Maestría en Tecnología de Cómputo

Curso propedéutico de Programación

Tema IV: Apuntadores y estructuras de datos dinámicas.

Conferencia No. 4

Título: Estructuras de datos dinámicas. Tipos de Datos Abstractos. Apuntadores. Manejo de memoria.

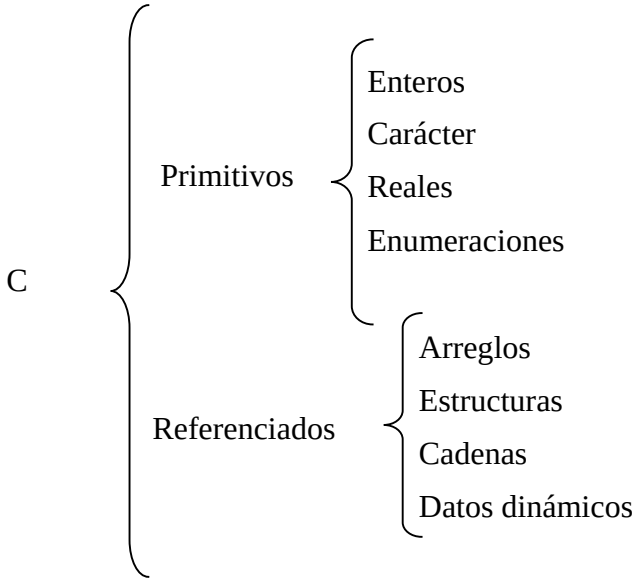
Objetivo: Identificar y utilizar las estructuras de datos dinámicas en el lenguaje C para la implementación de algoritmos sencillos que involucren colecciones dinámicas de datos.

Sumario:

Introducción

Muchas veces el concepto de tipo de dato se asocia solo con los valores que puede tomar una determinada variable, pero hay que tener en cuenta también las operaciones que se realizan con esos valores. Por ejemplo al decir en C **int b;**

Estamos diciendo que b es entero, por lo tanto puede almacenar números enteros, pero sabemos además que podemos utilizar los operadores +, -, *, /, que son entre otros los operadores que permiten trabajar con ese tipo de dato en C. Los lenguajes de programación solo proporcionan algunos tipos:



¿Se pueden resolver todos los problemas con esos tipos de datos?.

1. Tipos de datos abstractos

Un **Tipo de Dato Abstracto** (TDA), es un modelo matemático de una parte de la realidad, que tiene que cumplir ciertas condiciones para que sea considerado como un TDA.

Un TDA se define por dos partes: el **invariante** (modelo) y un **conjunto de operaciones**.

- El **invariante** es una proposición que expresa el conjunto de valores válidos del TDA. El invariante permite saber qué elementos pertenecen al TDA y qué elementos no.
- El **conjunto de las operaciones** define el comportamiento del TDA, esto es, define lo que se puede hacer o no con el TDA.

¿Qué es una instancia de TDA?

Una instancia de TDA, o simplemente instancia, es un elemento que pertenece al TDA. El TDA es abstracto mientras que la instancia es concreta.

Para facilitar la construcción de un TDA se definen etapas que son consideradas el ciclo de vida del mismo. Como vemos en la figura 1, el ciclo de vida de un TDA, se compone de tres etapas: Diseño, Implementación y Uso. La etapa de diseño se refiere al momento en que se está definiendo (concibiendo) el TDA. Y la etapa de implementación consiste en convertir la definición del TDA en un código utilizable por una aplicación. La etapa de uso es cuando se toma la librería de implementación del TDA o el paquete donde se implementó y se usa para resolver un problema en una aplicación particular.

Figura 1: Ciclo de Vida de un TDA

En la salida de cada etapa se obtienen uno o varios documentos que sirven de entrada para la siguiente, como se ve en la figura 2. En la entrada de la etapa de diseño del TDA están los requerimientos; estos es lo que se requiere que cumpla el TDA, es decir, lo que debe incorporar el TDA para que cumpla con determinada funcionalidad. Como salida de esta etapa, está el documento de diseño el cual, es una especificación de como es el TDA expresado mediante alguna formalización, por ejemplo UML. Este documento se convierte en entrada de la etapa de implementación y es una guía para el proceso de codificación del TDA que realizan los programadores. Como resultado de esta última etapa, se tiene una librería de código fuente (en C) y esta se usa en la siguiente etapa para la construcción de una aplicación o programa.

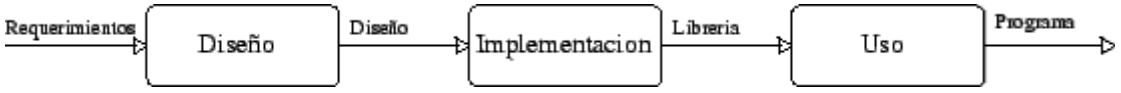


Figura 2: Ciclo de vida y su documentación.

2. Punteros

Como se mencionaba en clases anteriores, un puntero es una variable que guarda la dirección de memoria donde se almacena un cierto dato. Para declarar una variable de tipo puntero, de utiliza la siguiente sintaxis:

tipo *nombre_var_puntero

No es posible enunciar las estructuras de datos dinámicas sin el uso de variables de tipo puntero. A continuación se explican algunos elementos adicionales para el manejo de estas variables.

2.1 Operaciones con variables de tipo puntero

Las operaciones válidas con variables de tipo puntero incluyen el uso del operador dirección de (&) y el uso del operador de indirección (*). Además, se les pueden aplicar los operadores de suma y resta (solo con tipos de datos enteros), los operadores de incremento y decremento (++ y --) y los operadores de relación.

Ejemplo:

```
int x[100];

int *pa, *pb, b;

pa = &x[3];

b = *pa + 1; // el resultado es la suma del valor apuntado (x[3]) + 1

b = *(pa + 1) // el resultado es el siguiente entero apuntado por pa, es decir, x[4]

pb = &x[4];

*pb = 0 // el resultado es poner el elemento x[4] = 0
```

2.2 Punteros genéricos y puntero nulo

Los punteros genéricos son un tipo especial de punteros, denotado por void *. Un puntero a cualquier tipo de datos se puede convierte a void *.

Ejemplo:

```
#include<stdio.h>
void function(void *, int n);
main()
{
   int x[3] = {1, 2, 3};
   int *q = x;
   function(q, 100);
}
void function(void *p, int n)
{
   int *q = p;
   printf("El valor de q es: %d \n", q[0]);
}
```

Como se había visto anteriormente, la constante NULL se utiliza para identificar a un puntero nulo, es decir, un puntero que no se encuentra apuntando a una dirección de memoria.

2.3 Asignación dinámica de memoria

La asignación dinámica de memoria consiste en asignar la cantidad de memoria necesaria para almacenar un valor de una variable durante la ejecución de un programa, a diferencia de la asignación estática, que reserva el espacio durante la compilación.

Para poder realizar una asignación dinámica, es necesario declarar una variable de tipo puntero al tipo de datos que se desea asignar la memoria. Por ejemplo: si se desea asignar memoria para una matriz de enteros, es necesario declarar una variable de tipo puntero a enteros, como int *p;

La función **malloc** de la librería stdlib.h de C permite asignar una cierta cantidad de bytes consecutivos de memoria a un cierto tipo de datos. Esta función devuelve un puntero genérico (void *) que hace referencia al espacio asignado.

Ejemplo:

```
#include<stdio.h>
#include <stdlib.h>
main()
{
   int *p = NULL;
```

```
   int nbytes;
   printf("Entre la cantidad de bytes a asignar\n");
   scanf("%d", &nbytes);
   p = (int *)malloc(nbytes);
   if (p == NULL)
      printf("Insuficiente espacio de memoria\n");
   else
      printf("Se han asignado %d bytes de memoria \n", nbytes);
}
```

Es posible utilizar la función malloc para asignar espacio para datos de tamaño desconocido, como por ejemplo, datos de tipo estructura. Para ello, se calcula el tamaño del tipo de datos, mediante la función sizeof. A continuación se muestra un ejemplo.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

struct estudiante{
   char nombre[20];
   char apellidos[40];
   char sexo;
   int edad;
};

main()
{
    struct estudiante *p;
    p = malloc(sizeof(struct estudiante));
    if(p == NULL)
     printf("Error de asignacion\n");
   else
      printf("Direccion de memoria donde apunta p: %p en dec: %d \n", p, p);
   free(p);
}
```

Al contrario que la función malloc, la función **free** permite liberar un bloque de memoria. Si el puntero que se desea liberar tiene valor NULL, la función free no realiza ninguna acción.

Ejemplo:

```
#include<stdio.h>
#include <stdlib.h>
main()
{
   int *p = NULL;
   int nbytes;
   printf("Entre la cantidad de bytes a asignar\n");
   scanf("%d", &nbytes);
   p = (int *)malloc(nbytes);
   if (p == NULL)
      printf("Insuficiente espacio de memoria\n");
   else
      printf("Se han asignado %d bytes de memoria \n", nbytes);
   free(p);
   printf("Se ha liberado la memoria asignada\n");
}
```

Otra función de importancia al manejar memoria dinámica es la función **realloc**. Esta función cambia el tamaño de un bloque de memoria asignado. Su definición está dada por:

void *realloc(void *pBlomem, size_t nbytes);

El parámetro nbytes representa el número e bytes que serán asignados al bloque de memoria apuntado por pBlomem. La función realloc devuelve un puntero al espacio asignado. Dicho puntero tomará valor NULL en dos escenarios:

1. Si el valor de nbytes es cero (no hay nada para asignar)
2. Si hay suficiente memoria disponible para expandir el bloque apuntado por pBlomem (no cambia la dirección, es decir, la memoria se mantiene apuntada por pBlomem).

A continuación se ofrece un ejemplo del uso de la función realloc.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
main()
{
    char *p;
    p = malloc(21);
    if(p == NULL)
```

```
      printf("Error de asignacion\n");
   else
   {
      strcpy(p, "p tiene 15 caracteres");
      printf(p);
      printf("\n");
      p = realloc(p, 38);
         if(p == NULL)
         printf("Error de asignacion\n");
      else
      {
         strcat(p, " y ahora tiene 38");
         printf(p);
      }
   }
   free(p);
}
```

3. Estructuras de datos

Se trata de un conjunto de variables de un determinado tipo agrupadas y organizadas de alguna manera para representar un TDA. Lo que se pretende con las estructuras de datos es facilitar un esquema lógico para manipular los datos en función del problema que haya que tratar y el algoritmo para resolverlo. En algunos casos la dificultad para resolver un problema radica en escoger la estructura de datos adecuada. Y, en general, la elección del algoritmo y de las estructuras de datos que manipulará estará muy relacionada.

Según su comportamiento durante la ejecución del programa distinguimos estructuras de datos:

- **Estáticas**: su tamaño en memoria es fijo. Ejemplo: arreglos
- **Dinámicas**: su tamaño en memoria es variable. Ejemplo: listas enlazadas

4. TDA Lista Lineal

Las listas son estructuras que permiten representar un conjunto de datos de forma que se preserve la relación de orden lineal entre ellos. Una lista está compuesta por elementos o nodos que pueden contener tanto tipos de datos primitivos como referenciados.

4.1 Modelo:

*Una **lista lineal** es un conjunto de nodos x_1 , x_2 , ... x_n, cuyas propiedades estructurales esenciales incluyen sólo las posiciones lineales (unidimensionales) relativas de los nodos*

- x_i es de un determinado tipo.
- n representa el tamaño de la lista lineal.
- Si $n = 0$ entonces la lista está vacía.
- Propiedades:

  Sea $n \geq 1$

  x_1 es el primer ítem de la lista.

  x_n es el último ítem de la lista.

  x_i precede a x_{i+1} para i = 1,2, ... , n-1.

  x_i sucede a x_{i-1} para i = 2,3, ... , n.

4.2 Operaciones:

1. Crear una lista lineal vacía.
2. Insertar un nuevo ítem en determinada posición de la lista.
3. Retirar el i-ésimo ítem.
4. Localizar el i-ésimo ítem para examinar y/o alterar el contenido de sus componentes.
5. Saber si la lista está vacía.
6. Determinar el número de ítems de una lista.

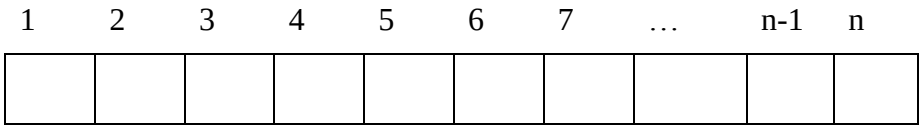
4.3 Implementación:

Los siguientes tipos de listas lineales pueden ser consideradas:

- Lista lineal secuencial (o contigua)
- Lista lineal simplemente enlazada (o encadenada)
- Lista lineal doblemente enlazada (o encadenada)

4.3.1 Listas lineales secuenciales (o contiguas)

Representación Genérica:



Características:

- Son implementadas a través de variables estáticas.

- El acceso a sus componentes es hecho de manera directa (a través de un índice).
- Las estructuras de representación que mejor permiten la implementación de listas lineales secuenciales (o contiguas) son los arreglos y archivos.
- Típicamente, a las listas lineales secuenciales se les conoce como matrices dinámicas.

Ventajas:

Además de todas las ventajas inherentes a la manipulación de variables estáticas, también tienen como característica importante el hecho de poder ser implementadas en prácticamente todos los lenguajes de programación más utilizados.

Desventajas:

También tienen todas las desventajas inherentes a la manipulación de variables estáticas, como por ejemplo, la necesidad de prefijar su tamaño inicial, la necesidad de redimensionar este tamaño prefijado en tiempo de ejecución si fuera necesario, la dificultad en realizar sobre ella ciertas operaciones (inserción y remoción, por ejemplo) etc..

Ej.: para la lista lineal secuencial que se muestra a continuación, que contiene una lista de nombres ordenados en orden alfabético, la inclusión de un nuevo nombre (Benedicto, por ejemplo), acarrearía la necesidad de desplazar todos los nombres a partir de Carlos, para poder abrir espacio para la inclusión del nuevo nombre.

1	Azucena
2	Carlos
3	Celia
4	Dalia
5	Emilio
6	Fabio
7	Francisco
8	Gustavo
9	Helio
10	Iván
11	Joaquín
12	José
13	Katia
14	Zulmira

15	

Después de la inclusión del nuevo nombre, nuevas inclusiones serían imposibles, ya que no habría espacio disponible para ello. Por ello sería necesario redimensionar el tamaño de la lista.

Implementación

Una de las formas más simples de implementación de este TDA parte del uso de un arreglo unidimensional. En este caso, todos los elementos que conforman la lista estarían almacenados en posiciones de memoria consecutivas. Por esta razón, se dice que esta implementación del TDA sigue una *disposición secuencial* en la memoria de la computadora. Si este es el caso, a la lista se le conoce como *lista secuencial*.

Una implementación de este TDA con esta disposición puede ser implementado si se le asigna un arreglo lo suficientemente grande como para mantener todos los elementos en memoria de la lista.

Para asignar dinámicamente el tamaño a una lista lineal secuencial (arreglo dinámico) se necesita utilizar la función calloc. Esta función devuelve un puntero al bloque de memoria de tamaño solicitado, o NULL si no se encuentra disponible un bloque del tamaño que se precisa. Dicha función está definida como:

void *calloc(size_t nelementos, size_t tamelem);

El parámetro nelementos especifica el número de elementos que tendrá el arreglo dinámico, mientras que el parámetro tamelem especifica el tamaño en bytes que tendrá cada uno de los elementos a almacenarse en el arreglo dinámico.

A continuación se muestra un ejemplo del uso de la función calloc.

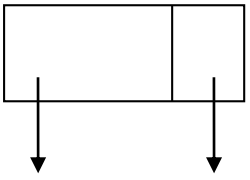
```
#include<stdio.h>
#include<stdlib.h>
main()
{
   int *p;
   p = (int *)calloc(2, sizeof(int));
   if (p != NULL)
      for (int i = 0; i < 2; i ++)
      {
         printf("Entre el valor del %d elemento: \n", i + 1);
         scanf("%d", &p[i]);
      }
   printf("Los valores son: ");
```

```
   for (int i = 0; i < 2; i ++)
      printf("%d ", p[i]);
   printf("\n");

    p = realloc(p, 3*sizeof(int));
    if(p == NULL)
      printf("Error de asignacion\n");
   else
   {
      printf("Entre el valor del 3 elemento: \n");
      scanf("%d", &p[2]);
      printf("Los valores ahora son: ");
      for (int i = 0; i < 3; i ++)
         printf("%d ", p[i]);
   }
   free(p);
}
```

4.3.2 Listas lineales simplemente enlazadas (o encadenadas)

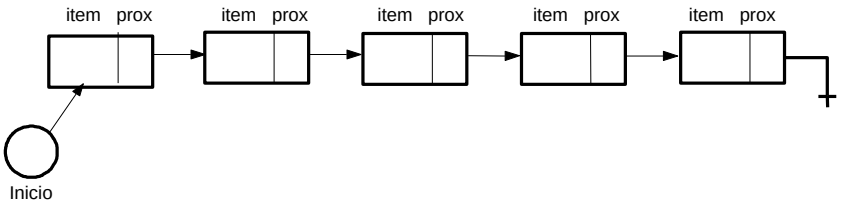
Representación Genérica:

En una lista simplemente enlazada los elementos se almacenan de forma no contigua, en vez de en un vector de posiciones de memoria consecutivas. Para conseguir esto cada elemento se almacena en un nodo que contiene la información y una referencia al siguiente elemento de la lista. Cada elemento o nodo puede ser visto como la figura que se muestra a continuación.



Información Próximo

Entonces una lista simplemente enlazada puede ser vista de esta forma, donde el inicio que apunta al primer nodo de la lista y el último nodo apunta a nulo.



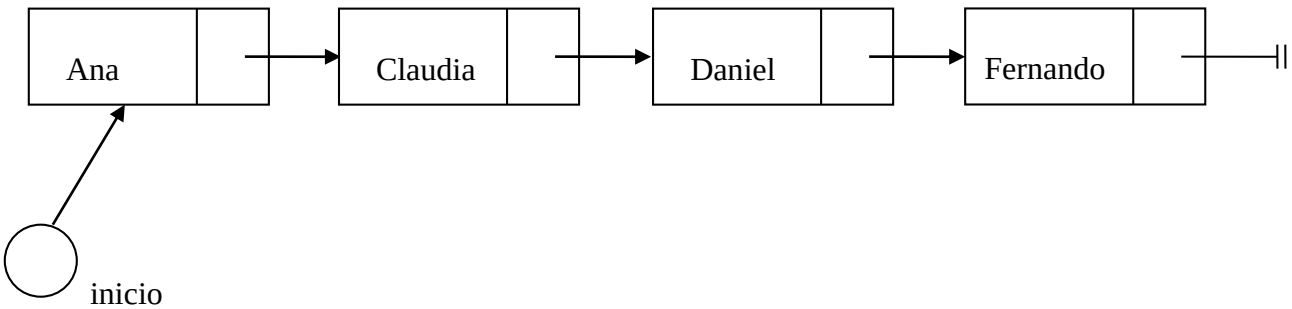
Características:

- Son implementadas a través de variables dinámicas.
- Los nodos que componen la lista deben contener, por lo menos, dos campos: un campo de tipo simple o referenciado para guardar las informaciones almacenadas en la lista y un campo de tipo puntero para guardar la dirección del nodo siguiente de la lista.
- El acceso a los nodos componentes de la lista es secuencial (para acceder al 4º elemento, es necesario pasar antes por el 3º, para acceder al 3º elemento, es necesario pasar antes por el 2º y así sucesivamente).
- Las estructuras de representación de las Listas Enlazadas deben obligatoriamente soportar conceptos como punteros y asignación dinámica.
- Debe existir, por lo menos, un puntero externo a la lista, que apunte para ella, permitiendo así accederla. En las representaciones genéricas a este puntero se le denomina *inicio o cabeza*.

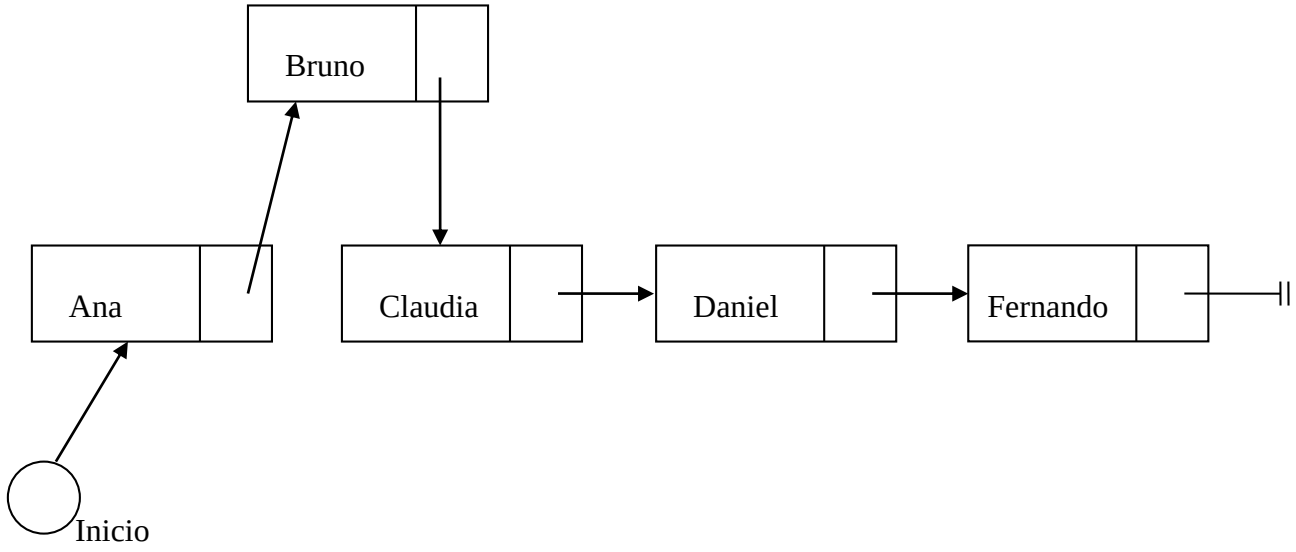
Ventajas y Desventajas: son las inherentes a la utilización de variables dinámicas, como por ejemplo:

- Mayor complejidad inherente a la manipulación de punteros (desventaja)
- El hecho de que no todos los lenguajes de programación permiten la construcción de estructuras para la representación de Listas Enlazadas (desventaja)
- La posibilidad de trabajar con listas de tamaños indefinidos, que puedan crecer y decrecer conforme la necesidad (ventaja)
- Mayor facilidad para la realización de operaciones de inserción y eliminación, que consisten básicamente en el reordenamiento de algunos punteros (ventaja).

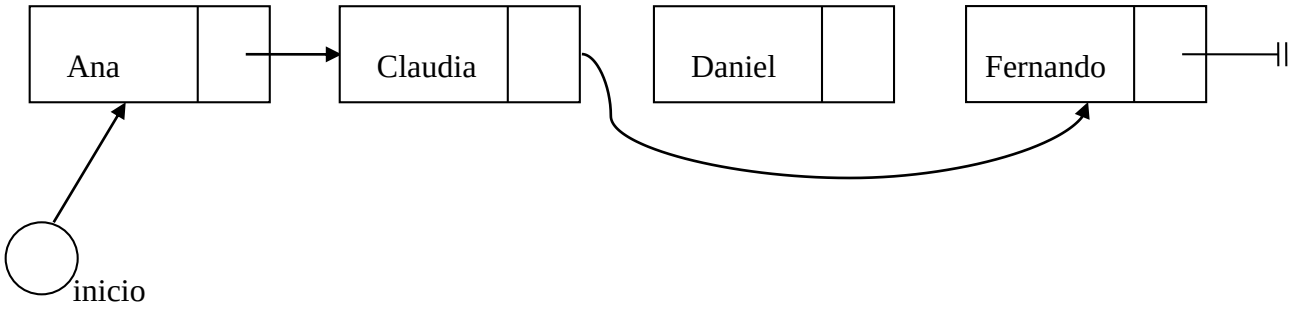
Ejemplo: suponga la lista



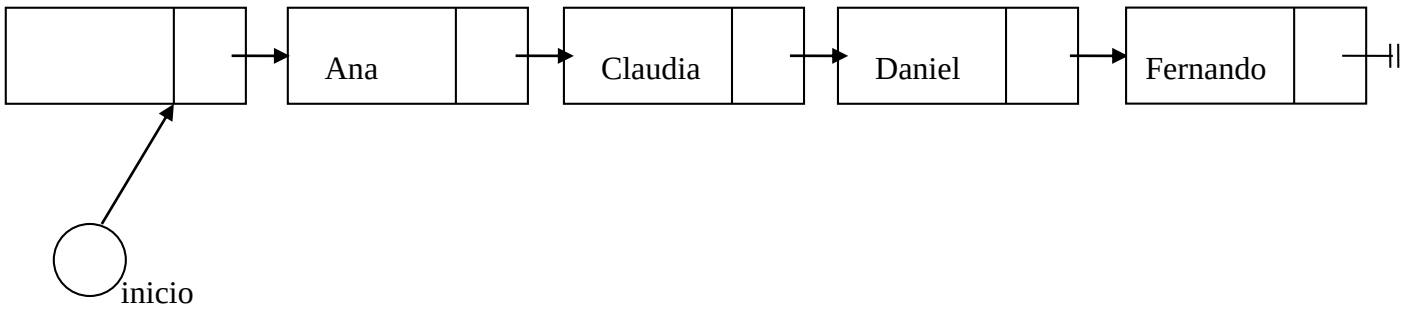
Para insertar un nuevo nodo en la lista (“Bruno”, por ejemplo), basta crear una nueva variable dinámica y reorientar 2 punteros, conforme se presenta en la figura:



Para realizar una eliminación, basta “desconectar” de la lista el elemento a ser eliminado, haciendo que el puntero del vecino (anterior) ignore el nodo a ser eliminado. La figura representa el esquema de una eliminación (ejemplo: Daniel)



Ver la lista de esta forma tiene inconvenientes a la hora de implementar algunos de sus operaciones, es por ello que en algunos libros, se considera que el primer elemento no tiene información, entonces la lista de los ejemplos anteriores se vería así:



En este caso en primer nodo de la lista se conoce como nodo cabeza.

Implementación

Para la implementación de este TDA debe implementarse primeramente la estructura Nodo_LSE y posteriormente las opciones de la estructura de datos.

```
#include<stdio.h>
#include<stdlib.h>

typedef struct s
{
   int value;
   struct s *next; //puntero al próximo elemento
} Node_List;

Node_List *first = NULL;

Node_List *new_element();
Node_List *find(int);
Node_List *get(int);

Node_List *new_element()
{
  Node_List *node = (Node_List *)malloc(sizeof(Node_List));
  return node;
}

void insert(int val)
{
  Node_List *p, *q, *n;
  p = first;
  while (p != NULL)
  {
    q = p;
    p = p->next;
  }
  n = new_element();
  n->value = val;
  n->next = NULL;
  q->next = n;
}

void delete(int val)
{
  Node_List *p, *q;

  if (first->value == val)
  {
    q = first->next;
    p = q->next;
    q->next = p;
    free(first);
    first = q;
```

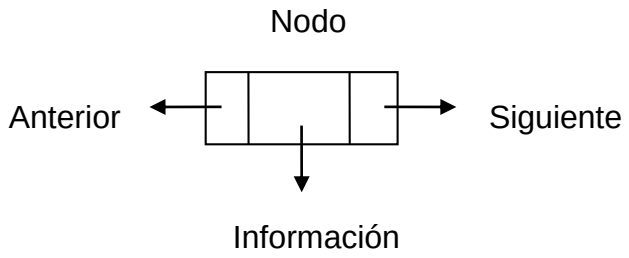
```
  }
  else
  {
    p = first;
    while (p != NULL)
    {
      q = p;
      p = p->next;
      if (p != NULL)
        if (p->value == val)
        {
          q->next = p->next;
          free(p);
          p = q->next;
        }
    }
  }
}

Node_List *find(int val)
{
  Node_List *p = NULL, *q = NULL;
  p = first;
  while (p != NULL && p->value != val)
  {
    q = p;
    p = p->next;
  }
  return p;
}

Node_List *get(int idx)
{
  Node_List *p, *q;
  p = first;
  int i = 0;
  while (p != NULL && i != idx)
  {
    q = p;
    p = p->next;
    i++;
  }
  return p;
}

int count()
{
  Node_List *p;
  p = first;
  int i = 0;
  while (p != NULL)
  {
    p = p->next;
    i++;
  }
  return i;
}
```

```
void show_all()
{
  Node_List *p;
  p = first;
  while (p != NULL)
  {
    printf("El valor es: %d\n", p->value);
    p = p->next;
  }
}

void delete_all()
{
  Node_List *p, *q;
  p = first;
  while (p != NULL)
  {
    q = p;
    p = p->next;
    free(q);
  }
  printf("\nSe han eliminado todos los elementos\n");
}

main()
{
  Node_List * node;
  first = new_element(); // se asigna la cabecera de la lista
  first->value = 1;
  first->next = NULL; // no se asigna ningún sucesor
  for (int i = 2; i < 10; i++)
    insert(i);//se introducen los valores del 1 al 9
  show_all();
  int val;
  printf("Entre el valor a buscar:\n");
  scanf("%d", &val);
  node = find(val);
  if (node != NULL)
    printf("Se ha encontrado el elemento\n");
  printf("Entre el valor a eliminar:\n");
  scanf("%d", &val);
  delete(val);
  show_all();
  delete_all();
}
```

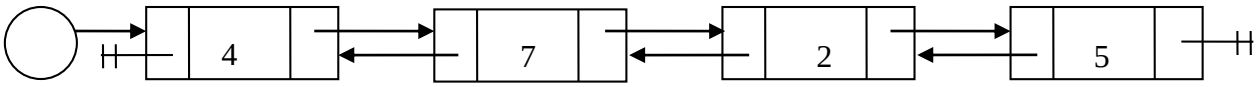
4.3.3 Listas lineales doblemente enlazadas (o encadenadas)

Representación Genérica:

Lo único que diferencia una lista simplemente enlazada de una doblemente enlazada es precisamente el nodo, que en estas últimas además del campo información tiene dos campos más uno para hacer referencia al siguiente nodo de la lista y el otro hace referencia al nodo anterior. En la figura que se muestra a continuación se puede ver la representación de un nodo de una lista doblemente enlazada.



Veamos un ejemplo de los que sería una lista doblemente enlazada de números enteros.



Como se puede observar en la lista no es necesario ya el nodo cabeza que se implementaba en la lista simplemente enlazada. Una de las ventajas de las listas doblemente enlazadas es que la implementación de las operaciones se simplifica más que en las demás listas, además de que una vez referenciado un nodo en cualquier lugar de las lista esta se puede recorrer en ambos sentidos.

5. TDA Lista Circular

Hasta ahora se han venido estudiando las distintas implementaciones del TDA lista, se estudió la lista secuencial la cual se implementa mediante arreglos estáticos y la lista simplemente enlazada que esta se hace mediante punteros, vimos las ventajas y desventajas que presenta una con relación a la otra llegando a la conclusión de que la segunda pese a sus desventajas es mejor. Una de las desventajas que presenta la implementación de la lista simplemente enlazada es que teniendo un puntero a uno de sus elementos solo se puede ver los demás elementos que le siguen a este o sea que solo se puede recorrer en un solo sentido, es por ello que surge la idea de llevar dos punteros uno al elemento anterior y otro al siguiente; así resolver el problema mediante lo que se conoce como lista doblemente enlazada, la cual es aún mejor ya que facilita la implementación y el recorrido puede hacerse en ambos sentidos.

5.1 Modelo.

Una lista circular es un TDA tal que el último elemento de la lista apunta al primero. Las listas circulares se pueden implementar usando los tres tipos vistos anteriormente:

- lista secuencial circular
- lista simplemente enlazada circular
- lista doblemente enlazada circular

5.2 Operaciones:

Las operaciones que pueden realizarse con una lista circular, son muy similares a las operaciones de una lista secuencial:

1. Crear una lista lineal vacía.
2. Insertar un nuevo ítem en determinada posición de la lista.
3. Retirar el i-ésimo ítem.
4. Localizar el i-ésimo ítem para examinar y/o alterar el contenido de sus componentes.
5. Saber si la lista está vacía.
6. Determinar el número de ítems de una lista.

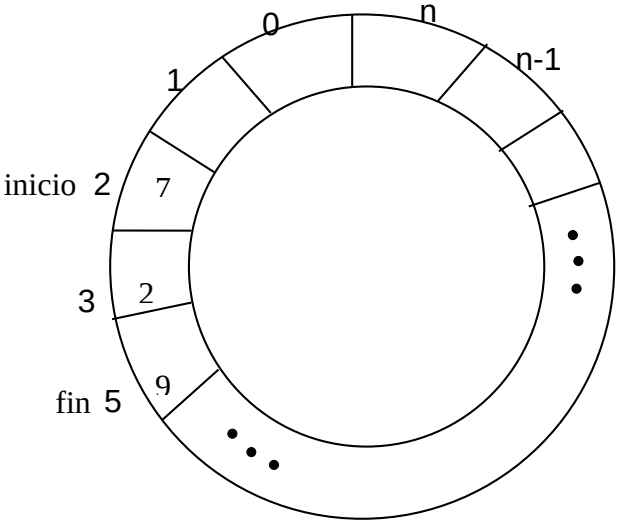
5.3 Implementación:

Los siguientes tipos de listas circulares pueden ser consideradas:

- Lista secuencial circular
- Lista simplemente enlazada circular
- Lista doblemente enlazada circular

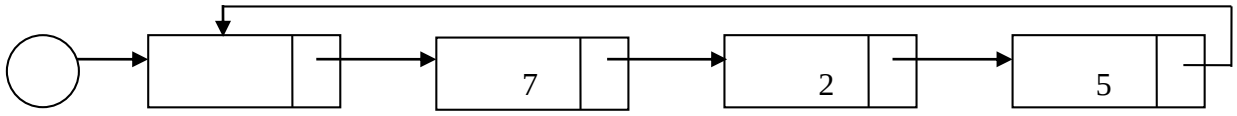
5.3.1 Lista secuencial circular

Hay que tener en cuenta que si se reserva memoria para n elementos el elemento que se encuentra en la posición n+1 es el primer elemento de la lista que estaría en la posición 0, esto se debe tener presente en el momento de implementar las operaciones de dicha lista. En la figura siguiente se muestra como se vería una lista secuencial circular. Para facilitar la implementación se llevan dos posiciones una para ubicar el inicio de la lista (donde se encuentra el 1er elemento) y otra para el final (donde se encuentra el último elemento), pues dada la circularidad se puede dar el caso de que los elementos estén en el medio de la lista como se muestra en el ejemplo de la figura siguiente.



5.3.2 Lista simplemente enlazada circular

Las listas simplemente enlazadas circulares son un poco más simples que las secuenciales ya que la memoria se reserva dinámicamente. En su diseño se cambiaría solo el último nodo de la lista, que ahora no apuntaría a null, simplemente el campo próximo del último elemento hace referencia a la cabeza de la lista. A continuación se muestra un ejemplo de una lista simplemente enlazada circular de números enteros.



A continuación se ofrecen los elementos fundamentales del código en C para el manejo de una lista simplemente enlazada circular de enteros.

La estructura que representa al nodo se mantiene idéntica a la de la lista simplemente enlazada. Y solo habría que realizarle pequeñas modificaciones a los métodos de manejo de la lista simplemente enlazada, ya que en la lista circular no existen referencias a null, estas modificaciones se pueden ver a continuación.

```
#include<stdio.h>
#include<stdlib.h>

typedef struct s
{
   int value;
   struct s *next; //puntero al próximo elemento
} Node_List;

Node_List *first = NULL;

Node_List *new_element()
{
   Node_List *node = (Node_List *)malloc(sizeof(Node_List));
   return node;
}

void insert(int val)
{
   Node_List *p, *q, *n;
   p = first;
```

```
   do
   {
      q = p;
      p = p->next;
   }  while (p != first);

   n = new_element();
   n->value = val;
   n->next = p;
   q->next = n;
}

void delete(int val)
{
   Node_List *p, *q;

   if (first->value == val)
   {
      q = first->next;
      p = q->next;
      q->next = p;
      free(first);
      first = q;
   }
   else
   {
      p = first;
      do
      {
         q = p;
         p = p->next;
         if (p != first)
            if (p->value == val)
            {
               q->next = p->next;
               free(p);
               p = q->next;
```

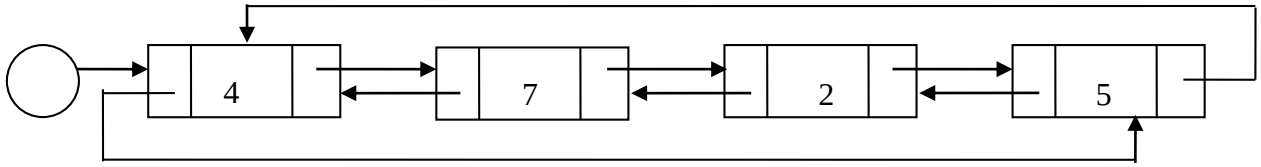
```
            }
        } while (p != first);
    }
}

Node_List *find(int val)
{
    Node_List *p, *q;
    p = first;
    while (p->next != first && p->value != val)
    {
        q = p;
        p = p->next;
    }
    return p;
}

Node_List *get(int idx)
{
    Node_List *p, *q;
    if (idx == 0) return first;
    else
    {
        int i = 0;
        p = first->next;
        while (p != first && i != idx)
        {
            q = p;
            p = p->next;
            i++;
        }
    }
    return p;
}

int count()
{
```

```
   Node_List *p;
   p = first;
   int i = 0;
   do
   {
      p = p->next;
      i++;
   } while (p != first);
   return i;
}

void show_all()
{
   Node_List *p;
   p = first;
   do
   {
      printf("El valor es: %d\n", p->value);
      p = p->next;
   } while (p != first);
}

void delete_all()
{
   Node_List *p, *q;
   p = first;
   do
   {
      q = p;
      p = p->next;
      free(q);
   } while (p != first);
   printf("\nSe han eliminado todos los elementos\n");
}

main()
{
```

```
   Node_List * node;
   first = new_element(); // se asigna el 1er elemento
   first->value = 1;
   first->next = first; // se enlaza consigo mismo
   for (int i = 2; i < 10; i++)
      insert(i);
   show_all();
}
```

5.3.3 Lista doblemente enlazada circular

La lista doblemente enlazada puede implementarse de manera circular, haciendo que el campo siguiente del último nodo apunte al primer nodo de la lista y el campo anterior del primer nodo apunte al último nodo de la lista. Como se puede ver a continuación.



6. TDA Pila.

Hasta el momento se ha estudiado en la asignatura el TDA lista y sus diversas formas de implementación. Se pueden crear condiciones especiales para la ejecución de operaciones de inserción y eliminación de elementos en una lista, las estructuras resultantes son conocidas por listas con políticas (o disciplinas) de acceso. Las principales listas con políticas de acceso son la PILA y la COLA.

Una pila es una estructura de datos en la cual el acceso está limitado al elemento más recientemente insertado. El último elemento añadido a la pila se coloca en la cima o tope, donde es accedido muy fácilmente, mientras que los elementos que llevan más tiempo son más difíciles de acceder. Por tanto, una pila es apropiada si sólo queremos acceder al elemento en la cima; el resto de los elementos son inaccesibles por el momento.

6.1 Modelo.

Una pila es una lista en la cual las operaciones de inserción y eliminación de elementos son ejecutadas solamente en uno de los extremos, denominado tope de la lista. De esta forma, el último elemento insertado es el primero a ser eliminado. Esta política de acceso es conocida por LIFO - "Last In First Out" (Último que Entra Primero que Sale).

En la práctica existen numerosos ejemplos de aplicaciones en las cuales es utilizada la disciplina de acceso LIFO; por ejemplo, en una pila de platos o de libros, la mudanza acomodada en un camión, los anillos en un dedo de la mano, etc.

6.2 Operaciones.

1. Adición de un ítem en el tope de la pila o apilar. (Push (item))
2. Eliminar un elemento del tope de la pila o desapilar. (Pop() -> item)
3. Pila vacía. (Empty () -> boolean)
4. Devolver el elemento que se encuentra en el tope de la pila en un determinado momento. (Top() -> item)

Las operaciones Top y Pop no pueden ser aplicadas en una pila vacía. En ese caso se comete el error conocido como *underflow* (tentativa de retirar un elemento de una pila vacía).

La representación de varios de los momentos de una pila es mostrada a continuación:

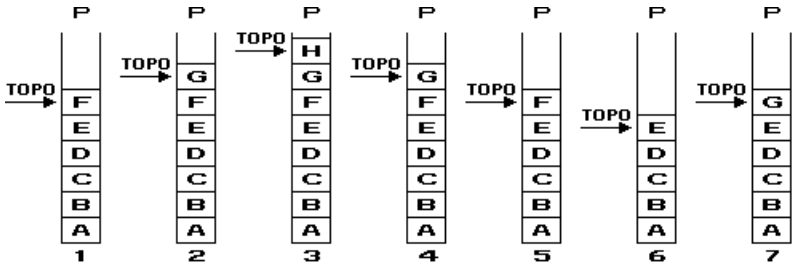


Figura 1: Representación de varios momentos de una pila

La figura comprueba la característica principal de las pilas, que es la inserción y eliminación efectuadas solamente sobre el último elemento (tope de la lista). Es interesante notar que no es posible diferenciar la pila en los instantes 1 y 5 ó 2 y 4; ya que en estos instantes, la pila contiene ítems idénticos, apilados en el mismo orden y con topes idénticos.

Esto sucede debido a que la estructura pila no mantiene ningún registro de los elementos que le son insertados o eliminados.

En la figura anterior, los 7 momentos serían representados de la siguiente manera, utilizando las operaciones apilar y desapilar:

1 - apilar(a), apilar(b), apilar(c), apilar(d), apilar(e), apilar(f)

2 - apilar(g)

3 - apilar(h)

4 - desapilar()

5 - desapilar()

6 - desapilar()

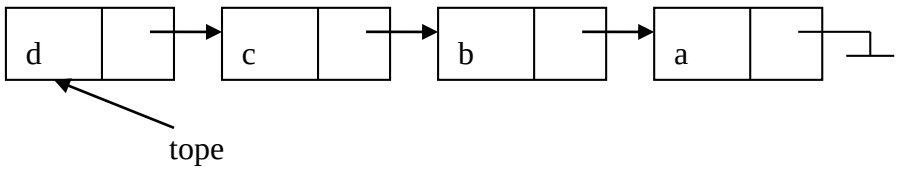
7 - apilar(g)

6.3 Implementación.

Una pila se puede representar a través de la utilización de cualquier implementación del TDA lista estudiado en clases anteriores:

- Pila mediante lista secuencial es vista como en la figura 1.
- Pila mediante lista lineal simplemente enlazada.

Una pila puede implementarse mediante una lista enlazada, en la que el tope de la pila está representado por el primer elemento de la lista.



Para implementar la operación apilar, creamos un nuevo nodo y lo incluimos en la lista como primer elemento. Para implementar el método desapilar, basta con desplazar la cima de la pila al segundo elemento de la lista (si es que existe).

La pila vacía se representa mediante la lista vacía.

Cada operación es de orden O(1), ya que solo actúan sobre el tope de la lista, por tanto, los cálculos son independientes del tamaño de la lista.

Hasta aquí hemos visto los diferentes métodos que precisa un TDA Pila para mantener su política de acceso. Podemos implementar la Pila mediante la lista enlazada que ya está implementada, el código en c se puede ver a continuación.

```
#include<stdio.h>
#include<stdlib.h>

typedef struct s
{
   int value;
   struct s *previous; //puntero al próximo elemento
} Node_List;

Node_List *top = NULL;

Node_List *new_element()
{
   Node_List *node = (Node_List *)malloc(sizeof(Node_List));
   return node;
}

void Push(int val)
{
   Node_List *p, *n;
```

```
   p = top;
   n = new_element();
   n->value = val;
   n->previous = p;
   top = n;
}

void Pop()
{
   Node_List *p, *q;

   if (top != NULL && top->previous != NULL)
   {
      q = top->previous;
      p = q->previous;
      q->previous = p;
      free(top);
      top = q;
   }
   else
   {
      free(top);
      top = NULL;
   }
}

int Empty()
{
   if (top == NULL)
      return 1;
   else
      return 0;
}

int count()
{
   Node_List *p;
   p = top;
   int i = 0;
   while (p != NULL)
   {
      p = p->previous;
      i++;
   }
   return i;
}

void show_all()
{
   Node_List *p;
   p = top;
   while (p != NULL)
   {
      printf("El valor es: %d\n", p->value);
      p = p->previous;
```

```
   }
}

void delete_all()
{
   Node_List *p, *q;
   p = top;
   while (p != NULL)
   {
      q = p;
      p = p->previous;
      free(q);
   }
   printf("\nSe han eliminado todos los elementos\n");
}

main()
{
   Node_List * node;
   top = new_element(); // se asigna el 1er elemento
   top->value = 1;
   top->previous = NULL;
   for (int i = 2; i < 10; i++)
      Push(i);
   show_all();
   printf("La cantidad es: %d\n", count());
   Pop();
   show_all();
   delete_all();
}
```

7. TDA Cola

Donde quiera que se oferten recursos o servicios y existan más de una entidad (personas, procesos en memoria, documentos a imprimirse…) existe la posibilidad de que se formen colas. Por ejemplo cuando vamos al comedor a almorzar. Lo normal que debe de suceder en una cola es que los que hacen la cola (personas, documentos enviados a la impresora) sean atendidos en el orden en que solicitan el servicio o recurso. Por lo tanto tiene que existir una disciplina y un control para que la cola "camine" y nos pueda llegar el turno.

7.1 Modelo.

Una cola es un TDA en el cual los elementos pueden ser eliminados solamente por uno de los extremos, denominado inicio o cabeza de la cola, y los elementos solamente pueden ser insertados en el otro extremo, denominado fin de la cola.

La política de acceso que rige la inserción y eliminación de elementos en una cola es denominada FIFO "First In First Out" (Primero que entra primero que sale).

7.2 Operaciones.

1. EnQueue(x): inserta x en la cola, siempre en un extremo (al final de la cola).
2. DeQueue(): eliminar un elemento, siempre en el otro extremo de la cola (al inicio de la cola) y devuelve el elemento que elimino.
3. Front(): devuelve el elemento que está en la cabeza de la cola (siempre el que llegó primero, se encuentra en el inicio de la cola.
4. Empty(): devuelve si la cola está vacía o no

7.3 Implementación.

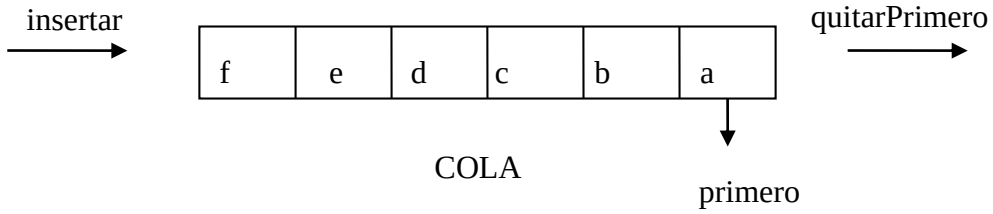
Al igual que el TDA Pila las colas también pueden implementarse usando arreglos y listas enlazadas puesto que son tipos especiales de listas con políticas de acceso diferentes.

Fundamentalmente, se consideran dos tipos de colas:

- Cola simple
- Cola con prioridad

7.3.1 TDA Cola simple

La figura siguiente representa el modelo de una cola simple: la entrada se realiza con insertar, la salida con primero y la eliminación con quitarPrimero.



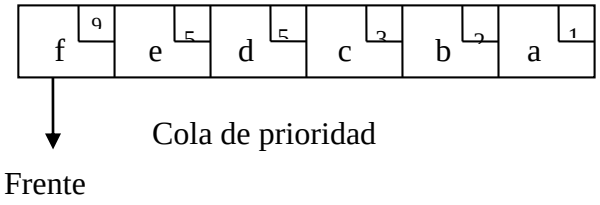
Por definición, no existe limitación del número de elementos que una cola puede contener, por tanto, la operación de EnQueue() siempre puede ser realizada.

Sin embargo, la práctica puede ser diferente, dependiendo de la estructura de datos utilizada para representar la cola.

7.3.2 TDA Cola con prioridad

Existen ejemplos de aplicaciones en las que el orden de llegada no es el orden en que se procesan los elementos, pero también es necesario priorizar a determinados elementos, para ello se definen las Colas de Prioridad. Por ejemplo, en el caso de atención médica en un hospital, los pacientes hacen una cola para hacer atendidos, pero esa atención también

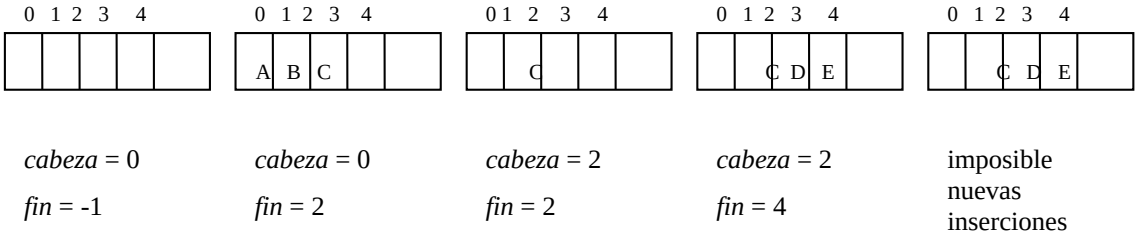
depende de la gravedad del paciente, aquellos con mayor gravedad son atendidos con prioridad. Para que esto ocurra habría que hacerle algunas modificaciones al TDA cola. En el diseño de la cola de prioridad a los elementos de las colas normales habría que adicionarle un campo que definiría la prioridad del mismo, y se organizarían de acuerdo a ese campo, e sea que a la hora de insertar un nuevo elemento se tendría en cuanta la prioridad para ver en qué posición estaría y siempre se elimina el elemento que se encuentra al frente. En la figura siguiente se puede observar cómo se vería una cola de prioridad, note que en este caso la prioridad es representada por un número entero y que entre mayor sea este mayor es la prioridad de ese elemento.



7.3.3 Implementación a través de un arreglo

La forma más sencilla de implementar una cola consiste en almacenar sus elementos en un vector, colocando el elemento en *cabeza* en la primera posición (es decir, en el índice 0)

A pesar de ser simple, la estructura presenta un serio problema. Una de las dos operaciones de EnQueue (insertar) o DeQueue (eliminar) elementos necesita hacer corrimientos de los elementos en el arreglo o quedan espacios vacíos en el arreglo. En la figura siguiente se presentan diferentes momentos de una cola en la que se han hecho inserciones y eliminaciones:

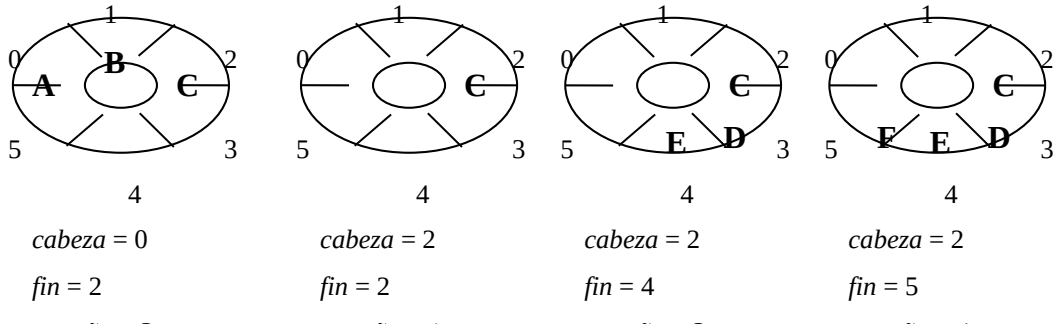


No podemos añadir más elementos, aunque la cola (más exactamente, el vector que la sustenta) no está realmente llena. Tal y como muestra la figura hay espacio libre en la cola; todas las posiciones anteriores a *cabeza* están libres y se pudieran reciclar.

Una posible solución para este problema sería, cada vez que se realice una eliminación, realizar un corrimiento de todos los elementos de la cola, de forma tal que la *cabeza* se mantenga en la primera posición del arreglo, pero esto tiene la desventaja de que cada operación DeQueue tendría un costo O(n).

7.3.4 Implementación a través de un arreglo circular

La solución más eficiente para este problema es simular que el arreglo sea circular, en vez de lineal; o sea, aun cuando la última posición esté ocupada, nuevos elementos pueden ser insertados en el comienzo del array, siempre que exista espacio disponible. Por ejemplo:

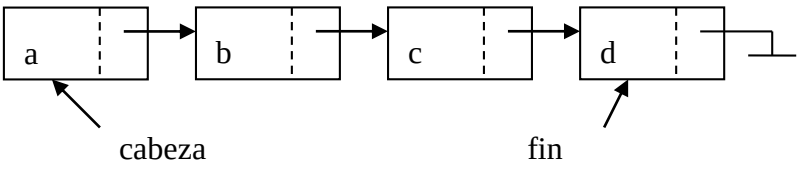


A partir del último caso se puede insertar un nuevo elemento, que ocupará la primera posición del vector, tomando *fin* el valor 0. Sin embargo, esta nueva forma de guardar los elementos en el array no permite identificar con claridad cuando la cola está vacía, ya que la comparación "fin < cabeza" ya no nos dice nada.

Para solucionar este problema, se debe incluir un nuevo campo en la declaración de la cola, responsable por el control de la cantidad de elementos guardados en ella. Este nuevo campo, denominado *tamaño*, es aumentado o disminuido en 1, siempre que un elemento es incluido o excluido de la cola. Si *tamaño* es igual a 0, entonces la cola está vacía. Análogamente, si *tamaño* es igual al número máximo de elementos del array, entonces la cola está llena.

7.3.5 Implementación con lista enlazada.

Las colas pueden implementarse empleando una lista enlazada, siempre que se mantengan referencias que apunten al principio y al final de la lista, donde siempre inserto después del fin y siempre elimino por la cabeza.



A continuación se muestra un ejemplo de código C para la implementación de una cola simple utilizando listas simplemente enlazadas.

```
#include<stdio.h>
#include<stdlib.h>

typedef struct s
```

```
{
   int value;
   struct s *next; //puntero al próximo elemento
} Node_List;

Node_List *front = NULL;
Node_List *last = NULL;

Node_List *new_element()
{
   Node_List *node = (Node_List *)malloc(sizeof(Node_List));
   return node;
}

void EnQueue(int val)
{
   Node_List *n;

   n = new_element();
   n->value = val;
   n->next = NULL;
   last->next = n;
   last = n;
}

Node_List *DeQueue()
{
   Node_List *p, *q, *n;
   if (front == NULL || front->next == NULL)
   {
      free(front);
      front = NULL;
      return front;
   }
   else
   {
      n = new_element();
```

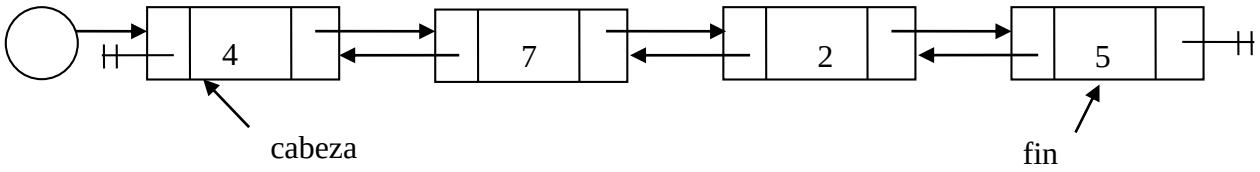
```
      n->value = front->value;
      n->next = front->next;

      q = front->next;
      p = q->next;
      q->next = p;
      free(front);
      front = q;

      return n;
   }
}

int Empty()
{
   if (front == NULL)
      return 1;
   else
      return 0;
}

int count()
{
   Node_List *p;
   p = front;
   int i = 0;
   while (p != NULL)
   {
      p = p->next;
      i++;
   }
   return i;
}

void show_all()
{
   Node_List *p;
```

```
   p = front;
   while (p != NULL)
   {
      printf("El valor es: %d\n", p->value);
      p = p->next;
   }
}

void delete_all()
{
   Node_List *p, *q;
   p = front;
   while (p != NULL)
   {
      q = p;
      p = p->next;
      free(q);
   }
   printf("\nSe han eliminado todos los elementos\n");
}

main()
{
   Node_List * node;
   front = new_element(); // se asigna la cabecera de la cola
   front->value = 1;
   front->next = NULL; // no se asigna ningún sucesor
   last = front;
   for (int i = 2; i < 10; i++)
   {
      EnQueue(i);//se introducen los valores del 1 al 9 a la cola
      printf("El ultimo elemento es: %d\n", last->value);
   }
   show_all();
   node = DeQueue();
   printf("Se ha quitado de la cola el elemento de valor: %d\n", node->value);
   show_all();
```

```
   delete_all();
}
```

7.3.6 Implementación con lista doblemente enlazada.

Es muy similar a la implementación mediante lista simplemente enlazada.



De forma general la implementación de la cola puede hacerme mediante cualquier tipo de listas vistas anteriormente y al igual que la pila podemos reutilizar el código de las mismas.

Conclusiones

En esta conferencia se estudiaron las estructuras de datos dinámicas en C y los aspectos más importantes de su manipulación. Como se pudo observar durante el transcurso de la clase la implementación del TDA lista simplemente enlazada tiene ventajas sobre las secuenciales, una de ellas es que no hay que reservar espacio en memoria para almacenar sus elementos, ya que esto se hace de forma dinámica, por otra parte aunque el tratamiento de punteros es nuevo y un poco complejo, las operaciones de inserción y eliminación computacionalmente mejoran el rendimiento, ya que en el caso de las listas secuenciales había que estar corriendo todos sus elementos.

Como se puedo ver en la conferencia, la implementación de la lista circular puede realizarse usando cualquiera de los tres tipos de listas vistas en clases anteriores (secuencial, simplemente enlazada y doblemente enlazada). Y durante la implementación también nos pudimos percatar de una de las ventajas que tiene la lista circular que es que no hay que recorrerla completa para llegar al último elemento.

También se ofrecieron variantes de implementación de las listas enlazadas con políticas de acceso (Pilas y Colas), mediante el uso de listas simplemente enlazadas.