AAR2805

TRAB
II 2008

UNIVERSIDAD CATÓLICA ANDRÉS BELLO
Urb. Montalbán – La Vega – Apartado 29086
Teléfono: +58212 – 4074407 Fax: +58212 – 4074446
Caracas, 1021 – Venezuela

Facultad de Ingeniería
Escuela de Ingeniería Informática

Manejo dinámico de memoria utilizando lenguaje C

Apuntadores, arreglos dinámicos , listas, pilas y colas

Trabajo de Ascenso para optar a la categoría de Profesor Asistente de la Escuela de Ingeniería Informática

Ing. Ricardo Casanova N.

2008

Índice general

Contenido		Página
Introducción	.....	3
Capítulo 1. Apuntadores y estructuras dinámicas		
1.0 Introducción	.....	7
1.1 Declaración de apuntadores	.....	8
1.2 Apuntadores como parámetros de función	.....	10
1.3 Apuntadores y arreglos dinámicos	.....	12
Repaso	.....	16
Capítulo 2. Listas		
2.0 Introducción	.....	18
2.1 Definición de un nodo	.....	20
2.2 Funciones para el manejo de una lista	.....	22
Repaso	.....	33
Capítulo 3. Pilas		
3.0 Introducción	.....	35
3.1 Definición de un nodo	.....	35
3.2 Funciones para el manejo de una pila	.....	37
Repaso	.....	44
Capítulo 4. Colas		
4.0 Introducción	.....	46
4.1 Definición de un nodo	.....	46
4.2 Funciones para el manejo de una cola	.....	48
Repaso	.....	57

Índice de figuras

Figuras		Página
Figura 1. Relación entre C y C++	.....	5
Figura 2. Relación entre ptrVar y var	.....	7
Figura 3. Representación de las listas estáticas utilizando arreglos	.....	18
Figura 4. Representación de las listas dinámicas utilizando una lista simplemente enlazada	.....	19
Figura 5 Representación de las listas dinámicas utilizando una lista doblemente enlazada	.....	19
Figura 6. Representación de las listas dinámicas utilizando una lista circular	.....	19
Figura 7. Tabla comparativa entre las lista estáticas y las listas dinámicas	.....	20
Figura 8. Representación de una lista simplemente enlazada en C	.....	21
Figura 9. Representación de una pila dinámica en C	.....	36
Figura 10. Representación de una estructura dinámica tipo cola en C	.....	47

INTRODUCCIÓN

int j, i++, do {...}while (x<=3), for(i=0 ; i<n ; i++). Resulta común encontrar este tipo de instrucciones en los lenguajes de programación más utilizados hoy en día: C++, Java, C#, etc. Todos ellos, de alguna u otra manera, deben mucha de su estructura y sintaxis a uno de los lenguajes de programación más completos que se hubiera creado en el siglo XX: Lenguaje C. Nacido de los lenguajes BCPL y B (de ahí su nombre), C se ha mantenido en los últimos años como uno de los lenguajes de programación más usados a nivel mundial, abarcando diferentes áreas del mundo de la computación: desarrollo de aplicaciones de escritorio, sistemas operativos, sistemas para la banca, puntos de venta (POS), comunicación de datos, etc.

Fue desarrollado por en los años setenta, en los laboratorios de Bell Telephone. No fue sino hasta 1.978 cuando Brian Kernighan y Ritchie publican una descripción definitiva del lenguaje. Dicha definición se denomina frecuentemente "K&R C". Al inicio, a pesar de las muchas características deseables del lenguaje, se presentaron incompatibilidades entre las diferentes implementaciones de C. Con el correr de los años, el Instituto Nacional Americano de Estándares (comité ANSI X3J11) ha desarrollado una definición estandarizada del lenguaje que ha sido adoptada por la mayoría de los compiladores e intérpretes comerciales de C, haciéndolo un lenguaje de programación portable, es decir, que funciona independientemente del compilador, intérprete o sistema operativo (obviando los comandos que desde C puedan ser invocados para cada uno de ellos).

Esta guía ha sido escrita con la intención de servir como un mecanismo amigable y sencillo para aprender todo lo relativo al manejo de memoria dinámica en C de forma estructurada, asumiendo un conocimiento básico del lenguaje. Se ha dividido la guía en capítulos muy bien definidos, los cuales llevan paso a paso al lector para adentrarlo, casi sin darse cuenta, en los detalles más recónditos del manejo de estructuras dinámicas en C. Se espera que una vez culminada la guía, el lector pueda utilizarla de forma práctica como manual de referencia para consultas futuras.

La guía viene acompañada de un CD, en el cual pueden encontrarse todos los códigos fuente de cada uno de los ejemplos tratados. Cada uno de ellos fue desarrollado utilizando el ambiente de desarrollo .NET 2005 bajo el estándar ANSI C, por lo que deberían poder ser ejecutados bajo cualquier compilador para ambiente Windows.

Adicionalmente, a lo largo de la guía es posible encontrar las siguientes secciones:

AL PROGRAMAR

Incluye tips de programación, producto del desarrollo de innumerables códigos para lenguaje C, así como de toda la experiencia profesional acumulada a lo largo de más de cuatro años trabajando en el lenguaje de programación.

REPASO

Luego de cado capítulo, se encuentra ésta sección, la cual busca reforzar lo aprendido a través de preguntas, reforzamiento de los conceptos, etc.

C Y C++

Muchas personas suelen escuchar de la existencia de C y C++ y en muchos casos, es posible llegar a confundir ambas cosas, asumiendo que se trata del mismo lenguaje. Pero pensar esto es un grave error. Como se dijo anteriormente, C fue creado en la década de los setenta en los laboratorios de la compañía Bell. En los años ochenta y basándose en la definición de C, Bjarne Stroustrup desarrolló lo que se conocería como lenguaje C++. Entre algunas de las mejoras que incluye C++ está lo que se conoce en el mundo computacional como el paradigma de Programación Orientada a Objetos (POO). En la siguiente figura es posible ver la relación existente entre C y C++:

Creado en la década de los 80

Figura 1. Relación entre C y C++
Fuente: elaboración propia

Esto permite visualizar lenguaje C como un subconjunto de C++, o lo que es lo mismo, todo lo que se pueda programar en lenguaje C puede ser hecho de la misma manera en C++, pero no aplica igual en caso contrario, debido a que existen librerías, palabras reservadas o estructuras en C++ que no existen en lenguaje C.

CAPÍTULO 1

Apuntadores y estructuras dinámicas

Contenido

1.0 Introducción
1.1 Declaración de apuntadores
1.2 Apuntadores como parámetros de función
1.3 Apuntadores y arreglos dinámicos

1.0 Introducción

Un apuntador (conocido también como puntero) puede ser visto como una variable que representa la posición (en lugar del valor) de otro tipo de dato, bien sea una variable o un conjunto elementos de un mismo tipo (arreglos).

Operador de dirección e indirección:

Supogamos el siguiente ejemplo: var es una variable que representa el valor de un determinado tipo de dato. El compilador asignará celdas de memoria para almacenar el dato var. La dirección de la celda de memoria donde se encuentre alojado el dato podrá ser determinada gracias al operador monario &, conocido como operador de dirección, de la siguiente manera:

&var

Asignemos ahora la dirección de var a otra variable, llamada ptrVar, de la siguiente manera:

ptrVar = &var

De ésta manera podemos decir que ptrVar es un apuntador a var, puesto que "apunta" a la dirección de memoria donde se almacena var. La relación entre var y ptrVar puede apreciarse en la Figura 2.

Figura 2. Relación entre ptrVar y var
Fuente: elaboración propia

El valor de var (el valor contenido en la dirección de memoria de var), puede ser obtenido mediante el operador monario *, conocido como operador de indirección, de la siguiente manera:

***ptrVar**

De ésta manera, *ptrVar (el valor contenido en la dirección de memoria almacenado en ptrVar) y var representan el mismo dato. Si se realizara lo siguiente:

ptrVar = &var
auxiliar = *ptrVar

var y auxiliar deben representar el mismo dato (asumiendo que auxiliar y var son variables de un mismo tipo).

1.1 Declaración de apuntadores

Los apuntadores, al igual que cualquier otra variable, deben ser declarados en C antes de ser utilizados. La sintaxis para declarar un apuntador es la siguiente:

TipoDeDatoAApuntar *nombreDelApuntador;

Donde TipoDeDatoAApuntar es alguno de los tipos de datos definidos en el lenguaje (int, char, float, etc) o incluso algún tipo de dato definido por el programador. Éste es el tipo de dato hacia el cual se apuntará.

También es posible definir apuntadores a un mismo tipo de dato en una sola línea de la siguiente manera:

TipoDeDatoAApuntar *nombreDelApuntador1,*nombreDelApuntador2,..., *nombreDelApuntadorN;

AL PROGRAMAR

*Nunca confundir el símbolo * al momento de la declaración de apuntadores con el operador monario * que retarna el valor almacenado en una dirección de memoria. El primero simplemente se utiliza como síntaxis en C para diferenciar la declaración de apuntadores de la declaración de variables tradicionales*

Ejemplo 1.1 *Aplicación básica en C para visualizar el manejo de apuntadores:*

```
#include <stdio.h>

void main()
{
        int aux = 3;
        int var;
        int *ptrAux;    [illegible]tador [illegible] entero
        int *ptrVar;    [illegible] entero

        ptrAux = &aux; /* asig[illegible]
        var = *ptrAux; /* [illegible]
        ptrVar = &var; /* asigna [illegible]

        printf("\naux=%d [illegible]aux=%X ptrA[illegible] ptrAux=%d" aux,&aux,ptrAux,*ptrAux);
        printf("[illegible] &var=%X pt[illegible] *ptrVar=%[illegible] var,&var,ptrVar,*ptrVar);
}
```

En este caso, puede verse como se definen cuatro variables (dos enteras y dos apuntadores al tipo de dato entero). El resultado en memoria pudiera ser representado de la siguiente manera:

0x4fffd34			174341E
326822D8	int aux = 3		5DCF4E0
3279E333		int var	5EE9B7F
8AB1DBB	int *ptrVar		1C8F1CC
89AFE009		int *ptrAux	7427CFF
0x65388E			24B6F65

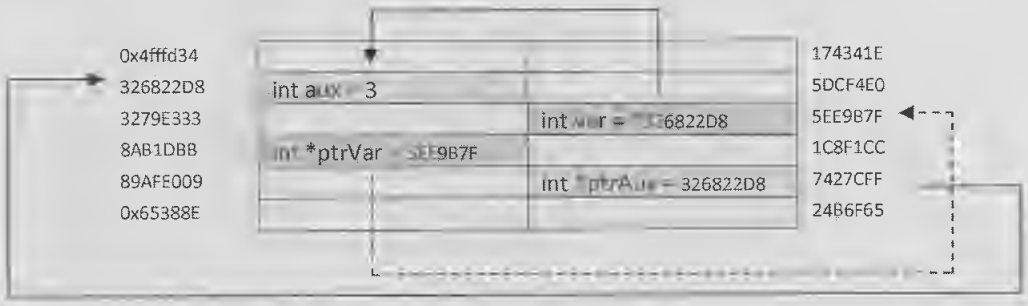
Dichas variables ocuparán posiciones de memoria, las cuales serán asignadas por el sistema operativo. De ésta manera podemos decir que según el esquema descrito arriba, la variable aux está almacenada en la dirección de memoria 326822D8 (recordar que las direcciones de memoria de la máquina son manejadas en valores hexadecimales), var ocupa la dirección 5EE9B7F, ptrAux la dirección 7427CFF y ptrVar la 8AB1DBB.

Al ejecutar las instrucciones:

```
ptrAux = &aux;
var = *ptrAux;
ptrVar = &var;
```

Se obtiene lo siguiente:

0x4fffd34			174341E
326822D8	int aux = 3		5DCF4E0
3279E333		int var = [illegible]6822D8	5EE9B7F
8AB1DBB	int *ptrVar = 5EE9B7F		1C8F1CC
89AFE009		int *ptrAux = 326822D8	7427CFF
0x65388E			24B6F65



Con lo cual ptrAux contendrá la dirección de memoria de aux (`ptrAux = &aux`), var contendrá el valor de la dirección de memoria que contiene ptrAux (`var = *ptrAux`) y ptrVar contendrá la dirección de memoria donde está almacenada la variable var (`ptrVar = &var`).

Al ejecutar las dos últimas instrucciones del ejemplo:

```
printf('\nau[illegible]X [illegible]ptrAux=[illegible]",aux,&aux,ptrAux,*ptrAux);
printf('\nva[illegible]",var,&var,ptrVar,*ptrVar);
```

se obtendrá una salida como la que sigue:

aux=3	&aux=326822D8	ptrAux=326822D8	*ptrAux=3
var=3	&var=5EE9B7F	ptrVar=5EE9B7F	*ptrVar=3

1.2 Apuntadores como parámetros de función

Una de las principales utilidades de la utilización de apuntadores radica en el pase de parámetros a una función. Como es bien sabido, C permite que una función devuelva un único valor. El pasar apuntadores como parámetros a una función ofrece la posibilidad de manejar lo que se conoce en el mundo de la programación como parámetros por valor y parámetros por referencia. Se dice que un parámetro es pasado por valor cuando cualquier modificación que se realice sobre dicho parámetro es válida sólo en el entorno local de la función o procedimiento que lo utiliza. En el caso de un parámetro pasado por referencia, cualquier modificación realizada sobre el parámetro no sólo será válida en el entorno local de la función o procedimiento que lo utiliza, sino en todo el programa.

Para ilustrar ambos casos, veamos el siguiente ejemplo.

Ejemplo 1.2 *Cómo pasar parámetros por valor y por referencia en C utilizando para ellos apuntadores:*

```
/* Se incluyen las librerías [illegible]
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* Se define una constante TAM con [illegible]
#define TAM    4

[illegible] func1 [illegible]
[illegible]
[illegible]ficación de [illegible]
[illegible]
void func1(int u, int v)
{
        u = 0;
        v = 0;

        return;
}
```

```
[illegible] Ésta [illegible]ecibe como [illegible] dos
[illegible] (.ca[illegible] dentro [illegible]
[illegible]r referencia, por lo que l[illegible]
[illegible] car[illegible] de l[illegible])

[illegible]d func2(int *pu, int *pv)
{
        *pu = 0,
        *pv = 0,

        retur[illegible]


//-
[illegible] Ésta [illegible]ción [illegible] como par[illegible]
[illegible], que [illegible] dentro [illegible] la mi[illegible] importa[illegible]
[illegible] pase como parámetro a una funcion [illegible]
 * por REFERE[illegible] por [illegible]
 * modificaci[illegible] aplicara [illegible]
 */
void func3(int arreglo[])
{
        int i;

        for(i=0 ; i<TAM , i++)
        {
                arreglo[i]   3,
        }

        return;
}

vo[illegible] main()
{
        int u   1,
        int v   3,
        int arreglo[]  [illegible] 6,9,12},
        int i;

        printf("[illegible]
        func1(u,v);  [illegible]
        printf("valores [illegible] de [illegible] llamada [illegible] ,u,v);

        printf("valores [illegible] de la llamada a func[illegible]d v=%d\n",u,v),
        func2(&u,&v); /* Se pasan [illegible]s [illegible]nes [illegible]ia de [illegible] enteros */
        printf([illegible]es despues de la llamada a func2[illegible]d\n",u,v);

        printf("contenido del arreglo antes de la llamada: "),
        for(i=0 ; i<TAM , i++)
                printf("%d ', arreglo[i]);
        printf("\n"),

        func3(arreglo);  [illegible] especifique, el arreglo se pasa por referencia */

        printf("contenido del arreglo d[illegible] de la llamada  "),
        for(i=0 ; i<TAM ; i++)
                printf("%d  , arreglo[i]);
        printf("\n");

        system("PAUSE");
}
```

1.3 Apuntadores y arreglos dinámicos

En principio, C sólo permite el manejo estático de arreglos, es decir, el tamaño de los mismos debe estar definido en tiempo de compilación, ya que de lo contrario genera un error. Aún así es posible simular el manejo de arreglos dinámicos en C (como en Java por ejemplo) utilizando apuntadores.

Para ello es posible definir los siguientes procesos:

- Crear un arreglo dinámico (inicializarlo)
- Aumentar el tamaño del arreglo dinámico (producto de una inserción)
- Buscar un elemento dentro del arreglo dinámico
- Reducir el tamaño del arreglo dinámico (producto de una eliminación)
- Imprimir el contenido del arreglo dinámico

Ejemplo 1.3 *Manejo de arreglos dinámicos en C utilizando apuntadores:*

```
[illegible] NO_SELECT      1      // Indica la opción [illegible]
[illegible] FALSE          0      // Definic[illegible]
[illegible] TRUE                  // Definicion[illegible] TRUE [illegible]

[illegible] maxNumeroRegistros;   // [illegible] define [illegible] máxi[illegible]
[illegible] *arregloDinamico;     [illegible] de [illegible]
                                  [illegible]

[illegible] liza [illegible]
[illegible]

*crearArregloDinamico(int *arreglo)
{
    return arreglo=NULL;
}

[illegible] egloFuent[illegible], int arreg[illegible]tino [illegible]
[illegible] liminar[illegible]
[illegible] de un [illegible] fuente a [illegible]

[illegible] donde se [illegible] los [illegible]
[illegible] donde se copiarán los [illegible]
[illegible] que se [illegible] en el proceso.
[illegible] función [illegible] realiza [illegible] proceso
[illegible] debe [illegible]
[illegible] FALS
[illegible] onAEliminar. [illegible] bit eliminando [illegible] en TRUE, éste valor [illegible]
[illegible] eliminar del arreglo fuente. En caso que el
[illegible] contenga [illegible] parámetro [illegible] onAEliminar [illegible] pasarse [illegible]

[illegible] copiarArreglo(int arregloFuente[], int arregloDestino[], int posicionesACopiar,
                    int eliminando, int posicionAEliminar)

    [illegible] posicionDestino, posicionFuente;

    [illegible] función [illegible] llamada desde [illegible] módulo de inserción [illegible]
    [illegible] (!eliminando)
    {
        [illegible] fuente, copiando [illegible] uno a uno
            [illegible]
        for (posicionDestino=0;posicionDestino<posicionesACopiar;posicionDestino++)
            arregloDestino[posicionDestino] [illegible] arregloFuente[posicionDestino];
```

```
        }
        else
        {
                posicionDestino = 0;

                [illegible] os [illegible]
                [illegible] arreglo destino [illegible]
                for (posicionFuente=0 ; posicionFuente<posicionesACopia [illegible] posicionFuente++)

                        * Si la posición [illegible] arreglo fuente [illegible]
                        /* [illegible] desea eliminar se copia dicha posición [illegible] el arreglo des[illegible]
                        [illegible] (posicionFuente [illegible] posicionAEliminar)
                        {
                                arregloDestino[posicionDestino]=arregloFuente[posicionFuente];
                                posicionDestino+-; // Se incrementa el índice del arreglo
                        }



}


/**
 * [illegible] tama[illegible] dinámico [illegible]
 * [illegible]
 * devuelve [illegible] posición [illegible]; [illegible] contrario [illegible]

[illegible] aumentarArregloDinamico()
[illegible]
        int *aux;
        int i;

        i = maxNumeroRegistros;

        [illegible] un arreglo dinám[illegible]
        [illegible]
        aux = (int *)malloc(sizeof(int [illegible] 1)),

        /* [illegible] no [illegible] NO_SELECT [illegible]
        if (!aux)
                return [illegible]O_SELECT;

        [illegible] número [illegible] registros [illegible]
        [illegible] (i [illegible]
        {
                [illegible]
                [illegible]
                [illegible]
                [illegible]
                [illegible]
                copiarArreglo(arregloDinamico, aux, maxNumeroRegistros, FALSE, NO_SELECT);
                free(arregloDinamico), // Se libera la memoria ocupada por el arreglo dinámico
                arregloDinamico = NULL; // Se hace que el arreglo apunte nuevamente [illegible]
        }

        maxNumeroRegistros++; [illegible]
        arregloDinamico = aux; [illegible]

        return i;
}
```

```
[illegible]
[illegible]lemento(int clave  : Busca la clave [illegible] como [illegible]
* dentro [illegible]
*/
[illegible] buscarElemento(int clave)
[illegible]
        int i;

        i = 0;

        /* Mientras [illegible] llegue a [illegible] clave sea dife[illegible] */
        /* al contenido de la posición a [illegible] */
        while ((i<maxNumeroRegistros) && (arregloDinamico[i] != clave))
        {
                i++;
        }

        /* Si aún [illegible] llegue [illegible] arreglo, [illegible] ida [illegible] */
        /* produce por la segunda [illegible] es decir, se encontró [illegible] */
        if (i<maxNumeroRegistros)
                return i;

        return NO_SELECT;
}


[illegible]
* Función eliminarElemento int po[illegible] del arregl[illegible] el elemento [illegible]
* se encuentra en la posición pasada como parámet[illegible]
*/
[illegible] eliminarElemento(int posicion)
{
        int *aux;
        int i;

        /* Si la posicio[illegible]
        if (posicion != NO_SELECT)
        {
                i = maxNumeroRegistros - 1;

                /* Se crea un arreglo d[illegible] posiciones */
                [illegible] */
                aux = (int *)malloc(sizeof(int [illegible](i));

                [illegible] !aux)

                        return NO_SELECT;
                }

                // Si el [illegible] de cero [illegible] e[illegible]
                if(i)

                        [illegible] en el [illegible], tomando en
                        [illegible]
                        [illegible]
                        [illegible]
                        /* posicionAEliminar = p[illegible]
                        copiarArreglo(arregloDinamico, aux, maxNumeroRegistros, TRUE,
                                        posicion);
                        free(arregloDinamico); // [illegible] memoria ocupada por el arreglo
                        arregloDinamico = NULL; // [illegible] apunte a [illegible]
                }

                maxNumeroRegistros--; /* [illegible] ar[illegible]
                arregloDinamico = aux;/* El arre[illegible]

                return i;
        }

        return NO_SELECT;
}
```

```
/*
 * Función imprim[illegible] Contra en
 * recorr[illegible] de [illegible]
 [illegible] e [illegible]

void imprimirArregloDinamico(i[illegible] arreglo[], int numeroPosiciones)
{
        int i;

        /* S[illegible] número de posic[illegible] or que cero
        if (numeroPosiciones > 0)
        {
                printf("Contenido del arreglo  "),

                [illegible] (i=0 , i<numeroPosiciones , i++)
                        printf("%d ", arreglo[i]),

                printf("\n"),
        }
        else
                printf("El arreglo esta vacio [illegible]
                        el contenido\n"),

        system( [illegible]
}
```

REPASO

- ¿Qué es un apuntador?
- ¿Cuáles son los tipos de operadores monarios que pueden ser aplicados sobre un apuntador?
- Describa la diferencia entre pasar un parámetro por valor y pasarlo por referencia
- ¿Cuál es la sintaxis para pasar un parámetro por referencia en lenguaje C?
- Describa la lógica para simular arreglos dinámicos en C

CAPÍTULO 2

Listas

Contenido

2.0 Introducción
2.1 Definición de un nodo
2.2 Funciones para el manejo de una lista

2.0 Introducción

Una lista no es más que un conjunto de estructuras autorreferenciadoras de información, todas ellas enlazadas gracias a un valor de referencia. Cada una de las estructuras que conforman el conjunto deben ser del mismo tipo.

Éstas estructuras no son únicas de lenguaje C. Por el contrario, pueden ser implementadas en cualquier lenguaje de programación. Las listas pueden ser clasificadas en dos grandes ramas:

- **Listas estáticas**: Pueden ser representadas a través de estructuras estáticas, es decir, con un tamaño definido en tiempo de compilación. Un ejemplo clásico de este tipo de listas son los arreglos, tal como puede apreciarse en la figura 3.

Figura 3. Representación de las listas estáticas utilizando arreglos
Fuente: elaboración propia

En este caso, es posible definir un arreglo de tamaño N, en el cual cada posición contiene un registro con un nombre, apellido, C.I., fecha de nacimiento, dirección, teléfono y email.

AL PROGRAMAR

Recordar que en C / C++ un arreglo de tamaño N tendrá N casillas, representadas desde la posición cero (0) hasta la posición N-1. Es un error común al comenzar a programar en C el realizar recorridos sobre un arreglo desde 1 hasta N, siendo N el tamaño de dicho arreglo. Esto producirá errores en tiempo de ejecución, ya que la posición N del arreglo no está definida.

- **Listas dinámicas**: Están basadas en estructuras de datos enlazadas. La idea básica de este tipo de estructuras es que cada componente dentro de la estructura posea un apuntador que contendrá la dirección de memoria donde estará almacenado el siguiente componente. Esto permite que cada uno de los elementos que constituyan la lista dinámica puedan ser fácilmente modificados, simplemente alterando los apuntadores de cada uno de ellos, como puede apreciarse en la figura 4.

Figura 4. Representación de las listas dinámicas utilizando una lista simplemente enlazada
Fuente: elaboración propia

Como resultado de este manejo, el tamaño de la lista dinámica no está definido. El mismo puede aumentar o disminuir en tiempo de ejecución.

Existen diferentes variantes de la lista simplemente enlazada, como lo son:

Listas doblemente enlazadas: En este caso cada nodo posee dos apuntadores, uno que indica la posición en memoria del siguiente nodo del conjunto de componentes y otro que indica la posición en memoria del nodo anterior al actual.

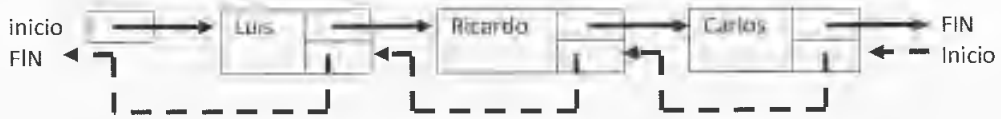


Figura 5. Representación de las listas dinámicas utilizando una lista doblemente enlazada
Fuente: elaboración propia

Listas circulares: Es un caso particular de las listas simplemente enlazadas. En las listas circulares, siempre el último nodo del conjunto de componentes no apunta a un FIN, sino por el contrario apunta de nuevo al inicio de la lista (de ahí su nombre de listas circulares).

Figura 6. Representación de las listas dinámicas utilizando una lista circular
Fuente: elaboración propia

Listas estáticas	Listas dinámicas
• Tamaño definido en tiempo de compilación	• Tamaño definido en tiempo de ejecución
• Es recomendable utilizar estructura de estructura o en caso de [illegible] el tamaño del conjunto de datos con los cuales se trabajará	• Si no se conoce el tamaño del conjunto de elementos con [illegible] no se trabaja [illegible] el mismo [illegible] constantemente en el [illegible] se recomienda utilizar listas dinámicas

Figura 7. Tabla comparativa entre las lista estáticas y las listas dinámicas
Fuente: elaboración propia

Independientemente si se manejan estructuras estáticas o dinámicas, cada registro o componente de la lista se denomina **nodo**.

Para fines prácticos, todas las implementaciones en C que se verán sobre listas serán hechas para listas dinámicas simplemente enlazadas, debido a que si se tiene claro este concepto, implementar una lista estática, una lista doblemente enlazada o una lista circular no debería acarrear mayores complicaciones.

2.1 Definición de un nodo

En términos generales, un nodo de una estructura dinámica puede definirse de la siguiente manera:

```
struct    nombreEstructura
{
          tipoDeDato NombreDelCampo1;
          tipoDeDato NombreDelCampo2;
          tipoDeDato NombreDelCampo3;
          tipoDeDato NombreDelCampo4;

          tipoDeDato NombreDelCampoN;

          struct nombreEstructura *sig;
},
```

Donde nombreEstructura corresponde al nombre que se le asignará a la estructura a definir, tipoDeDato NombreDelCampo1, tipoDeDato NombreDelCampo2, ..., tipoDeDato NombreDelCampoN hacen referencia a todos y cada uno de los campos que contendrá la estructura y struct nombreEstructura *sig define el apuntador que contendrá la dirección de memoria del siguiente nodo del conjunto de componentes.

Por definición, el último nodo del conjunto apuntará a un FIN o final de lista. En C este final de lista se define con la palabra reservada NULL.

Figura 8. Representación de una lista simplemente enlazada en C. Nótese que el último nodo de lista apunta al valor NULL, que indica el final de la misma
Fuente: elaboración propia

Una vez definida la estructura, es posible establecer un tipo de dato que haga referencia a dicha estructura. Para ello se utiliza la palabra reservada typedef, de la siguiente manera:

typedef struct nombreEstructura nombreDelTipoDeDato;

La instrucción permite al programador definir sus propios tipos de datos. En forma general, el puede ser usado como sigue:

typedef tipoDeDato nombreDelTipoDeDato;

En el caso que se está tratando, se está definiendo un nuevo tipo de dato (nombreDelTipoDeDato) que viene a ser del tipo struct nombreEstructura. En el ejemplo 2.1 puede apreciarse como realizar esta definición en conjunto con la creación de la estructura.

Ejemplo 2.1 *Definición de un nodo para una estructura dinámica en lenguaje C:*

```
/* [illegible] 'lista_elementos' qu[illegible]
   [illegible] lista simplemente enlazada
*
struct lista_elementos [illegible]
{
        int valor;                                    [illegible] nodo [illegible]
        struct lista_elementos *sig;   [illegible]
}

/* [illegible] est[illegible]
   [illegible] declarado [illegible]
*/
typedef struct lista_elementos _nodo;
```

Para el ejemplo 2.1, cada vez que se haga referencia al tipo de dato `_nodo`, se estará hablando de un estructura del tipo `lista_elementos`, la cual contiene un entero valor y un apuntador a otra estructura que es del mismo tipo.

2.2 Funciones básicas para el manejo de una lista

Existen funciones básicas para el manejo de la lista, independientemente de si se trata de una lista estática o dinámica:

- Crear una lista
- Verificar si una lista está vacía
- Mostrar el contenido de una lista
- Destruir una lista (eliminar todos y cada uno de sus elementos)
- Insertar elementos en una lista
- Buscar elementos en una lista
- Eliminar elementos de una lista

Ejemplo 2.2 *Funciones básicas para el manejo de una lista.*

```
efine FALSE          0        ;  Definicion del [illegible] maneja booleanos)
[illegible] TRUE           1        ;  Definicion [illegible] TRUE [illegible] no maneja [illegible]

[illegible]

nodo *crearLista(_nodo *registro)
{
        return registro=NULL;
}

[illegible]

[illegible] esListaVacia(_nodo *registro)
{
        [illegible] (registro==NULL)
                return TRUE;
        else
                [illegible]rn FALSE;
}

[illegible] *registro) [illegible]
[illegible] los nodos

[illegible]rarLista(_nodo *registro)
{
        _nodo *aux;

        system("cls");

        printf("lista [illegible]");
        aux [illegible] registro;

        while (aux [illegible] NULL)
        {
                printf("%d [illegible]",aux->valor),
                aux [illegible] aux->sig;
        }

        printf("NULL\n');
        system("PAUSE");

        return;
}[illegible] mostrarL[illegible]
```

```
 * li
 */
_nodo *destruirLista(_nodo *registro)
{
        nodo *p,*q;
        p = registro;

        while (p != NULL)
        {
                q = p;
                p = p->sig;
                free(q);
        }

        registro = NULL;

        return registro;
}
```

La función crearLista es la función básica para el manejo de listas dinámicas. Debe invocarse dicha función antes de realizar cualquier operación. crearLista recibe como parámetro la lista a crear (_nodo *registro) que se traducirá en el apuntador inicial de la lista (inicio). La función simplemente inicializa el apuntador al inicio de la lista con el valor NULL y retorna dicho valor como un apuntador a la misma estructura nodo.

esListaVacia simplemente valida si la lista dinámica pasada por parámetro está vacía o no (si tiene elementos o no). Nótese que al igual que crearLista, la referencia a la lista dinámica viene dada por el apuntador inicial (llamado registro en el parámetro). Para realizar la validación se hace uso de dos macros o constantes definidas en C:

```
#define FALSE       0       /* Defin
#define TRUE        1       /* Definic
```

Esto se debe a que C no maneja el tipo de dato booleano. Por ende, es simulado con ceros y unos (cero como FALSE, uno como TRUE). Si el apuntador inicial de la lista contiene el valor NULL, quiere decir que no existen elementos entre el inicio y el fin de la lista, por lo que se procede a retornar TRUE (lista vacía). En caso contrario, existe al menos un elemento en la lista, por lo que se procede a devolver FALSE.

La función mostrarLista imprime el contenido de la misma por pantalla. Para ello se utiliza un ciclo while que permite hacer el recorrido secuencial. Se emplea un apuntador auxiliar (aux) el cual es inicializado con la misma dirección de memoria del apuntador pasado como parámetro (_nodo *registro) que hace referencia al inicio de la lista. Mientras el valor de aux sea diferente a NULL, se imprime el contenido del nodo apuntado por aux y se procede a mover el apuntador, haciendo que el mismo contenga la dirección de memoria del siguiente nodo (`aux = aux->sig`). Es importante aclarar que el apuntador del inicio de la lista JAMÁS debe ser movido de su posición, ya que éste hace referencia al primer elemento de la lista. Hacer que el apuntador inicial contenga la dirección de cualquier otra casilla de memoria se traduciría en la pérdida TOTAL de la lista dinámica.

destruirLista es una función opcional, pero se recomienda su implementación, ya que la función permite eliminar todos y cada uno de los nodos que componen la lista referenciada por parámetro. Ésta función debe ser invocada luego de terminar de procesar la lista dinámica, ya que los nodos que la componen seguirán definidos, ocupando un espacio en memoria. La destrucción de un nodo se hace gracias a la llamada a la función free (incluida en la librería stdlib.h). Ésta instrucción libera el espacio en memoria que ocupa el apuntador pasado como parámetro (en este caso, todos y cada uno de los nodos de la lista dinámica).

Veamos la lógica a emplear para las funciones de inserción, búsqueda y eliminación de nodos, aplicadas a una lista dinámica simplemente enlazada:

Insertar elementos en una lista: Al insertar elementos en una lista, siempre se estará en alguno de los siguientes casos:

- La lista en la cual se desea realizar la inserción está vacía, es decir, no contiene elementos. En este caso, el procedimiento para insertar un nuevo elemento es el siguiente:

inicio → NULL	Supongamos que la lista sobre la cual se desea realizar la inserción es una lista de enteros y que se desea agregar el elemento 5 a la lista inicialmente vacía.
inicio → NULL aux →	Gracias a la utilización de un apuntador auxiliar (aux), se crea un nodo vacío. La dirección de memoria de dicho nodo será conocida, gracias al apuntador aux.
inicio → NULL aux → 5 → NULL	El nuevo nodo es cargado con la información a insertar. Así mismo, en apuntador a la siguiente estructura del nodo se hace apuntar a NULL (fin de lista).
inicio aux → 5 → NULL	Se hace que el apuntador inicio (inicio de la lista) apunte a la misma dirección de memoria de aux, es decir, al nuevo nodo.
inicio → 5 → NULL	De esta manera se ha insertado el nuevo nodo como primer elemento de la lista inicialmente vacía.

- La lista en la cual se desea realizar la inserción no está vacía, es decir, posee al menos un elemento. En este caso, el proceso para la inserción de un nuevo elemento a la lista es el siguiente:

inicio → 5 → 7 → NULL	Asumamos que la lista sobre la que se desea realizar la inserción del valor entero 99 ya contiene elementos (al menos uno).

finDeLista inicio 5 7 NULL finDeLista inicio 5 7 NULL aux 99 NULL	Gracias a la utilización de un apuntador auxiliar (aux), se crea un nodo vacío. La dirección de memoria de dicho nodo será conocida, gracias al apuntador aux. El nuevo nodo es cargado con la información a insertar. Así mismo, en apuntador a la siguiente estructura del nodo se hace apuntar a NULL (fin de lista). Seguidamente se define un apuntador finDeLista, con el cual se tratará de ubicar el último nodo de la colección de elementos ya insertados. Al inicio, dicho apuntador contiene la dirección de memoria del inicio de la lista (inicio). En este caso es necesario realizar un recorrido secuencial de la lista, preguntando en cada iteración si al apuntador sig del nodo actual contiene el valor NULL (indicando el final de la lista). Mientras dicha condición no se cumpla, finDeLista deberá ir modificando su valor, guardando la dirección de memoria del siguiente nodo de la colección.
finDeLista inicio 5 7 aux 99 NULL	Se hace que el apuntador sig del último nodo de la lista (se conoce gracias a finDeLista) apunte al nuevo nodo creado (es decir, a aux) en lugar de a NULL.
inicio 5 7 99 NULL	De esta manera se ha insertado el nuevo nodo como el último elemento de la lista.

En resumen, puede decirse que al insertar un elemento en una lista dinámica simplemente enlazada:

- Si la lista está inicialmente vacía, el nuevo nodo se inserta de primero.
- Si la lista posee al menos un elemento, debe ubicarse el final de la lista y allí realizar la inserción. Para este caso, es posible encontrar diferentes implementaciones, en las cuales la inserción se hace en el punto intermedio de la lista o al inicio de la misma, pero al final son simples modificaciones de la lógica descrita aquí.

Ejemplo 2.3 *Función básica para insertar elementos en una lista simplemente enlazada (asumiendo la estructura definida en el Ejemplo 2.1):*

```
/*
 * Func[illegible]ón [illegible]
 * puntero [illegible]
 * misma para [illegible]
 */
nodo *finDeLista(_nodo *registro)
{
        _nodo *aux;
        aux   registro;

        while (aux->sig != NULL)
        {
                aux = aux->sig;
        }

        return aux;
}
```

```
/*
 * [illegible]n [illegible]ertarEnLista(int clave [illegible] Permite [illegible]
 [illegible]

 * Cr terios [illegible] inse[illegible]
 *    [illegible] la [illegible]
 *
 * [illegible] menos [illegible]
 *    [illegible] tiene [illegible] de la [illegible]
 *    [illegible]
 */
nodo *insertarEnLista(int clave,_nodo *registro)
{
        _nodo *nuevoregistro;
        _nodo *aux;

        nuevoregistro = (_nodo *)malloc(siz[illegible]f(_nodo));

        /* [illegible] pudo crear [illegible] nuevo nodo [illegible] memoria */
        if (nuevoregistro != NULL)
        {
                nuevoregistro->valor = clave;
                nuevoregistro->sig = NULL;


                [illegible] lista está va[illegible]
                if (esListaVacia(registro) == TRUE)
                {
                        registro = nuevoregistro;
```

```
            e_se
            {
                    c[illegible] un puntero al final de l
                    s  (para saber d[illegible]
                    ",
                    [illegible] finDeLista(registro),

                    aux->sig = nuevoregistro;
                    del e  e /
        }

        return registro;
} [illegible] de insertarEnLista*,
```

La función insertarEnLista recibe como parámetro la clave a insertar (para este caso particular un valor entero) y un apuntador que será el inicio de la lista con la cual se va a operar. Como se describió anteriormente, es necesario definir un apuntador para el nuevo nodo que se creará. Para ello se utiliza el apuntador "nuevoregistro". Adicionalmente es necesario definir un apuntador que permita hacer el recorrido secuencial de la lista en caso de que se deba realizar la inserción del nodo nuevo al final (apuntador "aux"). Independientemente si el nodo se inserta al inicio o al final de la lista, en ambos casos debe crearse el nodo. Esto se hace gracias a la instrucción:

```
nuevoregistro = (_nodo *)malloc(s[illegible]eof( nodo)',
```

La función sizeof (incluida en la librería stdlib.h) devuelve el tamaño exacto necesario para almacenar en memoria el elemento pasado por parámetro. La instrucción malloc (incluida también en la librería stdlib.h) permite almacenar un espacio en memoria del tamaño pasado por parámetro. En resumen, la línea de código establece la reserva en memoria del espacio necesario para almacenar una estructura de tipo _nodo (recordar que esta estructura es un registro con dos campos: un valor entero y un apuntador a la siguiente estructura). La instrucción `( nodo  *)` simplemente hace lo que se conoce en muchos lenguajes de programación como cast o casteo, mediante el cual se obliga la conversión de tipos de datos. En este caso particular se busca que lo que devuelva la función malloc se convierta obligatoriamente a un apuntador al tipo de estructura _nodo (ya que nuevoregistro es de este tipo).

Una vez creado el nodo nuevo, se valida que el apuntador hacia dicho nodo sea diferente de NULL (`if (nuevoregistro != NULL)`). Esta validación debe realizarse siempre, ya que es posible que hubiera existido algún problema al crear el nodo nuevo en memoria (memoria llena, problemas con la asignación de las direcciones de memoria, etc). Si efectivamente el nodo pudo ser creado, se procede a la carga de la información que contendrá: el campo "valor" de la estructura será igual a la clave a insertar (`nuevoregistro->valor = clave;`) mientras que al apuntador a la siguiente estructura del nodo nuevo será igual a NULL (`nuevoregistro->sig = NULL;`).

Seguidamente se validan los dos casos posibles: si la lista está vacía (`if (esListaVacia(registro) == TRUE)`) se procede a insertar el nodo de primero, haciendo que el apuntador al inicio de la lista sea igual al nodo nuevo. Si la lista no está vacía, nos encontramos en el segundo caso, en el cual se debe hacer un recorrido secuencial de la lista hasta ubicar el último nodo de la colección. Esto puede hacerse gracias a la función `finDeLista`, la cual recibe como parámetro la lista sobre a la cual se le hará el recorrido (en este caso `registro`). La función devuelve un apuntador, que justamente contendrá la dirección de memoria del último nodo de la colección. Una vez hecho esto, la dirección "sig" del último nodo deberá apuntar al nodo nuevo creado (`aux->sig = nuevoregistro;`).

Independientemente del la acción realizada, la funcion insertarEnLista devuelve el apuntador del inicio de la lista actualizado.

Buscar elementos en una lista: Por lo general, sobre las listas se aplican algoritmos de búsqueda secuencial, mediante los cuales se examinan uno a uno los nodos del conjunto de datos hasta encontrar el elemento buscado o llegar al final de la lista, en cuyo caso, el elemento no se encuentra en la lista.

Ejemplo 2.4 *Función básica para buscar elementos en una lista simplemente enlazada:*

```
#define NO_SELECT      -1        /* Indica la opción de no selección

/*
 * Función para buscar un nodo registro en la lista que contenga el
 * elemento clave
 *
 * Criterios para buscar:
 * - Si la lista está vacía, no hay que hacer nada, simplemente
 *   no hay elementos
 * - Si existe al menos un elemento en la lista,
 *   recorrer la lista con un ciclo while. Hacer
 *   contador para saber en qué posición se encuentra
 *   la clave coincida con la clave buscada ya ocurre
 *   lo primero: el elemento no se encuentra en la lista o ocurre
 *   lo segundo: encontramos el elemento !!
 */
int buscarEnLista(int clave, nodo *registro)
{
        nodo  *aux;
        int   posicion;

        /* La lista al menos un elemento */
        if (!esListaVacia(registro))
        {
                aux           = registro;
                posicion      = 1;

                while ((aux->sig != NULL) && (aux->valor != clave))

                        aux = aux->sig;
                        posicion++;


                if (aux->valor == clave)
                {
                        return posicion;

        }

        return NO_SELECT;
}/* fin de buscarEnLista*/
```

La función buscarEnLista recibe como parámetro el elemento a buscar y la lista sobre la cual se realizará la búsqueda. Seguidamente se define un apuntador auxiliar ("aux") para poder realizar el recorrido secuencial a través de la lista y un entero posición que servirá como contador para devolver la posición de la lista donde fue encontrado el elemento. Luego debe validarse si la lista no está vacía (`if (!esListaVacia(registro))`) ya que de estarlo no hay razón alguna para realizar una búsqueda sobre ella. Si la lista efectivamente contiene elementos, se procede a asignarle al apuntador auxiliar la dirección de memoria del inicio de la lista y al entero posicion el valor 1, indicando que nos encontramos sobre la posición 1 del conjunto de elementos. Seguidamente se aplica un ciclo while para realizar el recorrido secuencial de la lista simplemente enlazada. Se valida en este ciclo que no estemos ubicados en el último nodo (`(aux->sig != NULL)`) y que el nodo actual no contenga la clave buscada (`(aux->valor != clave)`). Mientras ambas condiciones se cumplan, el apuntador auxiliar tomará la dirección de memoria del siguiente nodo (`aux = aux->sig;`) y se aumentará el contador de posición (`posicion++;`). En algún punto dejará de ejecutarse el ciclo (bien porque nos encontramos en el último nodo de la lista o porque estamos ubicados en el nodo que contiene la clave buscada), por lo que se procede a validar si el nodo actual contiene la clave que se está

buscando. Si esto es así, se retorna el valor de posicion. En caso contrario (o bien si la lista está vacía), la clave no se encuentra en la lista de elementos, por lo que se procede a devolver la constante NO_SELECT (que contiene el valor -1). Resulta sencillo saber si la clave buscada se encuentra o no en la lista una vez realizada la llamada a la función buscarEnLista. Si el valor retornado por la función es diferente a NO_SELECT, quiere decir que efectivamente la clave fue encontrada en la lista. En caso contrario, la lista está vacía o contiene elementos, pero no la clave buscada.

Es importante aclarar que esta búsqueda devuelve la primera aparición de la clave buscada en la lista (no contempla elementos repetidos dentro de la lista). Es posible modificar la función para devolver por ejemplo, el número de veces que se repite la clave buscada en la lista de elementos.

Eliminar elementos de una lista: Al eliminar elementos de una lista, siempre se estará en alguno de los siguientes casos:

- El elemento que desea eliminar no está en la lista: Éste caso es trivial. Puede realizarse una búsqueda en la lista antes de realizar la llamada al método para la eliminación. Si el elemento no está en el conjunto de datos de la lista, no es necesario ejecutar dicho método de eliminación.

- El elemento que se desea eliminar es el primer nodo de la lista:

inicio 5 7 99 NULL	Supongamos que se desea eliminar de la lista de la izquierda el nodo que contiene el elemento **5**.
inicio 5 7 99 NULL aux	Dado que el elemento **5** es el primer nodo de la lista, puede definirse un apuntador auxiliar (aux) que apunte precisamente hacia dicho nodo.
inicio 5 7 99 NULL aux free	Seguidamente, se hace el que apuntador inicial de la lista (inicio) contenga ahora la dirección de memoria del próximo elemento del conjunto. Una vez hecho el cambio de apuntadores, se procede a ejecutar la instrucción free sobre el apuntador aux, de forma de

	poder liberar el espacio de memoria ocupado por el que fuera el primer nodo de la lista.
inicio → 7 → 99 → NULL	De esta manera se ha eliminado el que fuera el primer nodo de la lista.

- El elemento puede encontrarse en la lista, pero no es el primer nodo del conjunto de elementos:

inicio → 5 → 7 → 99 → NULL	Supongamos que se desea eliminar de la lista el nodo que contiene el elemento **99**.
nodoAnterior inicio → 5 → 7 → 99 → NULL	Dado que el nodo que se desea eliminar no es el primero de la lista, se utiliza un apuntador (nodoAnterior) para hacer un recorrido secuencial hasta ubicarse en el nodo anterior al que se desea eliminar.
nodoAnterior inicio → 5 → 7 → 99 → NULL aux	Se define un apuntador auxiliar (aux) que contenga la dirección de memoria del nodo a eliminar (se puede obtener esta dirección gracias al apuntador nodoAnterior).
nodoAnterior inicio → 5 → 7 → 99 → NULL free Aux	Se produce un salto en la lista, haciendo que la siguiente dirección de memoria de nodoAnterior contenga la siguiente dirección de memoria del nodo a eliminar. Una vez hecho esto, se procede a

	ejecutar el comando free sobre el nodo referenciado por el apuntador aux (aux).
inicio → 5 → 7 → NULL	De esta manera se ha eliminado el nodo **99** de la lista.

Ejemplo 2.5 *Función básica para eliminar elementos en una lista simplemente enlazada:*

```
[illegible]
[illegible] *elim[illegible]
[illegible] de [illegible]
[illegible]
* 1.  Primero se [illegible] lista no esté [illegible] En caso [illegible]
*     [illegible]
*
* 2.  Se [illegible] el elemento [illegible] es el [illegible] lista.
*     De [illegible] de [illegible]
*     punteros [illegible]
*
*     [illegible] elemento, [illegible]
*     hasta encon[illegible]
*     porque [illegible] eleme[illegible]
*
nodo *eliminarDeLista(int clave, nodo *regis[illegible]
{
        nodo   *registroAEliminar, *aux;

        [illegible] (!esListaVacia(registro))
        {
                registroAEliminar [illegible] registro;

                /*Es [illegible] primer [illegible] lista
                [illegible]
                if (registroAEliminar->valor [illegible] clave)
                {
                        registro [illegible] registro->sig;
                        free(registroAEliminar),

                [illegible] car el [illegible] en la lista
                [illegible]

                        [illegible]e ((registroAEliminar->sig [illegible] NULL) &&
                            (registroAEliminar->sig->valor [illegible] clave))
                        [illegible]
                                registroAEliminar [illegible] registroAEliminar->sig;
                        [illegible]

                        [illegible]
                            procede [illegible]

                        if (registroAEliminar->sig [illegible] NULL)
                        {
                                aux [illegible] registroAEliminar->sig;
                                registroAEliminar->sig [illegible] registroAEliminar->sig->sig;
                                free(aux);
                        [illegible]

        }
        return registro;
[illegible] */
```

La función eliminarDeLista recibe como parámetros la clave que se desea borrar de la lista y el apuntador que hace referencia al inicio de la misma (recordar que es posible estar trabajando con varias listas, por lo que es recomendable indicarle a la función sobre cual de todas ella realizará la operación de eliminación).

Lo primero que se valida dentro de eliminarDeLista es que la lista pasada como parámetro no sea vacía (`if (!esListaVacia(registro))`) ya que de lo contrario no tiene sentido hacer ningún tipo de chequeo adicional. Si la lista no está vacía, un apuntador auxiliar procederá a contener la dirección de memoria del inicio de la lista (`registroAEliminar = registro;`) y se procede determinar en cual de los dos casos de la eliminación se está. Seguidamente se valida si el campo valor del nodo apuntado por registroAEliminar es igual a la clave que se desea borrar de la lista (primer caso de la eliminación). En este caso se produce un salto en la lista, haciendo que el inicio sea igual al siguiente nodo de la colección (`registro = registro->sig;`) y se libera la dirección de memoria referenciada por registroAEliminar (`free(registroAEliminar);`). Si no se cumple el primer caso de la eliminación, se debe realizar una búsqueda secuencial dentro de la lista para determinar si el elemento que se desea eliminar está dentro del conjunto de nodos (segundo caso de la eliminación). Para ello se ejecuta un ciclo while con dos condiciones para seguir iterando:

- Que el apuntador registroAEliminar no sea el último nodo de la lista (`registroAEliminar->sig != NULL`)
- Que el siguiente apuntador al que hace referencia registroAEliminar no contenga en su campo valor la clave que se desea borrar (`registroAEliminar->sig->valor == clave`)

Mientras ambas condiciones se cumplan, el apuntador registroAEliminar tomará la dirección de memoria del siguiente nodo (`registroAEliminar = registroAEliminar->sig;`). En algún punto el ciclo dejará de ejecutarse, bien sea porque se llegó al último nodo de la lista (`registroAEliminar->sig == NULL`) o bien porque registroAEliminar contiene la dirección de memoria del nodo con la clave que se desea eliminar (`registroAEliminar->sig->valor == clave`).

Una vez finalizado el ciclo, es necesario validar por qué se dejó de iterar. Para ello se verifica si no se está en el último nodo de la lista (`if (registroAEliminar->sig != NULL)`). En este caso quiere decir que el nodo siguiente a registroAEliminar contiene la clave que se desea borrar, por lo que utiliza un apuntador auxiliar (aux) para hacer referencia a dicho nodo y se procede a hacer el salto en la lista (`registroAEliminar->sig = registroAEliminar->sig->sig;`). Una vez hecho esto, se elimina el nodo referenciado por aux, gracias a la instrucción free (`free(aux);`). Al finalizar la función eliminarDeLista, se devuelve la lista actualizada (`return registro;`).

Es importante aclarar que la implementación de la función de eliminación sólo detecta la primera aparición de clave que se desea eliminar dentro de la lista (no se contemplan elementos repetidos). Es posible realizar modificaciones a ésta función para eliminar todas las apariciones de una clave dada dentro del conjunto de nodos de la lista.

REPASO

- ¿Qué es una lista?
- Describa los diferentes tipos de listas existentes
- Modifique las funciones insertar, buscar y eliminar vistas en el capítulo de forma de manejar una lista circular
- Realice una tabla comparativa entre las listas con arreglos y las listas con apuntadores
- Modifique las funciones insertar, buscar y eliminar vistas en el capítulo de forma de manejar una lista doblemente enlazada
- Modifique la función insertar vista en el capítulo para transformarla en una función de inserción recursiva

CAPÍTULO 3

Pilas

Contenido

3.0 Introducción
3.1 Definición de un nodo
3.2 Funciones para el manejo de una pila

3.0 Introducción

Una pila es un tipo especial de lista con características particulares de inserción y eliminación.

Las pilas se rigen bajo la política L.I.F.O (Last In, First Out) o lo que es lo mismo: "El último en entrar es el primero en salir". Esto se traduce en que el último elemento insertado en la pila será el primero en ser procesado o eliminado.

Al ser un tipo especial de lista, todas las características de una lista vistas en el capítulo 2 aplican para el manejo de una estructura tipo pila.

3.1 Definición de un nodo

En términos generales, un nodo de una estructura dinámica tipo pila puede definirse de la siguiente manera:

```
struct   nombreEstructura
{
         tipoDeDato NombreDelCampo1;
         tipoDeDato NombreDelCampo2;
         tipoDeDato NombreDelCampo3;
         tipoDeDato NombreDelCampo4;
         ...
         tipoDeDato NombreDelCampoN;

         struct nombreEstructura *sig;
}
```

Donde nombreEstructura corresponde al nombre que se le asignará a la estructura a definir, tipoDeDato NombreDelCampo1, tipoDeDato NombreDelCampo2, ..., tipoDeDato NombreDelCampoN hacen referencia a todos y cada uno de los campos que contendrá la estructura y struct nombreEstructura *sig define el apuntador que contendrá la dirección de memoria del siguiente nodo del conjunto de componentes.

Por definición, el último nodo del conjunto apuntará a un FIN o final de pila. En C este final de pila se define con la palabra reservada NULL.

Figura 9. Representación de una pila dinámica en C. Nótese que el último nodo de la pila apunta al valor NULL, que indica el final de la misma
Fuente: elaboración propia

Una vez definida la estructura, es posible establecer un tipo de dato que haga referencia a dicha estructura. Para ello se utiliza la palabra reservada typedef, de la siguiente manera:

typedef struct nombreEstructura nombreDelTipoDeDato;

La instrucción permite al programador definir sus propios tipos de datos. En forma general, el puede ser usado como sigue:

typedef tipoDeDato nombreDelTipoDeDato;

En el caso que se está tratando, se está definiendo un nuevo tipo de dato (nombreDelTipoDeDato) que viene a ser del tipo struct nombreEstructura. En el ejemplo 3.1 puede apreciarse como realizar esta definición en conjunto con la creación de la estructura.

Ejemplo 3.1 *Definición de un nodo para una estructura dinámica tipo pila en lenguaje C:*

```
/* Aq[illegible] se declara la [illegible] elementos  que [illegible]
   La[illegible] para definir [illegible]
*/
struct lista_elementos [illegible] lem[illegible]
{
        int valor;
        struct lista_elementos *sig;  [illegible] odo */
};

[illegible] tipo
[illegible] iba

typedef struc[illegible] lista_elementos _nodo;
```

Para el ejemplo 3.1, cada vez que se haga referencia al tipo de dato `_nodo`, se estará hablando de un estructura del tipo `lista_elementos`, la cual contiene un entero valor y un apuntador a otra estructura que es del mismo tipo.

3.2 Funciones básicas para el manejo de una pila

Al igual que se trató el tema para listas, existen una gama de funciones que pueden ser implementadas sobre una pila, independientemente si estamos en presencia de una pila estática o dinámica:

- Crear una pila
- Verificar si una pila está vacía
- Devolver el tope de la pila (el elemento al inicio de la misma)
- Mostrar el contenido de una pila
- Destruir una pila (eliminar todos y cada uno de sus elementos)
- Insertar elementos en una pila
- Buscar elementos en una pila
- Eliminar elementos de una pila

Ejemplo 3.2 *Funciones básicas para el manejo de una pila:*

```
#def e FALSE        0       /* De[illegible]
#define TRUE        1       /* Defin[illegible]

/**
 * Func[illegible] te
 * pila [illegible] princ[illegible]
 */
_nodo *crearPila(_nodo *registro)
{
        return registro=NULL;
}


 * Función esPilaVac[illegible] nodo [illegible] Permite determinar si la
 * pila [illegible] cia o no
 *
[illegible] esP laVacia(_nodo *registro)
{
            (registro==NULL)
                return TRUE;
        else
                return FALSE;
}


 */
 nodo *topePila(_nodo *registro)
{
        return registro;
}
```

```
 * [illegible] do
 * de [illegible]
 */
void mostrarPila(_nodo *registro)
{
        nodo *aux;

       system("cls");

       printf("Inicio [illegible]),
       printf("Pil [illegible]\n"),
       aux [illegible] registro;

       while (aux [illegible] NULL)
       {
              printf(" [illegible]\n ,aux->[illegible]lor),
              aux [illegible] aux->sig;
       }

       printf(" [illegible] NUL[illegible]
       system("PAUS[illegible]
       return;
}[illegible] de [illegible]rPil[illegible]


 * liberar [illegible]
 *
_nodo *destruirPila(_nodo *registro)
{
        nodo *p,*q;
       p [illegible] registro;

       while (p [illegible] NULL)
       {
              q [illegible] p;
              p = p->sig;
              free(q);
       }

       registro [illegible] NULL;

       return registro;
}
```

Al ser un tipo especial de lista dinámica, la pila hereda algunas funciones básicas del manejo de listas, como lo son crearPila, esPilaVacia, mostrarPila y destruirPila (variantes de crearLista, esListaVacía, mostrarLista y destruirLista respectivamente).

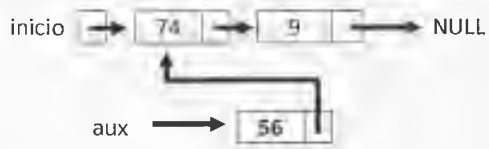
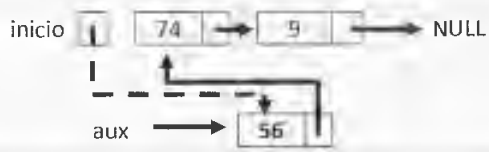
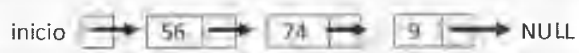
La función crearPila inicializa la pila que es pasada como parámetro (el apuntador referencia al primer nodo de la pila) haciendo que su inicio sea igual a NULL y devolviendo la pila inicializada (`return registro=NULL;`). esPilaVacia simplemente valida si una determinada pila está vacía o no (validando si su inicio pasado como parámetro es igual a NULL o no) y devuelve TRUE (1) o FALSE (0) dependiendo del caso. mostrarPila imprime el contenido de la lista pasada como parámetro, utilizando para ello un ciclo while para recorrer todos y cada uno de sus nodos, desde el inicio o tope hasta el final (NULL). Por último, la función destruirPila hace un borrado de todos y cada uno de los nodos de la pila que se recibe como parámetro, gracias a un ciclo while para aplicar la instrucción free a cada nodo (`free(q);`). Para mayores detalles, es posible consultar el capítulo 2, referente al manejo de listas dinámicas.

Una función particular del manejo de pilas es topePila, la cual devuelve la dirección de memoria del primer nodo de la colección de elementos, ya que en un momento dado pudiera resultar útil saber cuál es el elemento que está ubicado en el tope de la pila. Si la pila está vacía, la función devolverá el valor NULL (el valor de retorno de topePila debe ser validado luego de su llamada).

Veamos la lógica a emplear para las funciones de inserción, búsqueda y eliminación de nodos, aplicadas al tipo de estructura pila:

Insertar elementos en una pila: En el caso de las pilas, ésta función también es conocida como "push". Existe un solo caso posible para la inserción:

inicio → NULL	Supongamos que la pila sobre la cual se desea realizar la inserción es una pila de enteros y que se desea agregar el elemento 9 a la pila inicialmente vacía.
inicio → NULL aux → [ ]	Gracias a la utilización de un apuntador auxiliar (aux), se crea un nodo vacío. La dirección de memoria de dicho nodo será conocida, gracias al apuntador aux.
inicio → NULL aux → [9] → NULL	El nuevo nodo es cargado con la información a insertar. Así mismo, en apuntador a la siguiente estructura del nodo se hace apuntar a NULL (fin de pila).
inicio aux → [9] → NULL	Se hace que el apuntador inicio (inicio o tope de la pila) apunte a la misma dirección de memoria de aux, es decir, al nuevo nodo.
inicio → [9] → NULL	De esta manera se ha insertado el nuevo nodo como primer elemento de la pila inicialmente vacía.
inicio → [74] → [9] → NULL	Ahora bien, imaginemos que la pila sobre la que se desea realizar la inserción del valor entero 56 ya contiene elementos (al menos uno).
inicio → [74] → [9] → NULL aux → [ ]	Gracias a la utilización de un apuntador auxiliar (aux), se crea un nodo vacío. La dirección de memoria de dicho nodo será conocida, gracias al apuntador aux.

	El nuevo nodo es cargado con la información a insertar. Así mismo, en apuntador a la siguiente estructura del nodo se hace apuntar al primer elemento de la pila (inicio o tope de pila).
	Se hace que el apuntador inicio (inicio o tope de la pila) apunte a la misma dirección de memoria de aux, es decir, al nuevo nodo.
	De esta manera se ha insertado el nuevo nodo como primer elemento de la pila.

Como puede apreciarse, el nuevo elemento a insertar en la pila siempre es colocado al inicio de la misma, independientemente de si la pila está vacía o no (a diferencia del caso general del manejo de listas, donde había dos casos posibles).

Ejemplo 3.3 *Función básica para insertar elementos en una estructura tipo pila (asumiendo la estructura definida en el Ejemplo 3.1):*

```
[illegible]
[illegible]e,nodo *[illegible]   Permite insertar
[illegible]

[illegible]
[illegible] el inicio o tope de pila [illegible]
[illegible] independientemente [illegible] si la [illegible]
* [illegible]
[illegible]
nodo *push(int clave, nodo *registro)
{
        _nodo *nuevoregistro;

        nuevoregistro   (_nodo *)malloc([illegible](_nodo)),

        if (nuevoregistro [illegible] NULL)
        {
                nuevoregistro->valor   clave;

                nuevoregistro->sig   registro;
                registro [illegible] nuevoregistro;
        }

        return [illegible]egistro;
[illegible] de push*/
```

A diferencia de la función de inserción en listas, en la pila no existen dos casos de inserción posibles, ya que nodo nuevo siempre es insertado al inicio de la estructura. La función de inserción push recibe como parámetros la clave a insertar y el apuntador al inicio o tope de la pila (asumiendo la estructura definida en el ejemplo 3.1).

Al igual que en caso de la función insertarEnLista, es necesario definir un apuntador auxiliar al nuevo nodo a crear, llamado nuevoregistro (`_nodo *nuevoregistro;`). Lo primero que debe hacerse es reservar el espacio de memoria necesario para albergar una estructura de tipo _nodo (`nuevoregistro [illegible] (_nodo *)malloc(sizeof(_nodo));`). Seguidamente se valida si el nuevo nodo pudo ser creado, es decir, si el

apuntador nuevoregistro contiene una dirección de memoria válida (`if (nuevoregistro != NULL)`). Si efectivamente el nodo pudo ser creado, se carga el campo valor de la estructura con la clave que se desea insertar en la pila (`nuevoregistro->valor = clave;`) y se hace que el apuntador sig del nuevo nodo contenga la dirección de memoria del tope de la pila (`nuevoregistro->sig = registro;`). Una vez hecho esto, se reconstruye la pila haciendo que el tope (registro) apunte ahora al nuevo nodo creado (`registro = nuevoregistro;`). Al final de la función, la pila actualizada es retornada al punto donde se hizo la llamada (`return registro;`)

Buscar elementos en una pila: Al igual que en las listas, generalmente sobre las pilas se aplican algoritmos de búsqueda secuencial, mediante los cuales se examinan uno a uno los nodos del conjunto de datos hasta encontrar el elemento buscado o llegar al final de la pila, en cuyo caso, el elemento no se encuentra en la misma.

Ejemplo 3.4 *Función básica para buscar elementos en una estructura tipo pila:*

```
[illegible] NO_SELECT  -1     /  Indica [illegible] ción

/**
 * Función buscarEnPil[illegible]  Permite b[illegible] un
 * elemento dentro de [illegible]
 *
 * C[illegible]terios para buscar;
 * [illegible] S[illegible] pila [illegible] vacía [illegible] hay que [illegible] encillamente
 *    no [illegible] elementos.
 * [illegible] existe [illegible] menos u[illegible] l[illegible]
 *    recorrer [illegible] gracias [illegible]
 *    [illegible] para salir del [illegible] final de [illegible]
 *    pila o que [illegible] valor [illegible] ocurre
 *    lo primero, el elemento no se encuentra en la pila. Si ocurre
 *    lo segundo, y encontramos [illegible] elemento !!

int buscarEnPila(int clave, nodo *registro)
{
        _nodo   *aux;
        int             posicion;

        aux             = registro;
        posicion        = 1;

        /  Existe [illegible] menos un eleme[illegible] en la pila */
        if (!esPilaVacia(registro))
        {

                whil  ((aux->sig != NULL) && (aux->valor != clave))
                {
                        aux   aux->sig;
                        posicion++;


                if (aux->valor == clave)
                {
                        return posicion;
                }
        }

        return NO_SELECT;
} /[illegible] de buscarEnPila*/
```

La función buscarEnPila recibe como parámetro el elemento a buscar y al lista sobre la cual se realizará la búsqueda. Dentro de la función se realiza una búsqueda secuencial hasta encontrar el elemento o llegar al final de la pila, en cuyo caso la clave buscada no se encuentra en la pila. Para mayores detalles de la función buscarEnPila, es posible consultar la función análoga buscarEnLista del capítulo 2.

Eliminar elementos de una pila: Por regirse por la política L.I.F.O. (Last In, First Out) el único elemento que puede ser eliminado de la pila es aquel que se encuentre en el tope o inicio de pila. Para ello debe seguirse el siguiente procedimiento, conocido como "pop":

inicio → [ ] → [56] → [74] → [9] → NULL	Supongamos que se desea eliminar un elemento de la pila de la izquierda. En este caso no es necesario indicar cuál es el elemento que se desea eliminar, ya que se sabe que es el elemento que está en el tope o inicio de pila.
inicio → [ ] → [56] → [74] → [9] → NULL aux	Se define un apuntador auxiliar (aux) que apunte al elemento del tope o inicio de pila.
inicio [ ] → [56] → [74] → [9] → NULL aux free	Seguidamente, se hace el que apuntador del tope de la pila (inicio) contenga ahora la dirección de memoria del próximo elemento del conjunto. Una vez hecho el cambio de apuntadores, se procede a ejecutar la instrucción free sobre el apuntador aux, de forma de poder liberar el espacio de memoria ocupado por el que fuera el primer nodo de la pila.
inicio → [74] → [9] → NULL	De esta manera se ha eliminado el que fuera el tope de la pila.

Ejemplo 3.5 *Función básica para eliminar elementos en una estructura dinámica tipo pila:*

[illegible]

[illegible] es el que est[illegible]
(por la implementación del push). Por lo tanto [illegible]
recorrer [illegible] pila, basta sólo con eliminar el primer elem[illegible]

```
_nodo *pop(_nodo *registro)

        _nodo   *registroAEliminar;

        if (!esPilaVacia(registro))
        {
                registroAEliminar = registro;

                registro = registro->sig;
                free(registroAEliminar);

        [illegible] registro;
```

En este caso la función de eliminación pop puede ser vista como un caso particular de la eliminación de las listas (ver capítulo 2), en el cual siempre se elimina el primer elemento de la colección de nodos.

En primer lugar, pop recibe como parámetro sólo la pila sobre la cual se hará la eliminación. No es necesario especificar cual es la clave que se desea eliminar, ya que se sabe que el nodo a borrar debe ser el que se encuentra en el tope o inicio de pila. Primero se valida que la pila sobre la cual se operará no esté vacía, ya que en ese caso no hay un elemento a borrar. Si efectivamente la pila no está vacía se procede a realizar el cambio de apuntadores de la siguiente manera:

- Se utiliza un apuntador llamado registroAEliminar para que contenga la dirección de memoria del tope o inicio de pila (`registroAEliminar = registro;`)
- Se hace que el tope o inicio de pila contenga la dirección de memoria del siguiente nodo de la colección (`registro = registro->sig;`)
- Por último, se libera la dirección de memoria ocupada por el elemento que anteriormente estaba en el tope (`free(registroAEliminar);`)

Al final, se devuelve la pila actualizada sin el elemento que se encontraba en el tope (`re[illegible] registro;`).

REPASO

- ¿Cuál es la diferencia entre una lista y una pila?
- Modifique las funciones insertar, buscar y eliminar vistas en el capítulo de forma de manejar una estructura recursiva
- Describa brevemente el proceso de inserción en una pila con apuntadores
- Implementar la estructura tipo pila utilizando arreglos

CAPÍTULO 4

Colas

Contenido

4.0 Introducción
4.1 Definición de un nodo
4.2 Funciones para el manejo de una cola

4.0 Introducción

Al igual que la estructura tipo pila, una cola no es más que un tipo especial de lista con características particulares de inserción y eliminación.

Las colas se basan en la política F.I.F.O (First In, First Out) es decir, "el primero en entrar es el primero en salir". Esto implica que el primer elemento insertado en la cola será el primero en ser procesado o eliminado. Al pensar en una cola, la primera idea que viene a la mente es la de la cola del supermercado o la que hay que hacer para hacer alguna transacción en el banco. La persona que esté al inicio de la cola será la primera en ser atendida. Una vez hecho esto, dicha persona deberá salir de la cola, para darle paso a la siguiente. Si alguna persona nueva llega a la cola, deberá colocarse al final y esperar su turno. Ésta es la lógica básica para el manejo de este tipo de estructura.

Por ser un tipo especial de lista, todas las características de una lista vistas en el capítulo 2 aplican para el manejo de una estructura tipo cola.

4.1 Definición de un nodo

En términos generales, un nodo de una estructura dinámica puede definirse de la siguiente manera:

```
struct    nombreEstructura
{
          tipoDeDato NombreDelCampo1;
          tipoDeDato NombreDelCampo2;
          tipoDeDato NombreDelCampo3;
          tipoDeDato NombreDelCampo4;
           ..
          tipoDeDato NombreDelCampoN;

          struct nombreEstructura *sig;
};
```

Donde nombreEstructura corresponde al nombre que se le asignará a la estructura a definir, tipoDeDato NombreDelCampo1, tipoDeDato NombreDelCampo2, ..., tipoDeDato NombreDelCampoN hacen referencia a todos y cada uno de los campos que contendrá la estructura y struct nombreEstructura *sig define el apuntador que contendrá la dirección de memoria del siguiente nodo del conjunto de componentes.

Por definición, el último nodo del conjunto apuntará a un FIN o final de cola. En C este final de lista se define con la palabra reservada NULL.

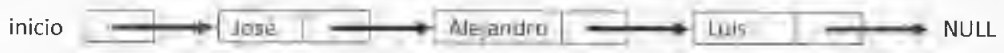


Figura 10. Representación de una estructura dinámica tipo cola en C. Nótese que el último nodo de cola apunta al valor NULL, que indica el final de la misma
Fuente: elaboración propia

Una vez definida la estructura, es posible establecer un tipo de dato que haga referencia a dicha estructura. Para ello se utiliza la palabra reservada typedef, de la siguiente manera:

typedef struct nombreEstructura nombreDelTipoDeDato;

La instrucción permite al programador definir sus propios tipos de datos. En forma general, el puede ser usado como sigue:

typedef tipoDeDato nombreDelTipoDeDato;

En este caso, se está definiendo un nuevo tipo de dato (nombreDelTipoDeDato) que viene a ser del tipo struct nombreEstructura. En el ejemplo 4.1 puede apreciarse como realizar esta definición en conjunto con la creación de la estructura.

Ejemplo 4.1 *Definición de un nodo para una estructura dinámica tipo colo en lenguaje C:*

```
/* Aquí [illegible] declara [illegible] lista_elementos que [illegible]
   [illegible] definir [illegible]

[illegible] lista_elementos [illegible] lista [illegible]
[illegible]
        int valor;                    [illegible] nodo
        struct lista_elementos *sig [illegible]
[illegible]

   [illegible] estrutura [illegible] que [illegible]
   [illegible]

typedef [illegible] lista_elementos nodo;
```

Para el ejemplo 4.1, cada vez que se haga referencia al tipo de dato `nodo`, se estará hablando de un estructura del tipo `lista_elementos`, la cual contiene un entero valor y un apuntador a otra estructura que es del mismo tipo.

4.2 Funciones básicas para el manejo de una cola

Al igual que en las listas y las pilas, existen funciones básicas para el manejo de una estructura tipo cola, independientemente de si se trata de una cola estática o dinámica:

- Crear una cola
- Verificar si una cola está vacía
- Devolver el elemento inicial de la cola
- Mostrar el contenido de una cola
- Destruir una cola (eliminar todos y cada uno de sus elementos)
- Insertar elementos en una cola
- Buscar elementos en una cola
- Eliminar elementos de una cola

Ejemplo 4.2 *Funciones básicas para el manejo de una estructura tipo cola*

```
#define FALSE          0        // Definicion del [illegible]
#define TRUE           1        [illegible]

[illegible]
[illegible]
[illegible] su puntero pr[illegible]

nodo *crearCola(_nodo *registro)
{
        return registro=NULL;
}


/**
 * Función esColaVacia [illegible]
 * cola es vacia o no
 [illegible]
[illegible] esColaVacia(_nodo *registro)
{
        if (registro==NULL)
                return TRUE;
        else
                return FALSE;
}


/**
[illegible] *registro [illegible] un a[illegible]
[illegible] (el [illegible] está en [illegible]
[illegible] está [illegible]

_nodo *inicioDeCola(_nodo *registro)
{
        return registro;
}
```

```
/*
 * Función mostrarCola [illegible] imprimir el [illegible]do
 * de la cola, [illegible] corriendo uno a uno [illegible]
 */
void mostrarCola(_nodo *registro)
{
        nodo *aux;

       system("cls"),

       printf('Inicio [illegible]),
       printf("Cola [illegible]");
       aux = registro;

       while (aux [illegible] NULL) {
              printf("[illegible]\n",aux->valor);
              aux = aux->sig;
       }

       printf("[illegible] NULL [illegible]);
       system("P[illegible]"),
       return;
} /*[illegible]

/*
 * Función *destruirCola [illegible]
 * liberar la memoria de [illegible]
 */
 nodo *destruirCola(_nodo *reg[illegible]ro)
{
        nodo *p,*q;
       p = registro;

       while (p [illegible] NULL)
       {
              q = p;
              p = p->sig;
              free(q),
       }

       registro = NULL;

       return registro;
```

Al igual que en el caso de las pilas, las colas heredan funcionalidades básicas del manejo de listas. Para mayores detalles respecto a las funciones crearCola, esColaVacia, mostrarCola y destruirCola es posible consultar los capítulos 2 y 3 referentes al manejo de listas y pilas respectivamente, y consultar las funciones análogas.

La función inicioDeCola es análoga a topePila (ver el capítulo 3 referente a las funciones básicas para el manejo de pilas). Ésta función devuelve la dirección de memoria del primer nodo de la colección de elementos, ya que en un momento dado pudiera resultar útil saber cuál es el elemento que está ubicado en el inicio de la cola (para saber cuál es el próximo elemento a procesar). Si la cola está vacía, la función devolverá el valor NULL (el valor de retorno de inicioDeCola debe ser validado luego de su llamada).

Veamos la lógica a emplear para las funciones de inserción, búsqueda y eliminación de nodos, aplicadas a una dinámica tipo cola:

Insertar elementos en una cola: Al insertar elementos en una cola, existen dos casos posibles:

- La cola en la cual se desea realizar la inserción está vacía, es decir, no contiene elementos. En este caso, el procedimiento para insertar un nuevo elemento es el siguiente:

inicio → NULL	Supongamos que la lista sobre la cual se desea realizar la inserción es una lista de enteros y que se desea agregar el elemento 26 a la cola inicialmente vacía.
inicio → NULL aux →	Gracias a la utilización de un apuntador auxiliar (aux), se crea un nodo vacío. La dirección de memoria de dicho nodo será conocida, gracias al apuntador aux.
inicio → NULL aux → 26 → NULL	El nuevo nodo es cargado con la información a insertar. Así mismo, en apuntador a la siguiente estructura del nodo se hace apuntar a NULL (fin de cola).
inicio aux → 26 → NULL	Se hace que el apuntador inicio (inicio de la cola) apunte a la misma dirección de memoria de aux, es decir, al nuevo nodo.
inicio → 26 → NULL	De esta manera se ha insertado el nuevo nodo como primer elemento de la cola inicialmente vacía.

- La cola sobre la cual se desea realizar la inserción no está vacía, es decir, posee al menos un elemento. En este caso, el proceso para la inserción de un nuevo elemento a la cola es el siguiente:

inicio → 26 → 20 → NULL	Asumamos que la lista sobre la que se desea realizar la inserción del valor entero 55 ya contiene elementos (al menos uno).
finDeCola inicio → 26 → 20 → NULL finDeCola inicio → 26 → 20 → NULL aux → 99 → NULL	Gracias a la utilización de un apuntador auxiliar (aux), se crea un nodo vacío. La dirección de memoria de dicho nodo será conocida, gracias al apuntador aux. El nuevo nodo es cargado con la información a insertar. Así mismo, en apuntador a la siguiente estructura del nodo se hace apuntar a NULL (fin de cola). Seguidamente se define un apuntador finDeCola, con el cual se tratará de ubicar el último nodo de la colección de elementos ya insertados. Al inicio, dicho apuntador contiene la dirección de memoria del inicio de la cola (inicio). En este caso es necesario realizar un recorrido secuencial de la cola, preguntando en cada iteración si al apuntador sig del nodo actual contiene el valor NULL (indicando el final de la cola). Mientras dicha condición no se cumpla, finDeCola deberá ir modificando su valor, guardando la dirección de memoria del siguiente nodo de la colección.
finDeCola inicio → 26 → 20 aux → 55 → NULL	Se hace que el apuntador sig del último nodo de la cola (se conoce gracias a finDeCola) apunte al nuevo nodo creado (es decir, a aux) en lugar de a NULL.
inicio → 26 → 20 → 55 → NULL	De esta manera se ha insertado el nuevo nodo como el último elemento de la cola.

Si se compara el proceso de inserción en una cola y en una lista (ver capítulo 2) se verá que ambos son idénticos. En resumen, puede decirse que al insertar un elemento en una estructura dinámica tipo cola:

- Si la cola está inicialmente vacía, el nuevo nodo se inserta de primero.
- Si la cola posee al menos un elemento, debe encontrarse el final de la cola y allí realizar la inserción.

Ejemplo 2.3 *Función básica para insertar elementos en una estructura dinamica tipo cola (asumiendo la estructura definida en el Ejemplo 4.1):*

```
/*
 * [illegible]ción finDeCola([illegible] [illegible]elve [illegible]
 * [illegible] [illegible]nal de la c[illegible] [illegible] la
 * [illegible] poder [illegible]erta[illegible]
 */
_nodo *finDeCola(_nodo *regis[illegible]o)
{
        nodo *aux;
        aux = registro;

        while (aux->sig != NULL)
        {
                aux = aux->sig;
        }

        return aux;
} [illegible]DeCola*/

/*
 * [illegible] [illegible]ite insertar
 * [illegible]
 * [illegible]
 * 1- Si la [illegible] la [illegible]ducida
 * [illegible]imero [illegible]
 * 2- Si ex[illegible]
 * [illegible] e la [illegible]ncior [illegible]
 * [illegible] se tiene [illegible] e la [illegible] proc[illegible]
 * insertar [illegible] clave [illegible]
 */
_nodo *insertarEnCola(int clave,_nodo *registro)
{
        nodo *nuevoregistro;
        nodo *aux;

        nuevoregistro = (_nodo *)malloc(sizeo (_nodo));

        if (nuevoregistro != NULL)
        {
                nuevoregistro->valor = clave;
                nuevoregistro->sig = NULL;

                [illegible] [illegible]mero
                [illegible] (esColaVacia(registro))
                [illegible]
                        registro = nuevoregistro;
                }

                [illegible] menos [illegible]
                debe [illegible]
                [illegible]
                [illegible]
```

```
                aux = finDeCola(registro);

                aux->sig = nuevoregistro;
            } /*fin de l else*/
        }

        return registro
} /* fin de insertarE
```

Como puede apreciarse, la función insertarEnCola es idéntica a la función insertarEnLista (ver capítulo 2), por lo que la primera sigue exactamente el mismo procedimiento de inserción:

- La función recibe como parámetros el elemento a insertar así como la cola (inicio de cola) sobre la cual se realizará la inserción.
- Se crea el nodo nuevo (`nuevoregistro = (_nodo *)malloc(sizeof(_nodo));`) utilizando el apuntador auxiliar "nuevoregistro".
- Si efectivamente se pudo reservar el espacio de memoria para el nodo nuevo (`if (nuevoregistro != NULL)`) se carga el campo valor del nodo con la clave a insertar (`nuevoregistro->valor = clave;`) y se hace que el siguiente apuntador de la estructura apunte al fin de cola (`nuevoregistro->sig = NULL;`).
- Se evalúa en cual de los dos casos posibles de inserción se está. Si la cola está vacía (`if (esColaVacia(registro))`), se inserta el elemento al inicio de la misma (`registro = nuevoregistro;`).
- Si la cola no está vacía (contiene al menos un elemento), se procede a ubicar el final de la misma gracias a un apuntador auxiliar llamado "aux" (`aux = finDeCola(registro);`). El apuntador aux contendrá la dirección de memoria del último nodo, por lo que sólo bastará modificar el siguiente apuntador de dicho nodo para que contenga la dirección de memoria del nodo nuevo a insertar (`aux->sig = nuevoregistro;`)

Independientemente del la acción realizada, la funcion insertarEnCola devuelve el apuntador del inicio de la cola actualizado.

Buscar elementos en una cola: Sobre una estructura tipo cola es posible aplicar algoritmos de búsqueda secuencial, mediante los cuales se examinan uno a uno los nodos del conjunto de datos hasta encontrar el elemento buscado o llegar al final de la cola, en cuyo caso, el elemento no se encuentra en la misma.

Ejemplo 4.4 *Funcion básica para buscar elementos en estructura dinámica tipo cola:*

```
#define NO_SELECT   -1

/*
 * Funcion buscarEnCola [illegible]
 * elemento [illegible]
 *
 * Criterios [illegible]
 * 1  Si la [illegible] hay que [illegible] llamente
 *     [illegible] hay [illegible]
 *     [illegible] existe [illegible] menos [illegible] elemento [illegible]
 *     recorrer [illegible]
 *     criterios para salir del while: Que [illegible]
 *     [illegible] que [illegible] valor coincida [illegible]
 *     lo primero, el elemento [illegible] en [illegible] la cola. [illegible]
 *     lo segundo, ya encontramos el [illegible]
 */
int buscarEnCola(int clave, _nodo *registro)
{
        _nodo   *aux;
        int             posicion;

        aux                     = registro;
        posicion                = 1;

        /* Liste [illegible]
        if (!esColaVacia(registro [illegible]
        {

                while ((aux->sig    NULL) && (aux->valor [illegible] clave))
                {
                        aux   aux->sig;
                        posicion++;
                }

                if (aux->valor [illegible] clave)
                {
                        return posicion;
                }
        }

        return NO_SELECT;
}
```

La función buscarEnCola es análoga a la función buscarEnLista (ver capítulo 2). Recibe como parámetros la clave a buscar así como el apuntador inicial a la cola sobre la cual se realizará la búsqueda. Si la cola no está vacía (`if (!esColaVacia(registro))`) se realiza una búsqueda secuencial dentro de la cola hasta encontrar el elemento o llegar al último nodo de la misma. Si la clave es encontrada en el recorrido secuencial (`if (aux->valor [illegible] clave)`) se retorna la posición de la cola donde fue encontrada. En caso contrario se retorna NO_SELECT (-1) indicando que no existe una posición dentro de la cola que contenga el elemento buscado.

Al igual que el caso de listas, ésta función devuelve la primera aparición de la clave buscada en la lista, es decir, no toma en consideración elementos repetidos.

Para más detalles acerca de la lógica de la función de búsqueda, es posible consultar el ejemplo 2.4 del capítulo 2.

Eliminar elementos de una cola: Por regirse por la política F.I.F.O. (First In, First Out) el único elemento que puede ser eliminado de la cola es aquel que se encuentre en el inicio de cola. Para ello debe seguirse el siguiente procedimiento:

inicio → 26 → 20 → 55 → NULL	Supongamos que se desea eliminar un elemento de la cola de la izquierda. En este caso no es necesario indicar cuál es el elemento que se desea eliminar, ya que se sabe que es el elemento que está en el inicio de cola.
inicio → 26 → 20 → 55 → NULL aux	Se define un apuntador auxiliar (aux) que apunte al elemento del inicio de cola.
inicio → 26 → 20 → 55 → NULL aux free	Seguidamente, se hace el que apuntador del inicio de la cola (inicio) contenga ahora la dirección de memoria del próximo elemento del conjunto. Una vez hecho el cambio de apuntadores, se procede a ejecutar la instrucción free sobre el apuntador aux, de forma de poder liberar el espacio de memoria ocupado por el que fuera el primer nodo de la cola.
inicio → 20 → 55 → NULL	De esta manera se ha eliminado el que fuera el inicio de la cola.

Ejemplo 4.5 *Función básica para eliminar elementos de una estructura dinámica tipo cola:*

```
 [illegible] *elimin[illegible]eCo[illegible]a [illegible]
 * [illegible] mi[illegible] elemento de [illegible]
 *
 * Como [illegible] de
 * [illegible] caso [illegible]
 * e[illegible]mento [illegible]
 * [illegible]r la implementa[illegible]EnCola). [illegible] lo [illegible]
 * [illegible]ace [illegible] recorrer l[illegible] [illegible] con eliminar [illegible]l primer
 [illegible]

_nodo *eliminarDeCola(_nodo *registro)
{
        _nodo  *registroAEliminar;

        if (!esColaVacia(registro))
        {
                registroAEliminar = registro;

                registro = registro->sig;
                free(registroAEliminar);
        }
        return registro;
} /*fin de e[illegible]nar[illegible]DeCola*/
```

La implementacón de la función eliminarDeCola es exactamente igual a la implementación de la función pop (ver ejemplo 3.5 del capítulo 3), ya que siempre se elimina el primer elemento de la colección.

Al igual que en el caso de la función pop, eliminarDeCola recibe como parámetro sólo la cola sobre la cual se hará la eliminación. No es necesario especificar cual es la clave que se desea eliminar, ya que se sabe que el nodo a borrar debe ser el que se encuentra en e inicio de la cola (por la plítica F.I.F.O.). Para proceder a la eliminación deben seguirse los siguientes pasos:

Primero se valida que la cola sobre la cual se operará no esté vacía, ya que en ese caso no hay un elemento a borrar. Si efectivamente la cola no está vacía se procede a realizar el cambio de apuntadores de la siguiente manera:

- Se utiliza un apuntador llamado registroAEliminar que contendrá la dirección de memoria del inicio de cola (`registroAEliminar = registro;`)
- Se hace que el inicio de cola contenga la dirección de memoria del siguiente nodo de la colección (`registro = registro->sig;`)
- Por último, se libera la dirección de memoria ocupada por el elemento que anteriormente estaba en el inicio (`free(registroAEliminar);`)

Al final, se devuelve la cola actualizada sin el elemento que se encontraba en el inicio (`return registro;`) el cual ya fue procesado.

REPASO

- ¿Cuál son las diferencias entre una lista, una pila y una cola?
- Modifique las funciones insertar, buscar y eliminar vistas en el capítulo de forma de manejar una estructura recursiva
- Describa brevemente el proceso de inserción en una cola con apuntadores
- Se dice que una palabra o frase es palíndromo si se lee exactamente igual al derecho que al revés. Por ejemplo, las palabras *"ala"* y *"rapar"* son palíndromos. También lo es la frase *"dábale arroz a la zorra el abad"* si no se tienen en cuenta los acentos y espacios en blanco. Implementar un programa en C que dada como entrada una palabra o frase, determine si dicha palabra o frase en es palíndromo. NOTA: utilizar pilas y colas