sobre él mismo, esto implica que al cambiarle el tamaño sería necesario copiar sus elementos en un arreglo temporal, lo que introduce un elemento de ineficiencia en la solución que las utilice.

Los arreglos lineales son convenientes por una razón fundamental, la simplicidad de su uso, además el tiempo de acceso a los elementos individuales de un arreglo es fijo para todas las operaciones, lo que resulta muy eficiente. La clase derivada se llamaría Lista basada en Arreglos Estáticos.

| | Arreglo | Lista Enlazada |
|---|---|---|
| Indexado | O(1) | O(*n*) |
| Inserción / Eliminación al final | O(1) | O(1) or O(*n*)[2] |
| Inserción / Eliminación en la mitad | O(*n*) | O(1) |
| Persistencia | No | Simples sí |
| Localización | Buena | Mala |

Las listas enlazadas poseen muchas ventajas sobre los arreglos. Los elementos se pueden insertar en una lista indefinidamente mientras que un arreglo tarde o temprano se llenará ó necesitará ser redimensionado, una costosa operación que incluso puede no ser posible si la memoria de la computadora se encuentra fragmentada.

En algunos casos se pueden lograr ahorros de la memoria de la computadora almacenando la misma “cola” de elementos entre dos o más listas, es decir, la lista concluye en la misma secuencia de elementos. De este modo, se pueden añadir nuevos elementos al frente de la lista manteniendo una referencia tanto al nuevo como a los viejos elementos, esto es un ejemplo simple de una estructura de datos persistente.

Por otra parte, los arreglos permiten accesos aleatorios mientras que las listas enlazadas solo permiten acceso secuencial a los elementos. Las listas simplemente enlazadas solo pueden ser recorridas en una dirección. Esto hace que las listas sean inadecuadas para aquellos casos en los que es útil buscar un elemento por su índice rápidamente, como el *heapsort*. El acceso secuencial en los arreglos también es más rápido que en las listas enlazadas.

Otra desventaja de las listas enlazadas es el almacenamiento extra necesario para las referencias, que a menudo las hacen poco prácticas para listas de pequeños datos como caracteres o valores booleanos.

También puede resultar lento y abusivo el asignar memoria para cada nuevo elemento. Existe una variedad de listas enlazadas que contemplan los problemas anteriores para resolver los mismos.

Un buen ejemplo que muestra los *pros* y los *contras* del uso de los arreglos sobre las listas enlazadas es la implementación de un programa que resuelva el problema de *Josephus*. Este problema consiste en un grupo de personas dispuestas en forma de círculo. Se empieza a partir de una persona predeterminada y se cuenta ***n*** veces, la persona *n-ésima* se saca del círculo y se vuelve a cerrar el grupo. Este proceso se repite hasta que queda una sola persona, que es la que gana.

Este ejemplo muestra las fuerzas y debilidades de las listas enlazadas frente a los arreglos, ya que viendo a la gente como nodos conectados entre sí en una lista circular se observa como es más fácil suprimir estos nodos. Sin embargo, se ve como la lista perderá utilidad cuando haya que encontrar a la siguiente persona a borrar. Por otro lado, en un arreglo el suprimir los nodos será costoso ya que no se puede quitar un elemento sin reorganizar el resto. Pero en la búsqueda de la *n-*

*ésima* persona tan sólo basta con indicar el índice **n** para acceder a él, resultando mucho más eficiente.

**Listas Doblemente Enlazadas vs. Listas Simplemente Enlazadas.**

Las listas doblemente enlazadas requieren más espacio por nodo y sus operaciones básicas resultan más costosas pero ofrecen una mayor facilidad para manipular los elementos ya que permiten el acceso secuencial a la lista en ambas direcciones. En particular, se puede insertar o borrar un nodo con un número fijo de operaciones dando la dirección de dicho nodo (las listas simples requieren la dirección del nodo anterior para insertar o suprimir correctamente.). Algunos algoritmos requieren el acceso en ambas direcciones.

**Listas Circulares Enlazadas vs. Listas Lineales Enlazadas.**

Las listas circulares son más útiles para describir estructuras circulares y tienen la ventaja de poder recorrer la lista desde cualquier punto. También permiten el acceso rápido al primer y último elemento por medio de un puntero simple.

Las listas circulares evitan excepciones en las operaciones que se realicen sobre ellas. No existen casos especiales, cada nodo siempre tiene uno anterior y uno siguiente.

# CAPÍTULO 2

## ESTRUCTURA DE DATOS: PILA.

A lo largo de este capítulo se muestra el concepto de la estructura de datos Pila y de algunas de sus formas de implementación, al mismo tiempo se exponen las ventajas y las desventajas de cada una, así como las aplicaciones que tienen en la vida práctica, además de mostrar el seudocódigo. Lo mencionado anteriormente permite tener un mejor conocimiento de este tipo de estructura de datos y saber en un problema determinado cual es la más óptima para darle solución al mismo.

Las pilas pueden representarse mediante el uso de:

- Arreglos.
- Listas enlazadas.

**¿Qué es una Pila?**

Una pila (*stack* en inglés) es una estructura sencilla, mucho más simple que la lista y se puede definir como una colección ordenada de elementos ***S = (S1, S2,…, Sn)*** donde se adicionan y eliminan por un mismo extremo conocido como *tope*. Se dice que ***S1*** es el elemento del fondo de la pila y ***Sn*** es el elemento que se encuentra en el tope. Se les conoce como estructuras LIFO[2] debido al orden en que se apilan y extraen los elementos en la misma. (3)

[2] LIFO es el acrónimo correspondiente a *Last In First Out*, cuya traduccin literal sería: último en entrar, primero en salir.

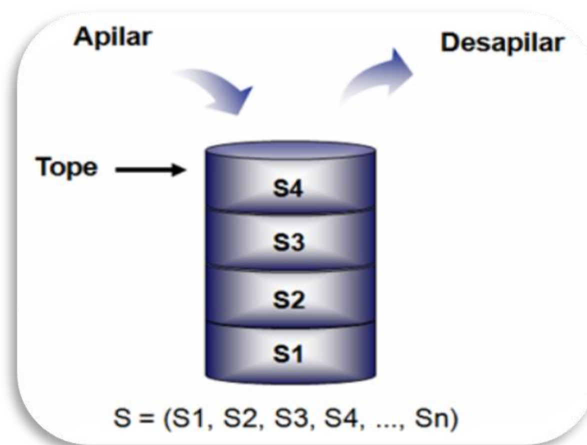


**Figura 2.1** Representación gráfica de una Pila.

La interfaz de este TDA provee las siguientes operaciones:

a. **Vacía ( ),** devuelve verdadero si la pila está vacía.
b. **Tope ( ),** devuelve el elemento que se encuentra en el tope de la pila, se conoce también con el nombre de *cima*. Si es llamado con la pila vacía lanza una excepción.
c. **Apilar (x),** coloca a ***x*** en el tope de la pila, se conoce también con el nombre de *adicionar*.
d. **Desapilar ( ),** elimina el elemento que se encuentra en el tope de la pila se conoce también con el nombre de *extraer*. Si es llamado con la pila vacía lanza una excepción.

**Aplicaciones:**

» Se usan en los compiladores (parsers: reconocedores sintácticos de los compiladores).
» En la programación de sistemas (para registrar llamadas a subprogramas y recuperar los datos anteriores, o recuperar los parámetros).
» Otra aplicación de las pilas lo constituye el mecanismo que establecen los lenguajes de programación para garantizar las llamadas anidadas a subprogramas dentro de una aplicación.

- Se aplican además en la recuperación de elementos en orden inverso al que fueron colocados (en un depósito, una pila de contenedores, sillas, etc.).
- Convertir notación infija a postfija o prefija.
- Para la implementación de la recursividad.

## 2.1 Pila utilizando arreglos.

**Definición 2.1:**

Una pila puede ser implementada usando un arreglo el cual se irá llenando en la forma usual, conjuntamente con este se tiene un atributo que almacena el índice de la última posición utilizada. (3).

Si se implementa una pila utilizando un arreglo se debe especificar primero el tamaño máximo de la pila y además definir una operación que nos indique que la misma está llena y en caso de que se desee adicionar un elemento sería necesario crear otro arreglo mayor que el anterior, copiar todos los elementos de la pila en este arreglo y finalmente liberar la memoria ocupada por el arreglo inicial.

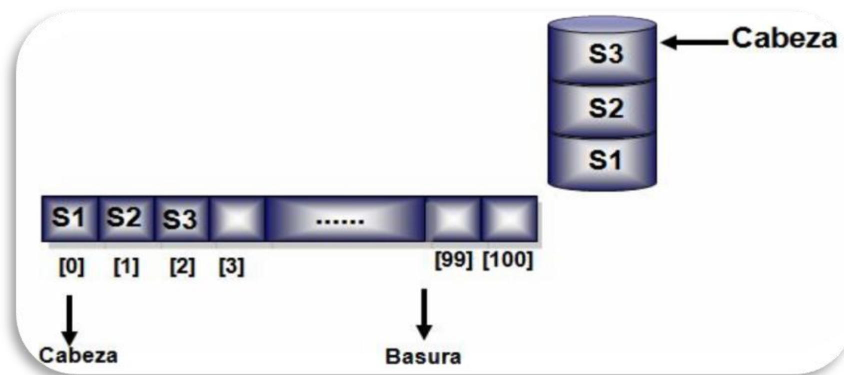


**Figura 2.2** Representación gráfica de una Pila utilizando arreglos.

**Desventajas:**

- El inconveniente de esta implementación es que es necesario fijar de antemano el número máximo de elementos que puede contener la pila,

*MAX_ELEM*, y por lo tanto al apilar un elemento es necesario controlar que no se inserte un elemento si la pila está llena.

**Seudocódigo:**

```
************************************************************
*                 Pila implementada sobre un Arreglo                 *
************************************************************
Vacía() : lógico
Inicio
    si (tope == Nulo) entonces
     retornar verdadero
sino
    retornar falso
Fin
.........................................................................................................
Llena(): lógico
Inicio
  retornar tope ← MAX_ELEMENTO-1
Fin
.........................................................................................................
Tope():Tipo
Inicio
    si ( no Vacía() )
        retornar arreglo[tope-1]
    sino
        Error(“LA PILA ESTA VACIA”)
Fin
.........................................................................................................
Apilar(x: Tipo )
Inicio
    si( no Llena( ) ) entonces
    inicio
        elementos [tope] ← x
```

```
        tope ← tope+1
    fin
    sino
        Error(“LA PILA ESTA LLENA”)
Fin
.........................................................................................................
Desapilar ( ): Tipo
Inicio
    si( no Vacía( ) )
    inicio
        tope ← tope – 1
        retornar arreglo[tope]
    fin
    sino
        Error(“LA PILA ESTA VACIA”)
Fin
```

**2.2 Pila utilizando listas enlazadas.**

**Definición 2.2:**

Otra implementación de la pila puede ser utilizando listas enlazadas, donde los nodos de la lista simplemente enlazada se emplean para almacenar la información de la pila. En este caso no existe el problema de tener que fijar el tamaño máximo de la pila pues la capacidad de la pila está acotada por la cantidad de memoria de la computadora y la operación que indica que la misma está llena no tiene sentido. (3)

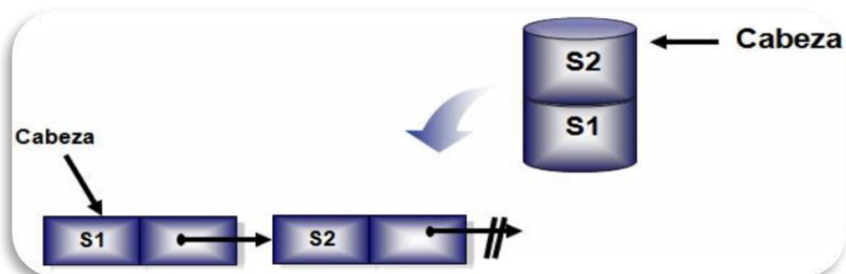


**Figura 2.3** Representación gráfica de una Pila con listas enlazables.

En este caso no existe el problema de tener que fijar el tamaño máximo de la pila (aunque siempre está acotado por la cantidad de la memoria disponible en la computadora).

**Ventajas:**

- En este caso no existe el problema de tener que fijar el tamaño máximo de la pila, es decir, no se necesita saber la cantidad de elementos que va a contener. Esto se debe a que al ser implementadas a base de punteros, se va tomando memoria a medida que se cargan los datos, si no hay más memoria disponible no se ingresan más datos.

**Aplicaciones:**

- Una de sus aplicaciones más sencillas es invertir una cadena.
- Otra aplicación más compleja consiste en identificar si una expresión tiene sus paréntesis balanceados[3].
- Permite la conversión de expresiones infijas a postfijas.
- Se utilizan en la implementación de la recursividad.(5)

**Seudocódigo:**

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                  Pila implementada utilizando listas enlazables                  *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Vacía () : lógico
Inicio
  retornar tope ← Nulo
Fin
.........................................................................................................
Tope(): Tipo
```

[3] Balanceados significa que para cada paréntesis abierto existe uno cerrado en el mismo orden.

```
Inicio
    si( no Vacía() )
        retornar tope.GetDato()
   sino
       Error(“LA PILA ESTA VACIA”)
Fin
.....................................................................................................
Apilar(x: Tipo)
Inicio
    nodo ← nuevo NodoSE (x, tope)
    tope ← nodo
Fin
.....................................................................................................
Desapilar(): Tipo
Inicio
    si( no Vacía() )
    inicio
        elemento ← tope.GetDato()
        nodo_a_eliminar ← tope
        tope ←  tope. siguiente
        eliminar nodo_a_eliminar
        retornar elemento
   fin
   sino
        Error(“LA PILA ESTA VACIA”);
Fin
```

**2.3 Observaciones Generales sobre el TDA Pila.**

El uso del arreglo es idóneo cuando se conoce de antemano el número máximo de elementos que van a ser apilados y el compilador admite una región contigua de memoria para el arreglo. En otro caso lo más recomendable sería usar la implementación por listas enlazadas, donde podría emplearse además si el número de elementos llegara a ser excesivamente grande.

La implementación por arreglo es ligeramente más rápida. En especial, es mucho más rápido a la hora de eliminar los elementos que hayan quedado en la pila. Por lista enlazada esto no es tan rápido. Por ejemplo, piénsese en un algoritmo que emplea una pila y que en algunos casos al terminar éste su ejecución deja algunos elementos sobre la pila. Si se implementa la pila mediante una lista enlazada entonces quedaría en memoria una serie de elementos que es necesario borrar. La única manera de borrarlos es liberar todas las posiciones de la memoria de la computadora que le han sido asignadas a cada elemento, esto es desapilar todos los elementos. En el caso de una implementación con arreglo esto no es necesario, salvo que quiera liberarse la región de la memoria ocupada por éste.(6)

No es difícil implementar las operaciones *Apilar* y *Desapilar* en tiempo **Θ (1)** utilizando tanto una lista con disposición secuencial, como una lista enlazada. Para ambas formas de listas se tienen dos opciones básicas: se puede mantener el tope de la pila en la cabeza o en la cola de la lista. En una lista con disposición secuencial es más eficiente mantener el tope de la pila en la cola de la lista. En este caso, en la operación de *Apilar* simplemente se añade un nuevo elemento a la cola de la lista y en una operación de *Desapilar* se elimina el elemento que se encuentra al final de la lista. Debido a que una lista con disposición secuencial permite acceso directo, ambas operaciones pueden implementarse en tiempo **Θ (1).**

Mantener el tope de la pila en la cabeza de una lista con disposición secuencial no permite implementar tan eficientemente las operaciones del *TDA Pila*. Para ver por qué, recuérdese que insertar un nuevo elemento en la cabeza de una lista requiere que los elementos de las posiciones ***1*** hasta ***n*** sean ascendidas a las posiciones ***2*** hasta ***n+1***, es una operación que necesita tiempo **Θ (n).** Además la eliminación de la cabeza de la lista también consume tiempo **Θ (n)**, dado que los elementos de las posiciones ***2*** hasta ***n*** deben ser descendidos. Por tanto, ambas operaciones *Apilar* y *Desapilar* requerirían **Θ(n)**, en este caso.

En una implementación con listas enlazadas, los elementos pueden ser almacenados tanto en la cabeza como en la cola de la lista (asumiendo que se almacena un puntero a la cola) y ambas operaciones *Apilar* y *Desapilar pueden implementarse en tiempo* **Θ (1).**

# CAPÍTULO 3

# ESTRUCTURA DE DATOS: COLA.

En el transcurso de este capítulo se aborda el concepto de la estructura de datos Cola y de algunas de sus formas de implementación, conjuntamente se exponen las ventajas y las desventajas de cada una, así como las aplicaciones que tienen en la vida práctica, además de mostrar el seudocódigo. Lo mencionado anteriormente permite tener un mejor conocimiento de este tipo de estructura de datos y saber en un problema determinado cual es la más óptima para darle solución al mismo.

**¿Qué es una cola?**

Una cola (*queue* en inglés) es una estructura de datos caracterizada por ser una lista ordenada de elementos ***Q = (Q1, Q2,..., Qn)***, donde la operación de adición se realiza por un extremo, llamado *cola*, y la operación de extracción por el otro, llamado *cabeza* y tiene un comportamiento de tipo FIFO[4] por el modo de acceso a sus elementos. (3)

[4] FIFO es el acrónimo correspondiente a *First In First Out,* cuya traducción literal sería: primero en entrar, primero en salir.

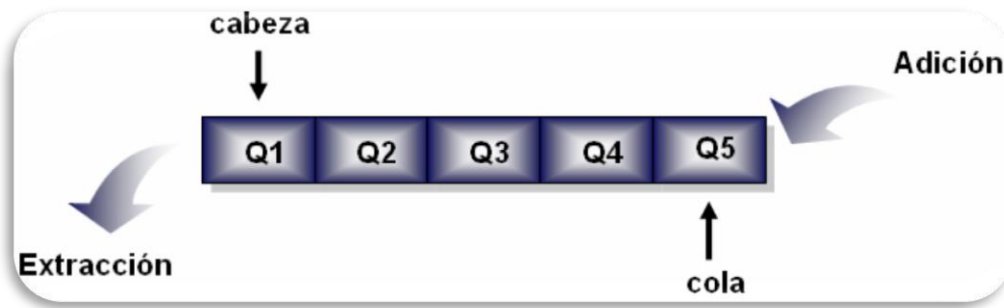


**Figura 3.1** Representación gráfica de una Cola.

La Cola tiene como operaciones básicas:

a. **Vacía ( ),** devuelve verdadero si la cola está vacía.
b. **Frente ( ),** devuelve el elemento que se encuentra en la primera posición de la cola. Se dispara una excepción cuando la cola está vacía.
c. **Fondo ( ),** devuelve el elemento que se encuentra en la última posición de la cola. Se dispara una excepción cuando la cola está vacía.
d. **Adicionar (x),** coloca a ***x*** en la cola (después del último elemento) de la cola.
e. **Extraer ( ),** elimina el elemento que se encuentra en la cabeza de la cola, si está vacía se dispara una excepción.

**Aplicaciones:**

» Las colas, al igual que las pilas, resultan de aplicación habitual en muchos problemas informáticos.

» Su utilización es infinita, sobre todo en aquellos problemas que tienen un componente de simulación de procesos, por ejemplo la simulación de una cola formada frente a un cajero automático.

» Para modelar 'colas reales' en el mundo de las computadoras: colas de tareas, colas de procesos, colas de impresión en el sistema operativo *Windows*, etc. Cada usuario de una red de *Windows* coloca sus trabajos de

impresión y el sistema lo imprime en el mismo orden en que fueron insertados en la cola de impresión.

» La aplicación más común de las colas es la organización de tareas de un ordenador. Los procesos forman colas para la utilización de los recursos de un sistema computacional.

## 3.1 Cola de prioridad.

**Definición 3.1:**

Una cola de prioridad es una cola cuyos elementos tienen asociado una prioridad y se atienden en el orden indicado por la misma, de forma que el orden en que los elementos son procesados sigue las siguientes reglas:

- El elemento con mayor prioridad es procesado primero.
- Si varios elementos tienen la misma prioridad, se atenderán en dependencia del orden en que fueron adicionados. Hay dos formas de implementación:

  1. A cada nodo añadirle un campo con su prioridad. Resulta conveniente mantener la cola ordenada por orden de prioridad.

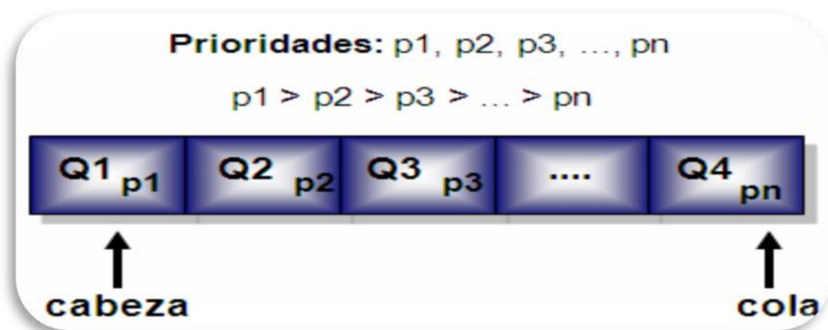


**Figura 3.2** Representación gráfica de una Cola de Prioridad.

1. Crear tantas colas como prioridades exista y almacenar a cada elemento en la cola correspondiente.

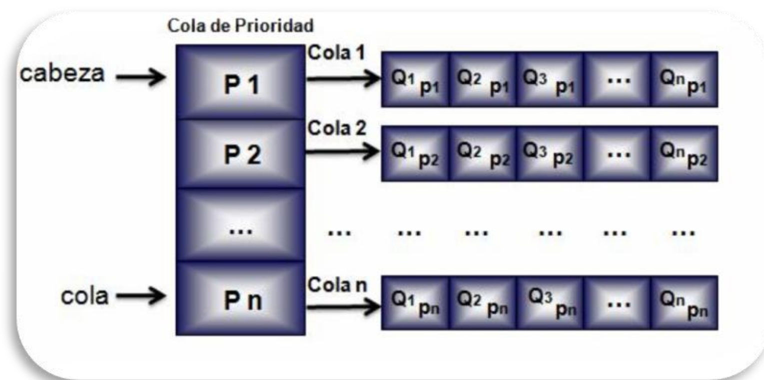


**Figura 3.3** Representación gráfica de una Cola de Prioridad con varias Colas asociadas a cada prioridad.

**Tipos de Colas de Prioridad**:

- Colas de prioridad con ordenamiento ascendente: en ellas los elementos se insertan de forma arbitraria, pero a la hora de extraerlos, se extrae el elemento de menor prioridad.
- Colas de prioridad con ordenamiento descendente: son iguales que las colas de prioridad con ordenamiento ascendente, pero al extraer el elemento se extrae el de mayor prioridad.

Operaciones de las colas de prioridad (en cada una de estas operaciones, ***P*** se supone que es una cola de prioridad arbitraria):

a. **Insertar (P, x)**, añade el elemento ***x*** a ***P***.
b. **Encontrar_Min (P)**, devuelve elemento de ***P*** con la prioridad más alta (menor valor de clave). Si ***P*** está vacía esta operación produce un error.
c. **Eliminar_Min (P)**, quita y devuelve el elemento de ***P*** con la prioridad más alta (menor valor de clave). Si ***P*** está vacía esta operación produce un error.(4)

**Desventajas:**

- La implementación de las colas de prioridad es costosa en algunas implementaciones:

- Si se mantiene la lista desordenada, la inserción queda constante, pero la eliminación y la búsqueda exigen una búsqueda lineal.
- Si se escoge ordenar la lista, entonces es la inserción la operación que requiere una búsqueda lineal a cambio del coste constante de la consulta, mientras que en la eliminación depende de la representación concreta de la lista.

- Una operación intermedia es mantener la lista desordenada durante las inserciones y ordenarla a continuación, siempre que todas las inserciones se hagan al principio y las consultas y las eliminaciones a continuación. En este caso el coste total resulta ***O(n log n)***, menor que el coste **On$^2$** de los dos casos anteriores.

**Aplicaciones:**

» Las colas de prioridad se aplican cuando las solicitudes deben procesarse en el orden de prioridad y no en el orden de llegada.

» Una aplicación de las colas de prioridad es la planificación de los trabajos o procesos en un sistema informático multiusuario. Un planteamiento simple consiste en utilizar una cola para atender a los trabajos con una filosofía de que el primero en llegar es el primero en ser atendido. Este planteamiento no funciona bien cuando trabajos cortos quedan “atrapados” en la cola detrás de trabajos largos en ejecución, generalmente es mejor dejar que los trabajos cortos se ejecuten primero.

» Gestión de los procesos en un sistema operativo. Los procesos no se ejecutan uno tras otro en base a su orden de llegada. Algunos procesos deben tener prioridad (por su mayor importancia, por su menor duración, etc.) sobre otros.

» Implementación de algoritmos voraces, los cuales proporcionan soluciones globales a problemas basándose en decisiones tomadas sólo con información local. La determinación de la mejor opción local suele basarse en una cola de prioridad.

- Algunos algoritmos sobre grafos, como la obtención de caminos o árboles de expansión de mínimo coste, son ejemplos representativos de algoritmos voraces basados en colas de prioridad.

» Implementaciones eficientes de algoritmos de simulación de sucesos discretos. En estas, el avance del tiempo no se gestiona incrementando un reloj unidad a unidad, sino incrementado sucesivamente el reloj al instante del siguiente suceso.

### 3.1.1 Cola de prioridad utilizando arreglos.

**Definición 3.2:**

La implementación más sencilla que puede llevarse a cabo es utilizar un arreglo lineal. Al igual que en la lista, a estas operaciones se les pueden agregar otras.

Por ejemplo si se implementa la cola utilizando arreglos se puede añadir una operación que nos indique cuando la misma está llena, en caso de que se quisiera adicionar más elementos habría que reservar más espacio en la memoria de la computadora. (4)

**Seudocódigo:**

```
****************************************************
*            Cola de Prioridad utilizando arreglos            *
****************************************************
Longitud():
Inicio
    si frente ≠ Nulo: entonces
        longitud ← longitud + 1
        arreglo ← arreglo[frente + 1]
    retornar longitud
Fin
```

```
.......................................................................................................
Frente():
Inicio
        si (no es Vacía()):
          retornar arreglo[frente]
        sino:
          Error ("COLA VACIA")
Fin
.......................................................................................................
Fondo():
Inicio
        si (no es Vacía()):
          retornar arreglo[fondo]
        sino:
          Error ("COLA VACIA ")
Fin
.......................................................................................................
Adicionar(elemento):
Inicio
       auxiliar ← arreglo[frente]
       anterior ← 0
       mientras (auxiliar ≠ Nulo y auxiliar.GetPrioridad() >= elemento.GetPrioridad()):
          Inicio
                arreglo[anterior] ←  arreglo[auxiliar]
                auxiliar ← arreglo[auxiliar+1]
           Fin
       x ← Nulo
       si (arreglo[anterior] == Nulo): entonces
          arreglo[frente] ← x
          si (arreglo[fondo] == Nulo): entonces
             arreglo[fondo] ← x
          sino:
             arreglo[anterior] ← x
             si (auxiliar == Nulo): entonces
```

```
                    arreglo[fondo] ← x
Fin
.........................................................................................................
Extraer():
Inicio
        si no es Vacía(): entonces
            elemento ← arreglo[frente]
            longitud ← longitud - 1
            retornar elemento
        sino:
            Error ("COLA VACIA")
Fin
```

### 3.1.2 Cola de prioridad utilizando listas enlazadas.

**Definición 3.3:**

Al igual que para la pila las implementaciones de Cola pueden ser obtenidas utilizando algunas implementaciones del TDA Lista, así como con la utilización de nodos enlazados, para esta variante la clase puede utilizar un nodo frente y un nodo fondo.

**Seudocódigo:**

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                    Cola de Prioridad utilizando Listas Enlazadas                    *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
  Vacía():
Inicio
        si frente == 0:
            retornar True
        else:
            retornar False
```

```
Fin
.....................................................................................................
Longitud():
Inicio
        longitud ← 0
        auxiliar ← NodoSE()
        mientras auxiliar ≠ Nulo:
            longitud ← longitud + 1
            auxiliar ← auxiliar.GetSiguiente()
        retornar longitud
Fin
.....................................................................................................
Frente():
Inicio
        si no es Vacía():
            retornar frente.GetDato()
        sino:
            Error ("COLA VACIA ")
Fin
.....................................................................................................
Fondo():
Inicio
        si no es Vacía():
            retornar fondo.GetDato()
        sino:
            Error ("COLA VACIA ")
Fin
.....................................................................................................
 Adicionar(elemento):
Inicio
       auxiliar ← frente.GetDato()
        anterior  ←  NodoSE()
        mientras (auxiliar ≠ Nulo y auxiliar.GetPrioridad() >= elemento.GetPrioridad()):
           inicio
```

```
            anterior ← auxiliar
            auxiliar ← auxiliar.GetSiguiente()
      fin
      nodo ← NodoSE()
      si (anterior == Nulo): entonces
         frente ← nodo
         si (fondo == Nulo): entonces
            fondo ← nodo
         sino:
            anterior.SetSiguiente(nodo)
            si (auxiliar == Nulo):
               fondo ← nodo
Fin
.............................................................................................................
Extraer():
Inicio
      auxiliar ← NodoSE()
      si no es Vacía():
         auxiliar ← frente
         frente ← frente.GetSiguiente()
         retornar auxiliar.GetDato()
      sino:
         Error ("COLA VACIA ")
Fin
```

### 3.2 Cola de amigos.

Una Cola de Amigos se comporta de manera similar a una Cola de Prioridad, con la particularidad de que en la misma para insertar un elemento se busca el primer elemento amigo partiendo desde el frente de la cola y se coloca delante él.

### 3.2.1 Cola de Amigos utilizando arreglos.

**Definición 3.4:**

La implementación más sencilla que puede llevarse a cabo es utilizar un arreglo lineal. Al igual que en la lista, a estas operaciones se les pueden agregar otras. Por ejemplo si se implementa la cola utilizando arreglos se puede añadir una operación que indique cuando la misma está llena. En caso de que se quisiera adicionar nuevos elementos habría que reservar más espacio en la memoria de la computadora.(3)

**Seudocódigo:**

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                    Cola de Amigos utilizando arreglos                    *
* * * * * * * * * * * * * * * * * * * * * * * * * * * ** * * * * * * * * * * * * * * * * * * * * * *
Adicionar(Tipo x)
```

### 3.2.2 Cola de Amigos utilizando listas enlazadas.

**Definición 3.5:**

Al igual que en la pila, las implementaciones de una cola pueden ser obtenidas utilizando algunas implementaciones del TDA Lista, así como con la utilización de nodos enlazados, para esta variante la clase puede utilizar un nodo frente y un nodo fondo.

**Seudocódigo:**

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                Cola de Amigos utilizando listas enlazadas                *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Adicionar_Amigo(elemento):
```

```
Inicio
        NuevoNodo ←  NodoSE()
        NuevoNodo.SetDato(elemento)
        si frente.dato == None: entonces
            frente ←  NuevoNodo
            fondo  ← self.frente
        sino:
        inicio
            si frente.GetDato().GetAmigo() >= elemento.GetAmigo(): entonces
                NuevoNodo.SetSiguiente(frente)
                frente ←  NuevoNodo
            sino:
                Anterior ←  frente
                Siguiente ←  frente.GetSiguiente()

                mientras Siguiente is not Nulo y Siguiente.GetDato().GetAmigo() <
elemento.GetAmigo():
                    Anterior ←  Siguiente
                    Siguiente ←  Siguiente.GetSiguiente()

                Si Siguiente == None:
                    fondo ←  NuevoNodo
                    Anterior.SetSiguiente(NuevoNodo)
                sino:
                    NuevoNodo.SetSiguiente(Siguiente)
                    Anterior.SetSiguiente(NuevoNodo)
             fin
Fin
```

**3.3 Observaciones Generales sobre los TDA Colas.**

Al igual que con las pilas, la mejor implementación depende de la situación particular. Si se conocen de antemano el número de elementos entonces lo ideal es

una implementación por arreglos. En otro caso se recomienda el uso de la lista enlazada.(8)

Implementar el TDA Cola supone añadir elementos en un extremo de la lista durante una operación *Adicionar*, y quitarlos del otro extremo de la lista durante al operación *Extraer*. Esto garantiza que los elementos son accedidos en orden FIFO.

- Esta estrategia puede ser implementada eficientemente utilizando una lista enlazada, dado que insertar un elemento en la posición **1** de una lista enlazada solo consume tiempo **Θ (1)**, lo mismo que añadir un elemento a la cola de una lista enlazada.
- Implementar la estrategia anteriormente mencionada utilizando un arreglo es, sin embargo, bastante ineficiente. Si el primero de la cola se fija en la cabeza de la lista, entonces la operación *Eliminar* conllevaría a descender una disposición en la lista **n-1** elementos. Similarmente, fijando el último de la cola en la cola de la lista provoca que la operación *Adicionar* realice **n-1** desplazamientos.(9)

# CAPÍTULO 4

# ESTRUCTURA DE DATOS: TABLA HASH.

Durante este capítulo se manifestará el concepto de la estructura de datos: tabla hash y de algunas de sus estrategias para la resolución de colisiones, simultáneamente se exponen las ventajas y las desventajas de cada una, así como las aplicaciones que tienen en la vida práctica, además de mostrar el seudocódigo. Lo mencionado anteriormente permite tener un mejor conocimiento de este tipo de estructura de datos y saber en un problema determinado cual es la más óptima para darle solución al mismo.

**¿Qué es una tabla hash?**

Una tabla hash, mapa hash o tabla de dispersión es una estructura de datos que está formada por las combinaciones de *llaves* o *claves* con *valores* organizados en "sectores de almacenamiento", para permitir realizar una búsqueda rápida. Constituye un TDA especial para la manipulación y almacenamiento de la información en la memoria secundaria de la computadora.

Una tabla hash se construye con tres elementos básicos.

- Una *estructura de acceso directo,* como un arreglo, capaz de almacenar ***N*** cantidad de elementos.
- Una *función de dispersión* que permita a partir de la clave obtener el índice donde estará almacenado el dato asociado a esa clave y cuyo dominio sea el espacio de claves y su imagen (o rango) los números naturales.
- Una *función de resolución de colisiones*. (5)

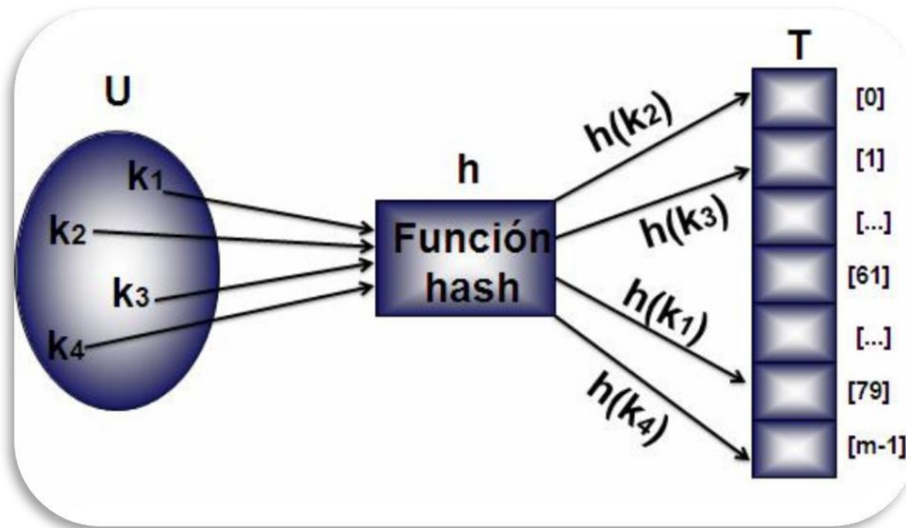


**Figura 4.1** Representación gráfica de una Tabla Hash.

La operación principal que soporta de manera eficiente es la *búsqueda,* que permite el acceso a los elementos (por ejemplo: teléfono y dirección) almacenados a partir de una clave generada (por ejemplo: usando el nombre o número de cuenta). Funciona transformando a través de una función de dispersión, la clave en un índice, que no es más *que* un número que la tabla hash utiliza para localizar el valor deseado.

Las operaciones básicas implementadas en las Tablas Hash son:

- **Insertar (clave, elemento),** inserta un elemento en un índice de la tabla hash generado por una función de dispersión dado una clave.
- **Buscar (clave),** busca el elemento que se encuentra almacenado en el índice de la tabla hash generado por una función de dispersión dado una clave.
- **Eliminar (clave),** elimina el elemento que se encuentra almacenado en el índice de la tabla hash generado por una función de dispersión dado una clave.

Es inevitable que claves diferentes lleguen a producir el mismo resultado de la función de dispersión y esto constituye un problema.

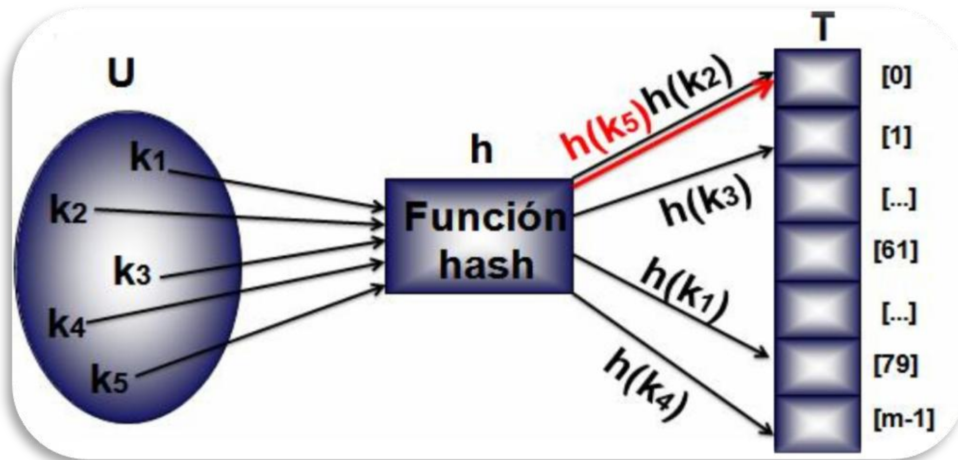


**Figura 4.2** Representación gráfica de una colisión.

Pero existen estrategias que tratan de resolver este problema de mejor o peor manera, tal es el caso de la:

- Resolución de colisiones por dispersión Abierta (o externa).
- Resolución de colisiones por dispersión Cerrada (o interna).

**Ventajas:**

» Una tabla *hash* tiene como principal ventaja que el acceso a los datos suele ser muy rápido si se cumplen las siguientes condiciones:

- Una razón de ocupación no muy elevada (a partir del 75% de ocupación se producen demasiadas colisiones y la tabla se vuelve ineficiente).
- Una función resumen que distribuya uniformemente las claves. Si la función está mal diseñada, se producirán muchas colisiones.

» Otra ventaja, es que están implementadas usando algoritmos de *hashing* que los hace muy rápidos (aunque las listas asociativas son más fáciles de crear y manipular).

» Comparada con otras estructuras de arreglos asociadas, las tablas hash son más útiles cuando se almacenan grandes cantidades de información.

**Desventajas:**

- Su principal inconveniente es que se trata de una estructura estática fija y al implementar cualquier otra operación el coste es muy elevado.
- Dificultad para recorrer todos los elementos. Se suelen emplear listas o colecciones (*Collection* usadas en *.net*) para procesar la totalidad de los elementos.
- Desaprovechamiento de la memoria. Si se reserva espacio para todos los posibles elementos, se consume más memoria de la necesaria; se suele resolver reservando espacio únicamente para punteros a los elementos.
- Las tablas hash almacenan la información en posiciones seudo-aleatorias, así que el acceso ordenado a su contenido es bastante lento.

**Aplicaciones:**

- Es uno de los medios más eficientes para implementar el TDA Diccionario.
- Se emplea en la implementación de la tabla de símbolos de un compilador.

## 4.1 Resolución de colisiones por dispersión abierta.

**Definición 3.1:**

Una de las estrategias más simples de resolución de colisiones, denominada ***dispersión abierta***, se basa en colocar en una lista enlazada todos los elementos que se dispersan en el mismo hueco. Así todos los elementos almacenados en una lista enlazada tendrán la misma clave. En este caso los huecos de la tabla dispersa no almacenan ya elementos, sino punteros a listas enlazadas asignadas dinámicamente que almacenan todos los elementos que se dispersan en ese hueco, como se muestra en la figura 4.2. Esta estrategia se extiende fácilmente para permitir cualquier estructura de datos dinámica, no solo listas enlazadas. Téngase en cuenta que con la dispersión abierta el número de elementos que pueden ser almacenados solo está limitado por la cantidad de memoria disponible.

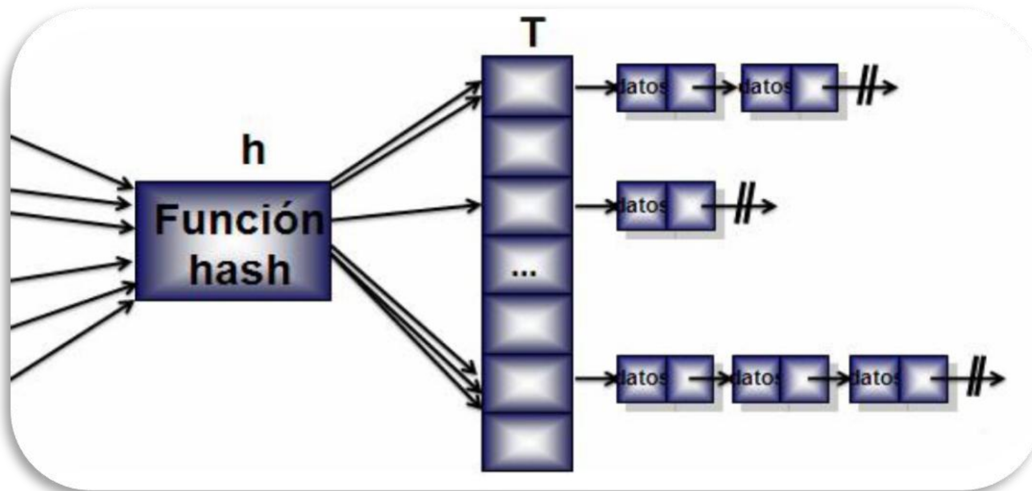


**Figura 4.3** Representación gráfica de la resolución de colisiones por dispersión abierta en una tabla hash.

**Ventajas:**

» El número de elementos que pueden ser almacenados solo está limitado por la cantidad de memoria disponible en la computadora.

**Seudocódigo:**

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*     Tabla Hash - Resolución de colisiones por dispersión abierta      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Funcion_Dispersión(clave, num_ensayos ← 0): # Método de la División
      índice ←  mod(clave, tamaño)
      retornar índice
..........................................................................................................
Insertar(elemento, clave):
Inicio
      índice ← Funcion_Dispersión (elemento.Clave(), num_ensayos = 0)
      si ( (índice < 0) o  (índice >= tamaño) ):
         retornar Error("Índice fuera de rango")
      sino:
```

```
        tabla[índice].Insertar(elemento, índice)
        retornar ("El elemento fue insertado correctamente")
Fin
.........................................................................................................
Buscar(clave):
Inicio
      índice ← Funcion_Dispersión (elemento.Clave(), num_ensayos = 0)
      si ( (índice < 0) o  (índice >= tamaño) ):
        retornar Error("Índice fuera de rango")
      sino:
        tabla[índice]. Obtener(índice)
        retornar ("El elemento fue encontrado")
Fin
.........................................................................................................
Eliminar(clave):
Inicio
      índice ← Funcion_Dispersión (elemento.Clave(), num_ensayos = 0)
      si ( (índice < 0) o  (índice >= tamaño) ):
        retornar Error("Índice fuera de rango")
      sino:
        tabla[índice]. Eliminar(índice)
        retornar ("El elemento fue eliminado correctamente")
Fin
```

### 4.2 Resolución de colisiones por dispersión cerrada.

**Definición 3.2:**

En la **dispersión cerrada** todos los elementos se almacenan en la propia tabla dispersa. En este caso, las colisiones se resuelven calculando una secuencia de huecos de dispersión. Esta secuencia es examinada o explorada sucesivamente hasta que se encuentra un hueco en la tabla dispersa vacío en el caso de *Insertar*, o se encuentre el elemento deseado en el caso de *Buscar* o *Eliminar.*

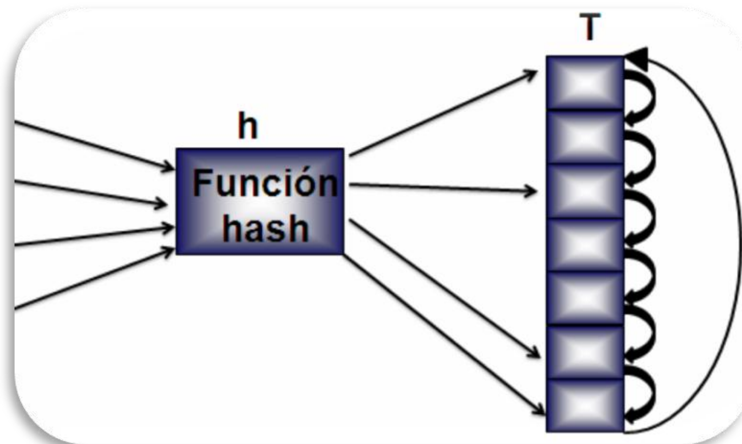


**Figura 4.4** Representación gráfica de la resolución de colisiones por dispersión cerrada en una tabla hash.

**Ventajas:**

- La ventaja de este enfoque es que evita el uso de punteros. La memoria que se ahorra al no almacenar puteros puede utilizarse para construir una tabla dispersa más grande si es necesario. De este modo, utilizando la misma cantidad de memoria se puede construir una tabla dispersa mayor, que potencialmente conduce a menos colisiones y por lo tanto, a unas operaciones más rápidas del TDA Diccionario.
- Evita el uso de una segunda estructura de datos.

**Desventajas:**

- Usa un espacio fijo para el almacenamiento de los elementos, por lo que limita el tamaño de los conjuntos.
- Se pueden llegar a producir colisiones en cadena.
- Necesidad de ampliar el espacio de la tabla si el volumen de datos almacenados crece. Se trata de una operación costosa.

**Seudocódigo:**

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*     Tabla Hash - Resolución de colisiones por dispersión cerrada     *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
Funcion_Dispersión(clave, num_ensayos): # Exploración Lineal
Inicio
   c1 ← 1
   retornar mod ( (mod(clave, tamaño) + c1 * num_ensayos), tamaño)
Fin
.........................................................................................................
Insertar(elemento):
Inicio
     mientras Vacío() es True  y  Eliminado() es True y num_ensayos < límite:
        índice ← Funcion_Dispersión (elemento.Clave(), num_ensayos)
        si num_ensayos >=  límite: entonces
           retornar ("Índice fuera de rango")
        sino
           tabla[índice] ← elemento
           retornar ("El elemento fue insertado correctamente")
Fin
.........................................................................................................
Buscar(clave):
Inicio
     mientras Vacio() is True  y  Eliminado() is True y num_ensayos < límite:
        índice ← Funcion_Dispersión (elemento.Clave(), num_ensayos)
        si num_ensayos >=  límite: entonces
           retornar ("Índice fuera de rango")
        sino
           elemento ←  tabla[índice]
           retornar ("El elemento fue encontrado")
Fin
.......................................................................................................
Eliminar(clave):
```

```
Inicio
    mientras Vacío() is True  y  Eliminado() is True and num_ensayos < límite:
        índice ← Funcion_Dispersión (elemento.Clave(), num_ensayos)
        si num_ensayos >=  límite: entonces
            retornar ("Índice fuera de rango")
        sino
            tabla[índice] ← tabla[índice+1]
            retornar ("El elemento fue eliminado correctamente")
Fin
```

## 4.3 Observaciones Generales sobre la estructura de datos: Tabla Hash.

### 4.3.1 Función de dispersión.

Las propiedades más importantes de una buena función de dispersión son:

- » Que pueda ser calculada muy rápidamente (que solo se requieran unas pocas operaciones simples).
- » Y que al mismo tiempo se minimicen las colisiones.

Con el fin de minimizar las colisiones, una función de dispersión no debería de mostrar un sesgo hacia ningún hueco en particular de la tabla dispersa. Idealmente, una función de dispersión tendrá la propiedad de que cada clave tiene la misma probabilidad de dispersarse en cualquiera de los huecos de la tabla dispersa.

Para cada uno de los métodos de dispersión expuestos a continuación se asume que:

**K**: representa una clave arbitraria.
**M**: representa el tamaño de la tabla hash.
**N**: representa el número de elementos almacenados en la tabla hash.
**H**: representa a la función de dispersión.
**H (K):** representa el índice o el valor de dispersión calculado por la función de dispersión ***H*** cuando se le suministró la clave ***K***.

## DISPERSION ABIERTA

### » Método de la División:

Las funciones de dispersión que hacen uso del *método de la división* generan los valores de dispersión calculando el resto de **K** dividido entre **M**:

**H(k) = K mod M**

Con esta función de dispersión, **H (K)** siempre calculará un valor entero en el rango: ***0, 1, …, M-1***

La elección de **M** resulta crítica para el rendimiento del método de la división. Por ejemplo, escoger **M** como una potencia de **2** es habitualmente poco recomendable, dado que **H (K)** será simplemente los ***p*** dígitos menos significativos de **K** cuando $M = 2^p$. En este caso, la distribución de claves en la tabla hash se basa solamente en una parte de la información contenida en las claves. Salvo que los bits de las claves sean realmente aleatorios, esto sesgará la función de dispersión hacia huecos particulares. Por razones similares, debería evitarse escoger **M** como una potencia de **10**. En este caso, cuando $M = 10^p$, **H (K)** es simplemente los últimos ***p*** dígitos de la representación decimal de **K**.

En general, las mejores elecciones para **M** cuando se utiliza el *método de la división* resultan ser números primos que no dividan a $r^l \pm a$, siendo $l$ y $a$ números naturales pequeños, y **r** la raíz del conjunto de caracteres que se esté usando (habitualmente **r = 128 ó 256**). Si el tamaño de la tabla **M**, viola esta condición, un gran números de valores de dispersión tenderán a ser simples superposiciones de los dígitos de la clave. Una elección particularmente mala es **M = r -1** (obsérvese que en este caso **l = a = 1**)

### » Método de la Multiplicación:

Aunque el *método de la división* tiene las ventajas de ser simple y fácil de usar, su sensibilidad a la elección de **M** puede ser demasiado restrictiva.

La principal ventaja del método de la multiplicación es que la elección de **M** no es crítica, de hecho, **M** se escoge a menudo como una potencia de **2** en las implementaciones con aritmética de coma fija.

Las funciones de dispersión que hacen uso del método de la multiplicación generan valores de dispersión en dos pasos:

Primero se calcula la parte decimal del producto de **K** y cierta constante real **A**, donde: $0 < A < 1$.

Este resultado es entonces multiplicado por **M** antes de aplicarle la función de truncado para obtener el valor de dispersión:

$$H(K) = [\, M\, (K*A - [K*A]\,)\,]$$

Obsérvese que ***K*A – [K*A]*** obtiene la parte decimal del número real **K*A**. Dado que la parte decimal es mayor o igual que **0** y menor que **1**, los valores de dispersión son enteros en el rango ***0, 1, …, M -1***. Una elección para **A** que habitualmente hace un buen trabajo al distribuir las claves por toda la tabla dispersa es la inversa de la razón áurea.

$$A = \Phi^{-1} \approx 0,61803399$$

El *método de la multiplicación* exhibe cierto número de agradables características matemáticas. Debido a que los valores de dispersión dependen de todos los bits de la clave, las permutaciones de una clave no tienen mayor probabilidad de colisión por cualquier otro par de claves. Además las claves como **«ptr1»** y **«ptr2»** son muy similares y por tanto tienen valores de claves transformados que se encuentran numéricamente cercanos uno al otro pero producirán valores de dispersión que están ampliamente separados.

## DISPERSION CERRADA

En la dispersión cerrada, las funciones de dispersión habituales comentadas anteriormente son modificadas para utilizar tanto una clave, como un número de ensayo al calcular un valor de dispersión. Esta información adicional se utiliza para

construir la secuencia de exploración. Más concretamente, en la dispersión cerrada, las funciones de dispersión son aplicaciones:

$$\mathbf{H: U \times \{0, 1, \ldots, M-1\}}$$

Y producen la secuencia de exploración:

$$\langle \mathbf{H\ (K, 0), (K, 1), (K, 2), \ldots} \rangle$$

Debido a que la tabla contiene **M** huecos, a lo sumo puede haber **M** valores distintos en una secuencia de exploración. Téngase en cuenta, sin embargo, que para una secuencia de exploración dada se permite la posibilidad de que **H (K, i) = H (K, j)** para **i ≠ j**. Por tanto, es posible que una secuencia de exploración contenga más de **M** valores.

Insertar un elemento utilizando dispersión cerrada conlleva sondear la tabla dispersa utilizando la secuencia de exploración calculada hasta que se encuentre un hueco vacío en el arreglo, o bien se cumplan algunos criterios de parada.

» **Exploración Lineal:**

Esta es una de las estrategias de exploración más simples de implementar, sin embargo, su rendimiento tiende a decaer rápidamente conforme aumenta el factor de carga. El estudio de la exploración lineal es también instructivo debido a que es fácil demostrar con este enfoque las condiciones patológicas que pueden surgir cuando se utiliza la dispersión cerrada.

Si la primera componente explorada es **j**, y $\mathbf{c_1}$ es una constante positiva, la secuencia generada por la exploración lineal es:

$$\langle j, (j + c1 \cdot 1) \bmod M, (j + c1 \cdot 2) \bmod M, \ldots \rangle$$

Dada una función de dispersión ordinaria **H´: U → {0, 1, …, M-1}**, una función de dispersión que utilice exploración lineal se construye fácilmente utilizando:

$$H(K, j) = (H'(K) + c_1 \cdot i) \bmod M$$

Donde **i = 0, 1, …, M-1** es el número de ensayo. Así el argumento suministrado al operador ***mod*** es una función lineal del número de ensayo.

Debería tenerse en cuenta que algunas elecciones de $c_1$ y **M** funcionan mejor que otras. Por ejemplo si se escoge **M** arbitrariamente y $c_1$ = 1 entonces todo hueco de la tabla dispersa puede ser examinado en **M** ensayos. Sin embargo, si escogemos un **M** que sea un número par y $c_1 = 2$, entonces solo la mitad de los huecos pueden ser examinados por una secuencia de exploración dada. En general, $c_1$ ha de ser escogido de tal forma que sea primo con respecto a **M** si se quiere que todos los huecos de la tabla dispersa sean examinados por la secuencia de exploración.

El empleo de la exploración lineal conduce a un problema conocido como *agrupamiento*, donde los elementos tienden a juntarse o agruparse en la tabla dispersa de tal forma que solo pueden ser accedidos por medio de una larga secuencia de exploración. Esto proviene del hecho de que tras aparecer un pequeño grupo en la tabla dispersa, se convierte en «objetivo» de las colisiones durante las inserciones subsiguientes.

Se ha observado que la secuencia de exploración en la exploración lineal está completamente determinada por el valor de dispersión inicial y dado que hay **M** posibles de estos, el número de secuencias de exploración distintas es **M.** Este hecho junto con los problemas de agrupamiento conspira para hacer que la exploración lineal sea una pobre aproximación a la dispersión uniforme cuando **N** se acerca a **M.**

» **Exploración Cuadrática:**

Esta es una extensión simple de la exploración lineal en la que uno de los argumentos suministrados a la operación ***mod*** es una función cuadrática del número de ensayo. Más concretamente, dada cualquier función de dispersión ordinaria **H´**,

una función de dispersión que utilice dispersión cuadrática puede construirse utilizando:

$$H(K, j) = (H'(K) + c_1 \cdot i + c_2 \cdot i^2) \bmod M$$

Donde $c_1$ y $c_2$ son constantes positivas. De nuevo las elecciones de $c_1$, $c_2$ y **M** resultan críticas para el rendimiento de este método.

Dado que el argumento izquierdo de la operación ***mod*** en la ecuación es una función no lineal del número de ensayo, las secuencias de exploración no pueden ser generadas a partir de otras secuencias de exploración por medio de simples desplazamientos cíclicos. Esto elimina el problema del agrupamiento primario y tiende a hacer que la exploración cuadrática funcione mejor que la exploración lineal. No obstante, como con la exploración lineal, el ensayo inicial **H (K, 0)** determina toda la secuencia de exploración y el número de secuencias de exploración distintas es **M**. Así el agrupamiento secundario es todavía un problema, y la exploración cuadrática solo ofrece una buena aproximación a la dispersión uniforme si **M** es grande comparado con **N**.

### 4.3.2 Desbordamiento de la tabla.

Hasta este punto se ha asumido que el tamaño **M** de la tabla dispersa será siempre suficientemente grande como para acomodar los conjuntos de datos con los que se está trabajando. En la práctica sin embargo, se debe considerar la posibilidad de una inserción en una tabla llena (un desbordamiento en la tabla). Si se está utilizando dispersión abierta, este no es habitualmente un problema dado que el tamaño total de las cadenas solo está limitado por la cantidad de memoria disponible de la zona libre. Así se restringe la exposición al desbordamiento de las tablas en la dispersión cerrada.

Se considerarán dos técnicas que evitan el problema del desbordamiento de la tabla asignando memoria adicional. En ambos casos, es mejor no esperar hasta que la tabla esté completamente llena antes de asignar más memoria; en su lugar, la

memoria será asignada cuando el factor de carga α supere cierto umbral que se denotará con α.

» **Expansión de la tabla:**

El planteamiento más simple para tratar el desbordamiento de las tablas consiste en asignar una tabla más grande cuando una inserción produzca que el factor de carga exceda y trasladar el contenido de la tabla antigua a la nueva.

Emplear esta técnica con las tablas dispersas se complica por el hecho de que el resultado de las funciones de dispersión depende del tamaño de la tabla. Esto significa que después de expandir (o contraer) la tabla, todos los elementos necesitan ser ***«re- dispersados»*** en la nueva tabla. Los costes adicionales debidos a la re-dispersión tienden a hacer que este método sea demasiado lento.

» **Dispersión extensible:**

La dispersión extensible limita los costes adicionales debido a la re-dispersión dividiendo la tabla dispersa en bloques. El proceso de dispersión se ejecuta entonces en dos pasos:

Primero se comprueban los *bits* de menor orden de una clave para determinar en que bloque será almacenado un elemento (todos los elementos de un bloque dado tendrán idénticos bits de menor orden), entonces el elemento se dispersa en un hueco particular de ese bloque utilizando los métodos comentados anteriormente. Las direcciones de estos bloques se almacenan en una tabla directorio, tal y como se muestra en la figura *4.5.* Además con la tabla se almacena un valor ***b***, que es el número de bits de menor orden a utilizar durante el primer paso del proceso de dispersión.

El desbordamiento de la tabla puede ahora ser tratado como sigue. Cuando se supera el factor de carga $\propto_{td}$ de cualquier bloque ***d***, se crea un bloque adicional ***d´*** del mismo tamaño que ***d***, y en el primer paso del proceso de dispersión los elementos que originalmente estaban en ***d*** son re-dispersados en ***d*** y ***d´*** utilizando

***b+1*** bits de orden inferior. Desde luego, el tamaño de la tabla directorio debe ser duplicado en este punto, dado que el valor de ***b*** se incrementa en uno. Este proceso se demuestra en la figura *4.5 (b)* donde el bloque $\boldsymbol{d_{00}}$ de la parte *(a)* ha sido dividido en dos bloques: $\boldsymbol{d_{000}}$ y $\boldsymbol{d_{001}}$. Téngase en cuenta que solo los elementos inicialmente almacenados en $\boldsymbol{d_{00}}$ necesitan ser re-dispersados.

Si el tamaño de los bloques se mantiene relativamente pequeño, el enfoque de dispersión extensible reducirá enormemente los costes adicionales debidos a la re-dispersión. Desde luego, esto se hace a expensas del tiempo adicional que se consume comparando los bits de orden inferior de la tabla directorio durante el primer paso del proceso de dispersión.

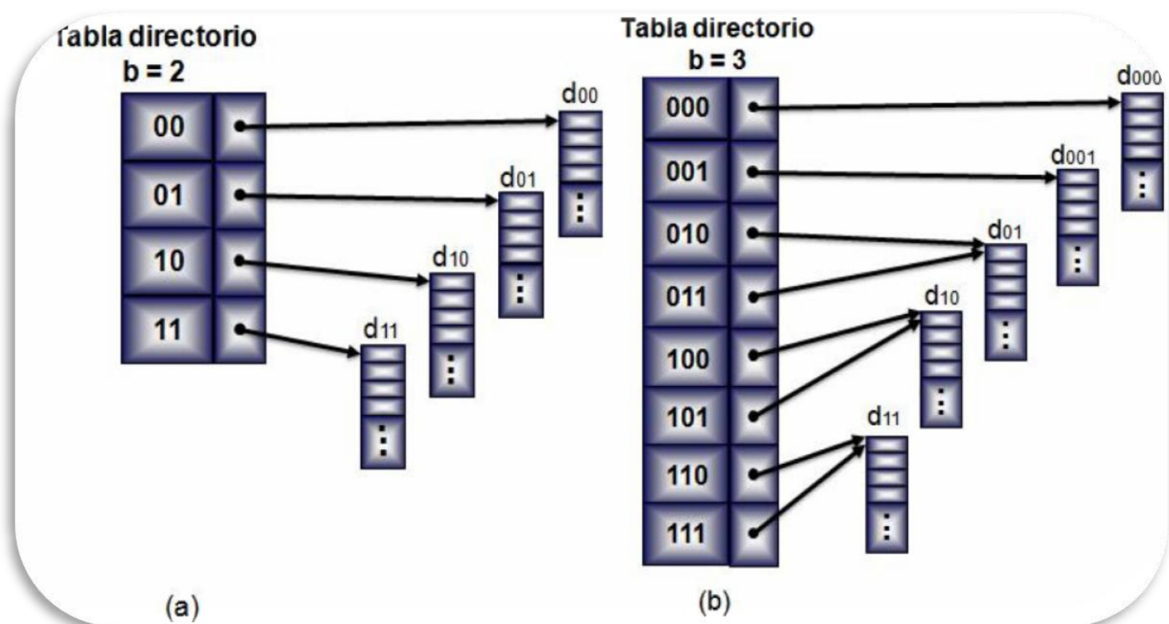


**Figura 4.5** Diagrama de la estructura de datos utilizada en la dispersión extensible.

# CAPÍTULO 5

# ESTRUCTURA DE DATOS: DCEL.

En este capítulo se muestra el concepto de la estructura de datos DCEL, conjuntamente se exponen sus ventajas, así como las aplicaciones que tiene en la vida práctica, además de mostrar el seudocódigo de alguna de sus operaciones.

**Definición 3.1:**

DCEL son las siglas inglesas de Lista de Lados Doblemente Enlazados. La estructura DCEL está formada por tres colecciones de registros:

- ✓ Lista de vértices: Un registro para un vértice ***v***, que almacena las coordenadas de ***v*** y un puntero a una arista arbitraria con origen en ***v***.

- ✓ Lista de aristas: Un registro para una arista ***a***, que almacena un puntero al vértice origen, un puntero a su arista gemela, un puntero a la cara en la que se encuentra, un puntero a la arista siguiente y otro a la anterior, todos ellos orientados en el sentido anti horario.

- ✓ Lista de caras: Un registro para una cara ***c***, almacena un puntero a un lado exterior a la cara y otro a uno interior. Se puede ver que es necesario aplicar una determinada orientación a las aristas, ya que se habla de medios-lados orientados, de esta forma una arista podría considerarse interna a una cara si está orientada en sentido anti horario con respecto a la cara.

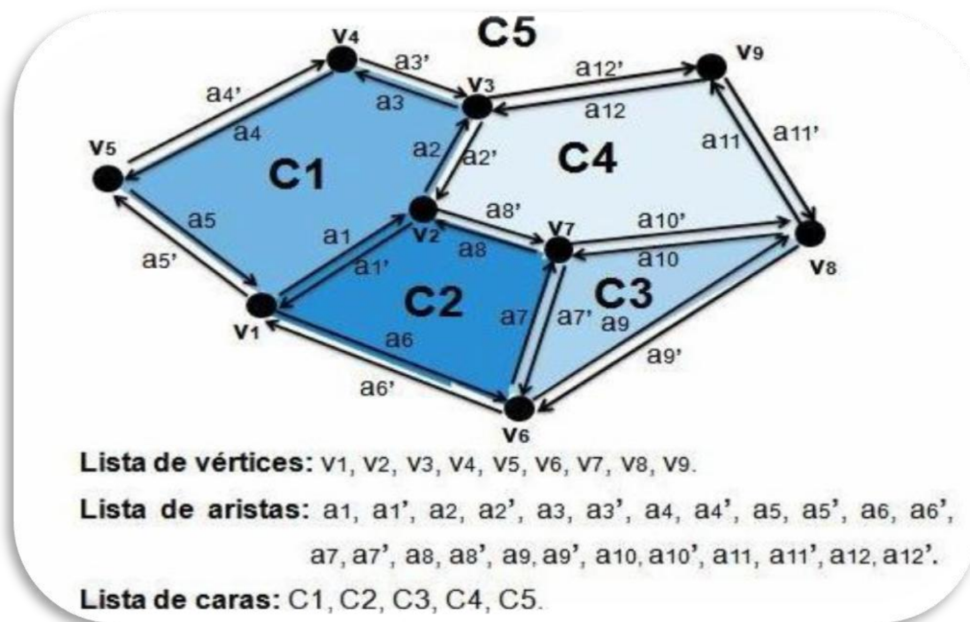


**Figura 5.1** Representación gráfica de DCEL.

**Ventajas:**

- Eficiente a la hora de obtener las siguientes características:
  - Aristas que conforman una cara.
  - Aristas que inciden en un vértice.
  - Caras adyacentes a una cara determinada.

**Aplicaciones:**

- Se puede utilizar para almacenar eficazmente información geométrica y topológica sobre superficies bidimensionales.
- Además, estas estructuras pueden almacenar otras informaciones no necesariamente topológicas, sino colores.
- Se emplean en la representación de grafos planares.
- Para calcular la intersección (o unión) de polígonos simples.
- DCEL se utiliza además para almacenar el Diagrama de Voronoi.

**Seudocódigo:**

```
****************** ******************************
*              DCEL (Lista de Lados Doblemente Enlazados)              *
************************************************************

Obtener_Aristas_de_Cara(cara):
Inicio
        lista_aristas ← ListaDE()
        arista ← Arista()
        vértice_origen ← Vértice()
         mientras arista.vértice ≠ vértice_origen:
              arista ← arista.siguiente
              lista_aristas.Adicionar(arista)
         retornar lista_aristas
Fin
.............................................................................................
Obtener_Aristas_Salientes_de_Vértice(vértice):
Inicio
        lista_aristas_salientes ← ListaDE()
        arista_inicial ← Arista()
        arista_ saliente ← Arista()
         mientras arista ≠ arista_inicial:
             arista. Arista_Jimagua(arista)
             arista_saliente ← Arista_Jimagua(arista).siguiente
              lista_aristas_salientes.Adicionar(arista)
         retornar lista_aristas_salientes
Fin
............................................................................................
Obtener_Cara_de_Arista.(arista)
Inicio
Fin
............................................................................................
Obtener_Vértices_de_Cara(cara)
Inicio
```

**Fin**

## 5.1- Observaciones Generales sobre la estructura de datos DCEL.

La lista de lados doblemente enlazados (DCEL) es una estructura clásica de la geometría computacional que permite almacenar cualquier grafo plano. Es una lista de segmentos orientados. Para cada segmento se guardan apuntadores a sus dos vértices $V_1$ y $V_2$, apuntadores al primer segmento vecino que se encuentran girando alrededor de cada uno de los vértices en sentido anti horario $A_1$ y $A_2$, opcionalmente puede contener un apuntador a cada una de las caras que comparten el segmento $C_1$ y $C_2$. Como ha de almacenar una triangulación restringida, un atributo adicional indica si el segmento es requerido o no.

La estructura de datos DCEL se emplea para almacenar el Diagrama de Voronoi. Un Diagrama de Voronoi de un conjunto de puntos en el plano, no es más que la subdivisión del mismo en regiones formadas por los lugares más próximos a cada uno de los puntos.

Son muchas las utilidades de los Diagramas de Voronoi entre las cuales se encuentran:

- Posicionamiento de torretas en telefonía móvil. La región de Voronoi de cada una de las torretas determinaría qué teléfonos deberían realizar la conexión a través de la misma.
- Control aéreo. El Diagrama de Voronoi de cada centro de control, determinaría la zona del espacio aéreo a controlar por dicha estación.
- Distribución de servicios públicos (hospitales, centros comerciales...) La ubicación de dichos establecimientos "debería ser" (al menos teóricamente) la que tenga la mayor área de región de Voronoi con respecto al resto de establecimientos del mismo tipo, para así aumentar la hipotética clientela. Como comentario a este último caso, se podría realizar una serie de modificaciones al planteamiento: por ejemplo en el caso de la ubicación de los hospitales. A un determinado enfermo siempre le interesaría acudir al

hospital más cercano a su domicilio según la distancia euclídea. Para saber a cual acudir no tendría más que mirar el Diagrama de Voronoi de los puntos que indican la posición de los hospitales y comprobar en que región se encuentra su casa. Ahora bien, también podría ser interesante considerar a ciertos hospitales como más cercanos si tienen, por ejemplo, mejor acceso (autopista...) o si tienen un mayor número de camas. En este caso cada uno de los puntos (hospitales en este caso) puede tener asignado un determinado "peso", con lo que el Diagrama de Voronoi de la nube de hospitales podría cambiar notablemente al llevarse una mayor zona de influencia (cercanía) el hospital con más peso.

# COCLUSIONES.

El objetivo por el cual se realizó la presente documentación de la Biblioteca de Estructuras de Datos Avanzadas, fue la de establecer un material de apoyo y consulta para los estudiantes de Ingeniería en Informática y de Sistemas o de cualquier otra área afín, en la cual se imparta la materia de Estructura de Datos.

Cada estructura de datos ofrece ventajas y desventajas en relación a su simplicidad y eficiencia para la realización de cada operación. Una vez que se haya terminado de revisar detenidamente este material, el usuario será capaz de establecer estructuras de datos lógicas que permitan hacer un uso más eficiente del espacio ocupado en la memoria de la computadora, de minimizar los tiempos de acceso, así como de lograr formas más efectivas de inserción y eliminación de los datos en estructuras de almacenamiento, para lograr una solución óptima.

# GLOSARIO DE TÉRMINOS Y SIGLAS

**A**

Apilar: operación en la que un elemento de datos se coloca en el lugar apuntado por el puntero de la pila, y la dirección en el puntero de la pila se ajusta por el tamaño de los datos de partida.

Arreglo: o como también se le conoce: vector, *array*, o alineación, en programación no es más que un conjunto o agrupación de variables del mismo tipo cuyo acceso se realiza por índices. Se almacenan en forma contigua en la memoria RAM y son referenciados con un nombre común y una posición relativa.

**D**

Desapilar: operación en la que un elemento de datos en la ubicación actual apuntada por el puntero de la pila es eliminado, y el puntero de la pila se ajusta por el tamaño de los datos de partida.

**E**

Estructura de datos: son colecciones de variables, no necesariamente del mismo tipo, relacionadas entre sí de alguna forma, sobre la que se han implementado las operaciones definidas para el TDA y que permite finalmente la implementación de un tipo de dato (a través de una clase).

**F**

Función hash: es una función para resumir o identificar probabilísticamente un gran conjunto de información, dando como resultado un conjunto imagen finito generalmente menor (por ejemplo un subconjunto de los números naturales). Una buena función de dispersión es una que experimenta pocas colisiones en el conjunto esperado de entrada; es decir que se podrán identificar unívocamente las entradas.

**S**

Seudocódigo: se utiliza para expresar los algoritmos en una forma que sea independiente de cualquier lenguaje de programación. El prefijo *seudo* se usa para resaltar que no se pretende que este código sea compilado y ejecutado en una computadora. La razón para usar seudocódigo es que permite transmitir en términos generales las ideas básicas de un algoritmo. Una vez que los programadores entienden el algoritmo que está siendo expresado por el seudocódigo, pueden implementarlo en el lenguaje de programación de su elección.

**T**

Tabla de símbolos: es una forma concreta del TDA Diccionario que utilizan los compiladores para mantener los distintos trozos de información simbólica que deben accederse cuando se genera un código ejecutable.

Tipo de dato abstracto (TDA) o Tipo abstracto de datos (TAD): es un modelo matemático compuesto por una colección de operaciones definidas sobre un conjunto de datos para el modelo.

# REFERENCIAS BIBLIOGRAFICAS.


1. PROGRAMACIÓN, D. D. T. D. *P2. Conferencia #1. Estructuras de Datos Lineales. Los Tipos de Datos Abstractos. El TDA Lista.* 2006-2007.

2. P2. CONFERENCIA #2. *Estructuras de Datos Lineales. Implementación del TDA Lista utilizando listas enlazadas. .* Departamento de Técnicas de Programación ed. curso: 2006-2007.

3. *ALGORITMIA.NET* de 2008]. Disponible en: http://www.algoritmia.net/articles.php?id=13.

4. P2. CONFERENCIA #3. *Estructuras de Datos Lineales.Los TDA Pila y Cola. .* Departamento de Técnicas de Programación ed. curso: 2006-2007.

5. COTA, J. M. G. *Tutorial de Estructuras de Datos (I).* Instituto Tecnológico de La Paz.: de 2008]. Disponible en: http://sistemas.itlp.edu.mx/tutoriales/estru1/index.htm.

6. *ALGORITMIA.NET* de 2008]. Disponible en: http://www.algoritmia.net/articles.php?id=14.

7. HEILMAN, G. L. *Estructuras de datos, algoritmos y programación orientada a objetos.* La Habana : Félix Varela, 2003.

8. *ALGORITMIA.NET* Disponible en: http://www.algoritmia.net/articles.php?id=15.

9. GREGORY, H. *Estructuras de datos, algoritmos y programación orientada a objetos.* La Habana: Félix Varela, 2003.

10. MICTLAN. *Página de entrenamiento para el ACM ICPC de la Universidad Tecnológica de la Mixteca. Diccionarios* de 2008]. Disponible en: http://mictlan.utm.mx/html/jaws/html/index.php?page/diccionarios.

11. FERRERA, J. C. *Diagramas de Voronoi* Facultad de Informática. Universidad Politécnica de Madrid: Disponible en: http://www.dma.fi.upm.es/mabellanas/voronoi/voronoi/voronoi.html.

**Universidad de las Ciencias Informáticas.**

**Facultad 9.**