Segundo momento de la clase …

Unidad 1: Estructuras de Datos Dinámicas

Tema I. Estructuras Dinámicas. Estructuras lineales. Listas enlazadas. Punteros. Declaración de listas enlazadas. Operaciones sobre listas enlazadas. Declaración de Listas circulares. Listas doblemente enlazadas. Pilas y Colas dinámicas.

Por que usar “Punteros”

- Las estructuras de datos estudiadas hasta ahora se almacenan estáticamente en la memoria física del computador.

- Esta rigidez en las estructuras de datos estáticas hace que no pueden crecer o menguar durante la ejecución de un programa.

- La representación de ciertas construcciones (como las listas) usando las estructuras conocidas (arrays) tiene que hacerse situando elementos consecutivos en componentes contiguas, de manera que las operaciones de inserción de un elemento nuevo o desaparición de uno ya existente requieren el desplazamiento de todos los posteriores para cubrir el vacío producido, o para abrir espacio para el nuevo.

Problema

¿Qué sucede si a priori no conocemos la cantidad de espacio de almacenamiento que vamos a precisar?

Solución ⟺ Hacer una previsión??

Ejemplo:

```
Tipos
 TipoPersona TipoPoblacion[1..TAMMAX]
Variables
 TipoPoblacion poblacion
```

MEMORIA DINÁMICA

- La memoria dinámica es un espacio de almacenamiento que se solicita en tiempo de ejecución. De esa manera, a medida que el proceso va necesitando espacio para más líneas, va solicitando más memoria al sistema operativo para guardarlas.

- El medio para manejar la memoria que otorga el sistema operativo, es el **puntero**, puesto que no podemos saber en tiempo de compilación dónde nos dará "huecos" el sistema operativo (en la memoria de nuestro PC).

- Este tipo de datos se crean y se destruyen mientras se ejecuta el programa y por lo tanto la estructura de datos se va dimensionando de forma precisa a los requerimientos del programa, evitándonos así perder datos o desperdiciar memoria (Ej.: definir la cantidad de memoria a utilizar en el momento de compilar el programa).

MEMORIA DINÁMICA

VENTAJAS:

- Es posible disponer de un espacio de memoria arbitrario que dependa de información dinámica (disponible sólo en ejecución): Toda esa memoria que maneja es implementada por el programador cuando fuese necesario.

- Se puede ir incrementando durante la ejecución del programa. Esto permite, por ejemplo, trabajar con arreglos dinámicos.

- Es memoria que se reserva en tiempo de ejecución. Su tamaño puede variar durante la ejecución del programa y puede ser liberado mediante la función free.

MEMORIA DINÁMICA

DESVENTAJAS:

- Es “compleja” su implementación (en el programa o aplicación).

- Es complejo implementar estructuras de datos recursivos (árboles, grafos, etc.).Por ello necesitamos una forma para solicitar y liberar memoria para nuevas variables que puedan ser necesarias durante la ejecución de nuestros programas: Heap.

- La memoria dinámica puede afectar el rendimiento. Puesto que con la memoria estática el tamaño de las variables se conoce en tiempo de compilación, esta información está incluida en el código objeto generado. Cuando se reserva memoria de manera dinámica, se tienen que llevar a cabo varias tareas, como buscar un bloque de memoria libre y almacenar la posición y tamaño de la memoria asignada, de manera que pueda ser liberada más adelante. Todo esto representa una carga adicional, aunque esto depende de la implementación y hay técnicas para reducir su impacto.

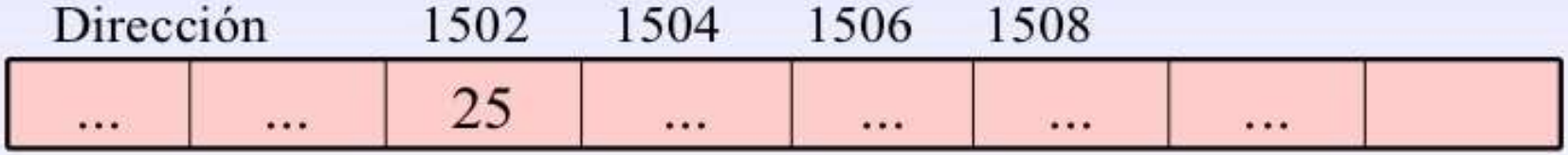
PUNTEROS

¿Qué es un PUNTERO?:

Un puntero es un objeto que apunta a otro objeto. Es decir, una variable cuyo valor es la dirección de memoria de otra variable.

No hay que confundir una dirección de memoria con el contenido de esa dirección de memoria.

```
int x = 25;
```



La dirección de la variable x (&x) es 1502

El contenido de la variable x es 25

Magter. Oscar Adolfo Vallejos
FaCENA - UNNE

PUNTEROS

Las direcciones de memoria dependen de la arquitectura del ordenador y de la gestión que el sistema operativo haga de ella.

En lenguaje ensamblador se debe indicar numéricamente la posición física de memoria en que queremos almacenar un dato. De ahí que este lenguaje dependa tanto de la máquina en la que se aplique.

En C no debemos, ni podemos, indicar numéricamente la dirección de memoria, si no que utilizamos una etiqueta que conocemos como variable. Lo que nos interesa es almacenar un dato, y no la localización exacta de ese dato en memoria.

Magter. Oscar Adolfo Vallejos
FaCENA - UNNE

PUNTEROS

Una variable puntero se declara como todas las variables. Debe ser del mismo tipo que la variable apuntada. Su identificador va precedido de un asterisco (*):

```
int *punt;
```

Es una variable puntero que apunta a variable que contiene un dato de tipo entero llamada punt.

```
char *car:
```

Es un puntero a variable de tipo carácter.

```
long float *num;

float *mat[5];   // . . .
```

FUND. PROG.

Un puntero tiene su **propia dirección** de memoria:

&punt

&car 4

PUNTEROS

```
int *punt = NULL, var = 14;

punt = &var;
printf("%#X,    %#X", punt, &var)  //la misma salida: dirección

printf("\n%d,  %d", *punt, var);   //salida:  14,  14
```

Hay que tener cuidado con las direcciones apuntadas:

```
printf("%d,  %d", *(punt+1), var+1);
```

*(punt + 1) repesenta el valor contenida en la dirección de memoria aumentada en una posición (int=2bytes), que será un valor no deseado. Sin embargo var+1 representa el valor 15.

punt + 1 representa lo mismo que &var + 1 (avance en la dirección de memoria de var).

PUNTEROS

Al trabajar con punteros se emplean dos operadores específicos:

- Operador de dirección: **&** Representa la dirección de memoria de la variable que le sigue:

  &fnum representa la dirección de fnum.

- Operador de contenido o indirección: *

El operador * aplicado al nombre de un puntero indica el valor de la variable apuntada:

```
float altura = 26.92, *apunta;

apunta = &altura; //inicialización del puntero
```


Magter. Oscar Adolfo Vallejos
FaCENA - UNNE

Un puntero siempre está asociado a objetos de un tipo → sólo puede apuntar a objetos (variables o vectores) de ese tipo.

```
int *ip;    /* Sólo puede apuntar a variables enteras */
char *c;    /* Sólo puede apuntar a variables carácter */
double *dp, /* dp sólo puede apuntar a variables reales */
      atof (char *); /* atof es una función que devuelve un real
                        dada una cadena que se le pasa como
                        puntero a carácter*/
```

Es necesario utilizar paréntesis cuando aparecen en la misma expresión que otros operadores unarios como ++ o --, ya que en ese caso se evaluarían de izquierda a derecha.

```
++*ip;
(*ip)++;
```

Dado que los punteros son variables, también pueden usarse como asignaciones entre direcciones. Así:

```
int *ip, *iq;
iq = ip;   /* Indica que iq apunta a donde apunte el puntero ip. */
```

En C, por defecto, todos los parámetros a las funciones se pasan por valor (la función recibe una copia del parámetro, que no se ve modificado por los cambios que la copia sufra dentro de la función).
Ejemplo: Intercambio de dos valores

{VERSION ERRONEA}

```
intercambia (int a, int b) {
int tmp;

tmp = a;
a = b;
b = tmp
}
```

{VERSION CORRECTA}

```
intercambia (int *a, int *b) {
int tmp;

tmp = *a;
*a = *b;
*b = tmp
}
```

Para que un parámetro de una función pueda ser modificado, ha de pasarse por referencia, y en C eso sólo es posible pasando la dirección de la variable en lugar de la propia variable.
Si se pasa la dirección de una variable, la función puede modificar el contenido de esa posición (no así la propia dirección, que es una copia).

Punteros y vectores

En C los punteros y los vectores están fuertemente relacionados, hasta el punto de que el nombre de un vector es en sí mismo un puntero a la primera (0-ésima) posición del vector. Todas las operaciones que utilizan vectores e índices pueden realizarse mediante punteros.

int v[10];

1	3	5	7	9	11	13	15	17	19
v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]

v: designa 10 posiciones consecutivas de memoria donde se pueden almacenar enteros.

int *ip;

Designa un puntero a entero, por lo tanto puede acceder a una posición de memoria. Sin embargo, como se ha dicho antes v también puede considerarse como un puntero a entero, de tal forma que las siguientes expresiones son equivalentes:

ip = &v[0]
x = *ip;
*(v + 1)
v + I

ip = v
x = v[0];
v[1]
&v[i]

Aritmética de punteros

El compilador C es capaz de "adivinar" cuál es el tamaño de una variable de tipo puntero y realiza los incrementos/decrementos de los punteros de la forma adecuada. Así:

```
int v[10], *p;
p = v;

/* p              Apunta a la posición inicial del vector*/
/* p + 0 Apunta a la posición inicial del vector*/
/* p + 1 Apunta a la segunda posición del vector*/
/* p + i  Apunta a la posición i+1 del vector*/

p = &v[9];  /* p apunta ahora a la última posición (décima) del vector */

/* p - 1  Apunta a la novena posición del vector*/
/* p - i  Se refiere a la posición 9 - i en v*/
```

Aritmética de punteros

Ejemplo de recorrido de un vector utilizando índices y punteros.

```
/* prog3.c*/
main(){
int v[10];
int i, *p;

for (i=0; i < 10; i++) v[i] = i;

for (i=0; i < 10; i++) printf ("\n%d", v[i]);

p = v;
for (i=0; i < 10; i++) printf ("\n%d", *p++);
/* Tras cada p++ el puntero señala a la siguiente posición en v */
}
```


Mostrar programa en “C” …

Asignación dinámica de memoria

- `void *calloc(size_t nobj, size_t size)`

calloc obtiene (reserva) espacio en memoria para alojar un vector (una colección) de nobj objetos, cada uno de ellos de tamaño size. Si no hay memoria disponible se devuelve NULL.
El espacio reservado se inicializa a bytes de ceros.
Obsérvese que calloc devuelve un (void *) y que para asignar la memoria que devuelve a un tipo Tipo_t hay que utilizar un operador de ahormado: (Tipo_T *)

Ejemplo:

```
char * c;
c = (char *) calloc (40, sizeof(char));
```

Asignación dinámica de memoria

- `void *malloc(size_t size)`

malloc funciona de forma similar a calloc salvo que: a) no inicializa el espacio y b) es necesario saber el tamaño exacto de las posiciones de memoria solicitadas.

El ejemplo anterior se puede reescribir:

```
char * c;
c = (char *) malloc (40*sizeof(char));
```

Asignación dinámica de memoria

- `void *realloc(void *p, size_t size)`

realloc cambia el tamaño del objeto al que apunta p y lo hace de tamaño size. El contenido de la memoria no cambiará en las posiciones ya ocupadas. Si el nuevo tamaño es mayor que el antiguo, no se inicializan a ningún valor las nuevas posiciones. En el caso en que no hubiese suficiente memoria para "realojar" al nuevo puntero, se devuelve NULL y p no varía.

- `void free(void *p)`

free() libera el espacio de memoria al que apunta p. Si p es NULL no hace nada. Además p tiene que haber sido "alojado" previamente mediante malloc(), calloc() o realloc().

Asignación dinámica de memoria

El puntero que se pasa como argumento ha de ser NULL o bien un puntero devuelto por malloc(), calloc() o realloc().

```
#define N 10
#include <stdio.h>

main(){
char c, *cambiante;
int i;

i=0;
cambiante = NULL;

printf("\nIntroduce una frase. Terminada en [ENTER]\n");
while ((c=getchar()) != '\n') {
  if (i % N == 0){
   printf("\nLlego a %d posiciones y pido hasta %d", i, i+N);
   cambiante=(char *)realloc((char *)cambiante,(i+N)*sizeof(char));
   if (cambiante == NULL) exit(-1);
  }
  /* Ya existe suficiente memoria para el siguiente carácter*/
  cambiante[i++] = c;
}
 /* Antes de poner el terminador nulo hay que asegurarse de que haya
 suficiente memoria */
 if ((i % N == 0) && (i != 0)){
       printf("\nLlego a %d posiciones y pido hasta %d", i, i+N);
       cambiante=realloc((char *) cambiante, (i+N)*sizeof(char));
         if (cambiante == NULL) exit(-1);
   }
 cambiante[i]=0;

 printf ("\nHe leido %s", cambiante);
 }
```

Bibliografía

ESTRUCTURA DE DATOS EN C. Luis Joyanes y Otros. 2007. Editorial MCGRAW-HILL. ISBN: 978-84-5645-9.

INTRODUCCION AL DISEÑO Y ANALISIS DE ALGORITMOS. UN ENFOQUE ESTRATEGICO. R.C.T. Lee; S.S. Tseng; R.C. Chang; Y.T. Tsai 2007. MCGRAW-HILL. ISBN: 978-970-10- 6124-4.

PROGRAMACIÓN EN C, C++, JAVA Y UML. Joyanes Aguilar, Luis. 2a Ed. Ed. Mc Graw-Hill, 2014.

ANÁLISIS Y DISEÑO DE ALGORITMOS: IMPLEMENTACIONES EN C Y PASCAL. López, Gustavo; Jeder, Ismael; Vega, Augusto. AlfaOmega.2009.

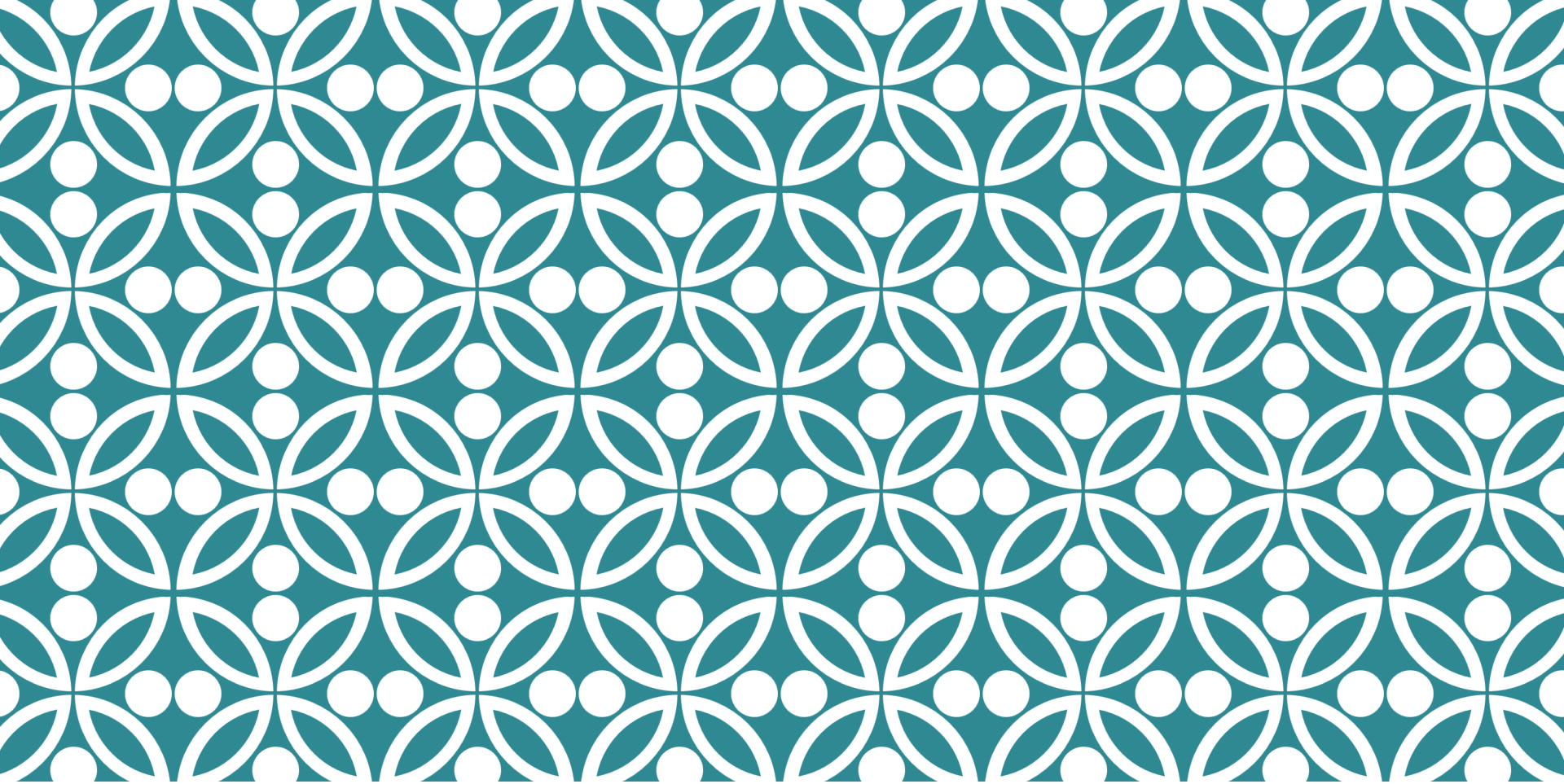
Otro Material didáctico:

Material de estudio y guías de trabajos prácticos para la asignatura Algoritmo y Estructura de Datos II. Cuerpo Docente de la asignatura Algoritmo y Estructura de Datos II. Área Programación. Dpto. Informática. FaCENA. Lugar: Espacio virtual de la asignatura.

Link de videos para tratamiento de listas enlazadas

- https://www.youtube.com/watch?v=2YPMA1p5KoM
- https://www.youtube.com/watch?v=fcpZ_77Ncm8
- https://www.youtube.com/watch?v=7yOnE3yVjMY
- https://www.youtube.com/watch?v=9-18Qp5oQDg
- https://www.youtube.com/watch?v=qI_09fwv3b4
- https://www.youtube.com/watch?v=ljYbVM6j11s
- https://www.youtube.com/watch?v=WxoGvBzWuGs

Magter. Oscar Adolfo Vallejos
FaCENA - UNNE

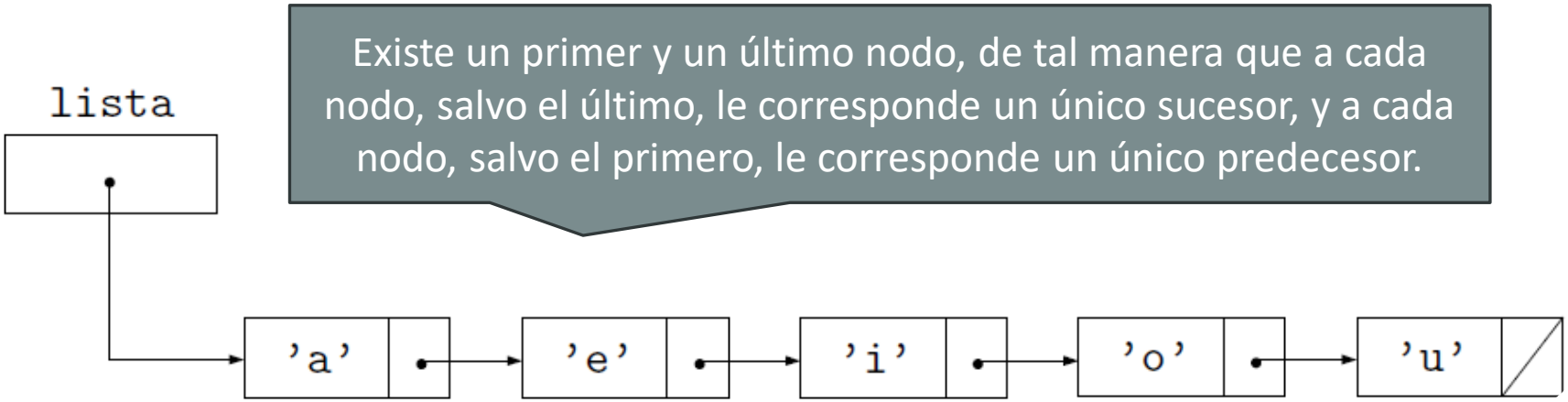


LISTAS ENLAZADAS

Algoritmos y Estructuras de Datos II
Lic. Ana María Company

QUÉ ES UNA LISTA?

- Es una colección de elementos, denominados nodos, dispuestos uno a continuación de otro, cada uno de ellos conectado al siguiente elemento por un enlace o puntero.
- Las listas enlazadas son estructuras muy flexibles y con numerosas aplicaciones en el mundo de la programación.



LISTAS - IMPLEMENTACIÓN DINÁMICA

- Gracias a la asignación dinámica de variables, se pueden implementar listas de modo que la memoria física utilizada se corresponda con el número de elementos de la lista.
- Para ello se recurre a los punteros que hacen un uso más eficiente de la memoria.
- La idea básica consiste en construir una lista cuyos elementos llamados nodos se componen de dos partes:
  1) La información (tElem)
  2) El puntero que apunta al siguiente elemento de la lista (siguiente)

DEFINICIÓN DEL TIPO LISTA

- Una lista será representada como un puntero que señala al principio (cabeza) de la lista.
- Esta forma de definir listas recibe el nombre de lista enlazada dinámica.
- Cada nodo de la lista es una estructura con dos componentes:
  - el contenido del nodo de la lista y
  - un puntero que señala al siguiente elemento de la lista, si existe, o con el valor NULL si es el último.

Listas simplemente enlazadas

Cada nodo (elemento) contiene un único enlace que conecta ese nodo al nodo siguiente o nodo sucesor.

La lista es eficiente en recorridos directos (adelante).

DECLARACIÓN DEL TIPO LISTA

La declaración utiliza el tipo **struct** para agrupar campos de diferentes tipos (***elem*** y ***siguiente***).

Con **typedef** se puede declarar un nuevo identificador de **struct nodo** (***tLista***)

Con el objeto de que el tipo de dato de cada nodo se pueda cambiar con facilidad, se suele utilizar una sentencia **typedef** para declarar el nombre del tipo de dato del elemento (***tElem***).

```
typedef int tElem;

typedef struct nodo {
   tElem elem;
    struct nodo *siguiente;
} tLista;

tLista * lista;
```

OPERADOR DE SELECCIÓN DE UN COMPONENTE ->

Es como una flecha que apunta del puntero ***p*** al objeto que contiene al miembro ***m***

- Si ***p*** es un puntero a una estructura y ***m*** es un miembro de la estructura entonces

p -> m

{ accede al miembro ***m*** de la estructura apuntada por ***p*** }

- El símbolo "->" se considera como un operador simple, aunque conste de dos símbolos independientes "-" y ">“.

- Se denomina **operador de selección de componente** o también **operador de selección de miembro**.

LISTAS ENLAZADAS - OPERACIONES

- Inicializar o crear una lista vacía.
- Comprobar si la lista está vacía.
- Insertar elementos en una lista.
- Eliminar elementos de una lista.
- Insertar un elemento en la posición k de una lista.
- Eliminar el elemento de la posición k de una lista.
- Recorrer una lista enlazada (visitar cada nodo de la lista).

LISTAS ENLAZADAS - OPERACIONES

- Inicializar o crear una lista vacía.
  - Simplemente la lista tiene que apuntar NULL

- Comprobar si la lista está vacía.
  - Una función que retorne true en el caso que la lista apunte a NULL y false en caso contrario

INSERTAR ELEMENTO EN UNA LISTA

- Hay que tener en cuenta si es el primer elemento que se inserta o no
  - Si es el primer nodo entonces:
    1. Se crea el nodo que se va a insertar
    2. Se asigna memoria al nodo
    3. Se asigna el dato recibido al componente correspondiente al elemento
    4. Se indica que el primer nodo apunta a NULL
    5. Se agrega el nodo a la lista: la lista debe apuntar al nuevo nodo

INSERTAR ELEMENTO EN UNA LISTA

- Si la lista ya tiene elementos entonces:
  1. Se crea el nodo que se va a insertar
  2. Se asigna memoria al nodo
  3. Se asigna el dato recibido al componente correspondiente al elemento
  4. Como la inserción es por la parte de adelante de la lista, se indica que al nuevo nodo le sigue el resto de la lista
  5. Como en el nuevo nodo quedó la lista completa, nos queda indicar que la lista que se recibe como parámetro es igual a nuevo nodo

ELIMINAR ELEMENTO EN UNA LISTA

- Se debe eliminar el primer elemento de la lista, para ello:

1) Se guarda en una variable auxiliar el primer nodo de la lista
2) Se avanza el puntero una vez, es decir se pasa al siguiente nodo de la lista
3) Se libera la memoria del nodo a suprimir que contenía el primer elemento de la lista
4) Se asigna NULL a la variable auxiliar que guarda el nodo a suprimir

RECORRER UNA LISTA ENLAZADA

- Consiste en visitar cada nodo de la lista

  1. Se deberá utilizar una variable auxiliar para recorrer la lista

  2. Verificar que la lista no esté vacía: si no está vacía entonces se puede recorrer la lista

INSERTAR UN ELEMENTO EN LA K-ÉSIMA POSICIÓN

- Consiste en insertar un elemento en una posición determinada (k).
  1. Se debe utilizar una lista auxiliar (aux)
  2. Utilizar un bucle para avanzar aux hasta el nodo K-1
  3. Crear el nodo que se va a insertar
  4. Reservar memoria para el nodo a insertar
  5. Se asigna el dato recibido al componente correspondiente al elemento
  6. Se actualizan los punteros:
     a) Se indica a qué nodo tiene que apuntar el nuevo nodo: al siguiente de aux
     b) Se indica a qué nodo tiene que apuntar aux: al nuevo nodo

ELIMINAR EL ELEMENTO DE LA K-ÉSIMA POSICIÓN

- Consiste en eliminar un elemento que se encuentra en una posición determinada (k).
  1. Se debe utilizar una lista auxiliar (aux)
  2. Utilizar un bucle para avanzar aux hasta el nodo K-1
  3. Se resguarda el nodo que se va a suprimir en una variable auxiliar
  4. Se indica a qué nodo tiene que apuntar aux: al siguiente del que se va a eliminar
  5. Se libera la memoria del nodo a suprimir que contenía el elemento de la posición K de la lista
  6. Se asigna NULL a la variable auxiliar que guarda el nodo a suprimir

EJEMPLO

```c
/* Lista enlazada */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

typedef int tElem;

typedef struct nodo {
    tElem elem;
   struct nodo * siguiente;
}tLista;

tLista * lista;
```

```c
void InicializarLista();
bool ListaVacia( tLista * );
void InsertarPrimero( tElem );
void InsertarAdelante( tElem );
void InsertarElemento( tElem );
void EliminarPrimero();
void VisualizarElementos( tLista * );
void InsertarK( int k, tElem );
void EliminarK( int k );
void Menu();
```

```
void inicializarLista() {
    lista = NULL;
}

bool  listaVacia( tLista * pLista ) {
    return ( pLista == NULL );
}
```

```
void insertarElemento( tElem pElem ) {
    if ( lista == NULL )
        insertarPrimero( pElem );
    else
        insertarAdelante( pElem );
}
```

```
void insertarPrimero( tElem pElem ) {

    /* Se crea el nodo que se va a insertar */
    tLista * nuevoNodo;

    /* Se asigna memoria al nodo */
    nuevoNodo = ( tLista * ) malloc( sizeof( tLista ));

   /* Se asigna el dato recibido al componente correspondiente al
      elemento */
    nuevoNodo->elem = pElem;

    /* Se indica que el primer nodo apunta a NULL */
    nuevoNodo->siguiente = NULL;

    /* Se agrega el nodo a la lista: la lista debe apuntar a nuevoNodo */
    lista = nuevoNodo;

    printf("Primer elemento insertado!\n");
}
```

```
void insertarAdelante( tElem pElem ) {
    /* Se crea el nodo que se va a insertar */
    tLista * nuevoNodo;

    /* Se asigna memoria al nodo */
    nuevoNodo = ( tLista * ) malloc( sizeof( tLista ) );

   /* Se asigna el dato recibido al componente correspondiente al
      elemento */
    nuevoNodo->elem = pElem;

    /*  Como la inserción es por la parte de adelante de la lista,
        se indica que al nuevo nodo le sigue el resto de la lista */
    nuevoNodo->siguiente = lista;

    /* Como en nuevoNodo quedó la lista completa, nos queda indicar que
       la lista que se recibe como parámetro es igual a nuevoNodo */
    lista = nuevoNodo;

    printf("Elemento insertado!\n");
}
```

```
void eliminarPrimero() {
    tLista * nodoSuprimir;

    /* Se guarda en una variable auxiliar el primer nodo de la lista */
    nodoSuprimir = lista;

    /* Se avanza el puntero una vez, es decir se pasa al siguiente nodo
       de la lista */
    lista = lista->siguiente;

    /* Se libera la memoria del nodo a suprimir que contenía el primer
       elemento de la lista  */
    free( nodoSuprimir );

    /* Se asigna NULL a la variable auxiliar que guarda el nodo a
       suprimir */
    nodoSuprimir = NULL;

    printf("Primer elemento eliminado!\n");
}
```

```
void visualizarElementos( tLista * pLista ) {
   /* Se deberá utilizar una variable auxiliar para recorrer
      la lista */
   tLista * aux;
   aux = pLista;

   if ( !listaVacia( pLista ) ) {
      /* Se puede recorrer la lista */
      printf( "\n*** Detalle de elementos en la lista ***\n" );
      while(aux != NULL) {
         printf("%d ", aux->elem);
         aux = aux->siguiente;
      }
   } else {
      printf( "\nLa lista esta vacia!!\n" );
   }
   printf("\n\n" );
}
```

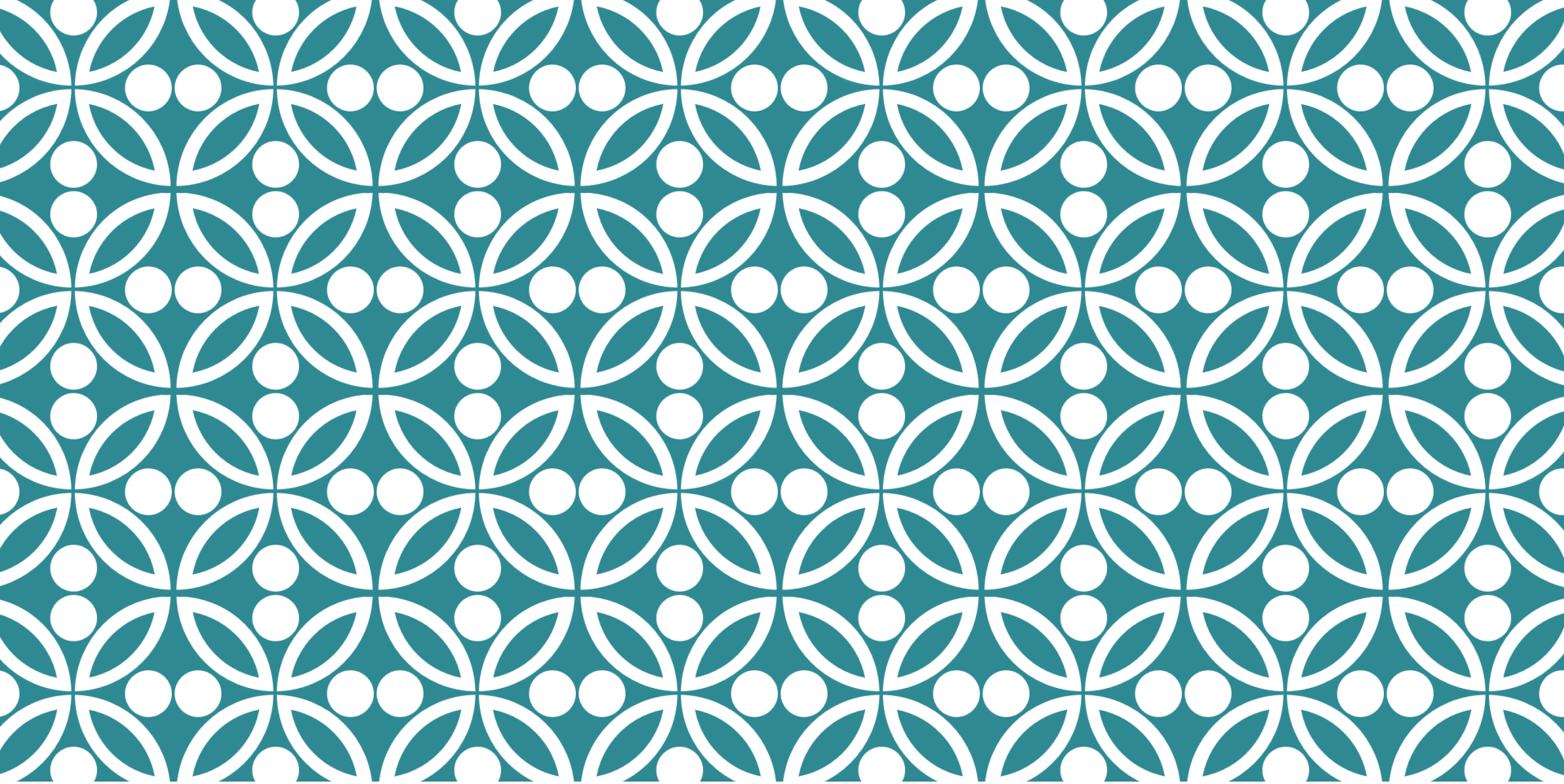
```
void insertarK( int k, tElem nuevoDato ) {
    tLista * nuevoNodo, * aux;
    int i;
    aux = lista;

    /* El bucle avanza aux hasta el nodo K-1 */
    for(i = 1; i < k-1; i++) {
        aux = aux->siguiente;
    }
    /* Se reserva espacio para el nodo a insertar */
    nuevoNodo = malloc(sizeof(tLista));

    /* Se asigna el dato recibido al componente correspondiente
       al elemento */
    nuevoNodo->elem = nuevoDato;

    /* Se actualizan los punteros */
        /* 1. Se indica a qué nodo tiene que apuntar nuevoNodo: al
           siguiente de aux */
        nuevoNodo->siguiente = aux->siguiente;

        /* 2. Se indica a qué nodo tiene que apuntar aux: a nuevoNodo */
        aux->siguiente = nuevoNodo;

    printf("Elemento insertado en la posicion %d!\n", k);
}
```

```
void eliminarK( int k ) {
    tLista * nodoSuprimir, * aux;
    int i;
    aux = lista;

    /* El bucle avanza aux hasta el nodo K-1 */
    for(i = 1; i < k-1; i++) {
        aux = aux->siguiente;
    }
    /* Se resguarda el nodo que se va a suprimir en la
       variable nodoSuprimir */
    nodoSuprimir = aux->siguiente;

    /* Se indica a qué nodo tiene que apuntar aux: al siguiente
      del que se va a eliminar */
    aux->siguiente = nodoSuprimir->siguiente;

    /* Se libera la memoria del nodo a suprimir que contenía
       el elemento de la posición K de la lista  */
    free( nodoSuprimir );

    /* Se asigna NULL a la variable auxiliar que guarda
       el nodo a suprimir */
    nodoSuprimir = NULL;

    printf("Elemento de la posicion %d eliminado\n", k);
}
```

```
int main() {
    inicializarLista();
    printf("Lista vacia? %s\n", listaVacia( lista ) ? "si" : "no");
    insertarElemento( 1 );
    printf("Lista vacia? %s\n", listaVacia( lista ) ? "si" : "no");
    insertarElemento( 2 );
    insertarElemento( 3 );
    visualizarElementos( lista );
    insertarK( 3, 5 );
    visualizarElementos( lista );
    insertarK( 2, 10);
    visualizarElementos( lista );
    eliminarPrimero();
    visualizarElementos( lista );
    eliminarK( 2 );
    visualizarElementos( lista );

    return 0;
}
```

REFERENCIAS

- A Tutorial on Pointers and Arrays in C. Ted Jensen. Disponible en: https://github.com/jflaherty/ptrtut13/blob/master/md/pointers.md
- Mark Allen Weiss. Estructuras de Datos y Algoritmos. Editorial: Addison-Wesley Iberoamericana.
- Joyanes Aguilar, Luis. Programación en Pascal. 4ª Edición. Editorial: McGraw-Hill/Interamericana de España, S.A.U.
- Cristóbal Pareja Flores, Manuel Ojeda Aciego, Ángel Andeyro Quesada, Carlos Rossi Jiménez. Algoritmos y Programación en Pascal.
- Joyanes Aguilar, Luis. Fundamentos de la Programación. Algoritmos, Estructuras de Datos y Objetos. 3ª Edición. Editorial: McGraw-Hill.
- Luis Joyanes Aguilar, Ignacio Zahonero Martinez. Programación en C. Metodología, algoritmos y estructura de datos. Editorial: McGraw-Hill.



PILAS. IMPLEMENTACIÓN CON PUNTEROS.

Algoritmos y Estructuras de Datos II

Lic. Ana María Company

QUÉ ES UNA PILA?

- Una Pila (stack) es una estructura de datos relativamente simple.

Es un conjunto ordenado de elementos de un tipo dado.

Cada elemento tiene un único predecesor y un único sucesor.

QUÉ ES UNA PILA?

- Es un tipo especial de lista lineal, en la cual las inserciones y eliminaciones se hacen por un extremo, llamado la cima, cabeza o tope (top) de la pila.

- Se conoce también como LIFO { last-in, first-out }

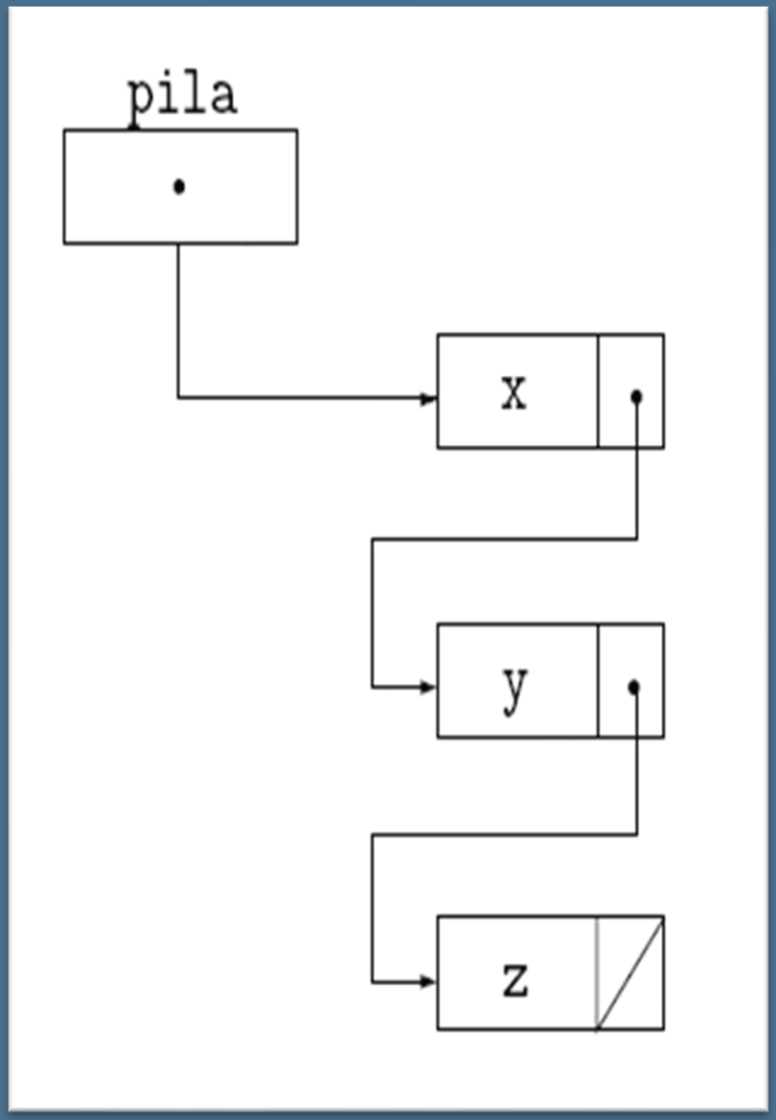
  → último en entrar, primero en salir

- Pertenece al grupo de estructuras de datos lineales

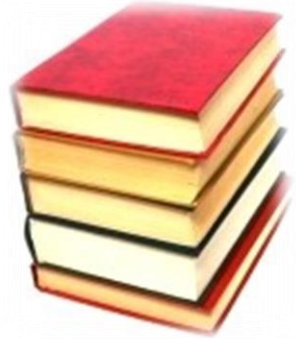
  → Los componentes ocupan lugares sucesivos en la estructura.

Pilas

Sólo se insertan o eliminan elementos por uno de sus extremos



EJEMPLOS COTIDIANOS



IMPLEMENTACIONES

- Puede implementarse de 2 maneras:
  - Con punteros

    → La ventaja es que es la forma dinámica

  - Con vectores - arrays -

    → Con la desventaja de que limita el máximo número de elementos que la pila puede contener y origina la necesidad de una función más

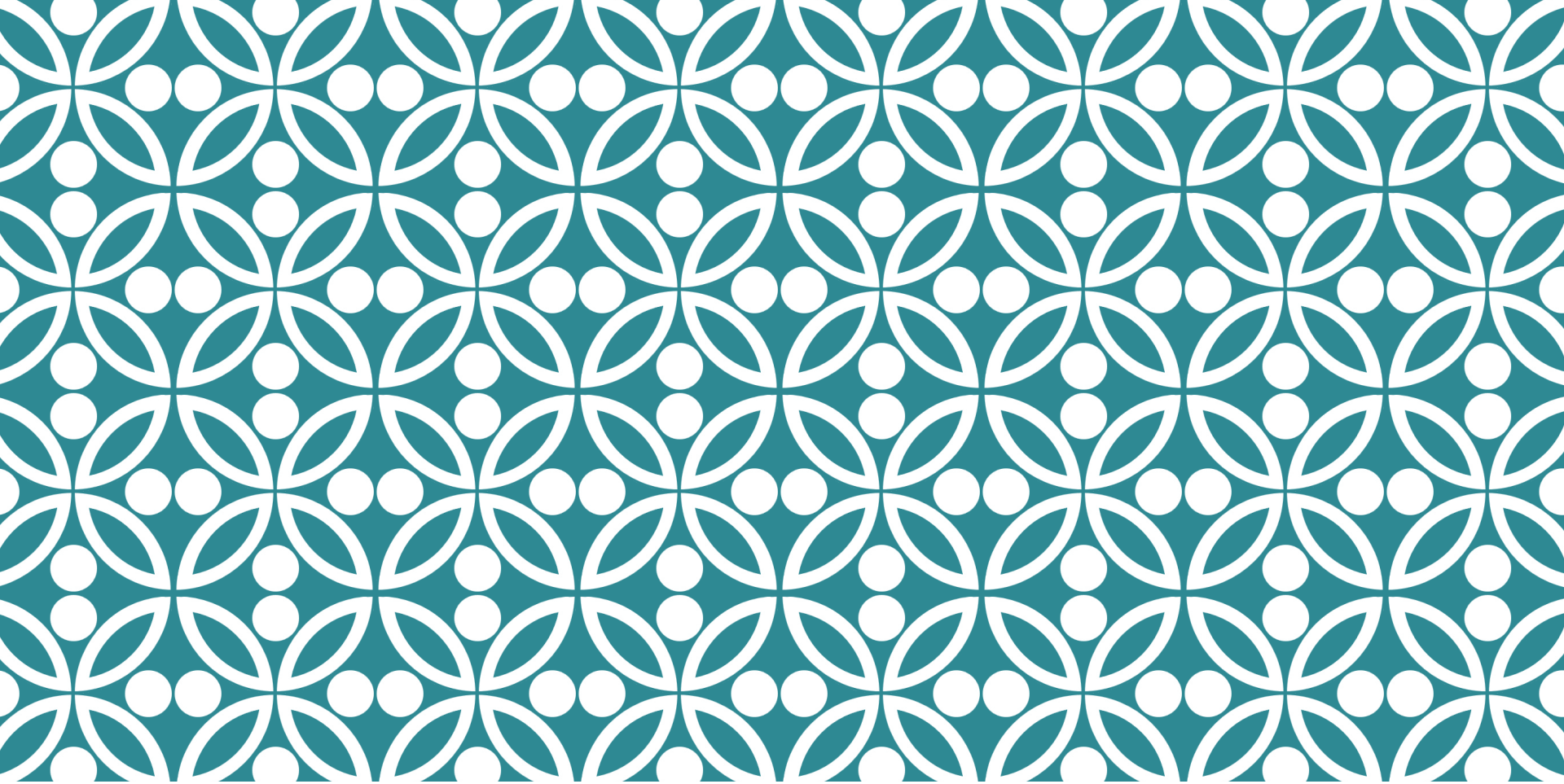
    preguntar si está llena ¿?

OPERACIONES BÁSICAS SOBRE PILAS

- Crear o inicializar
- Pila vacía (empty)
- Apilar (push)
- Desapilar (pop)
- Obtener la cima de la pila
- { Recorrer la pila }

REFERENCIAS

- A Tutorial on Pointers and Arrays in C. Ted Jensen. Disponible en: https://github.com/jflaherty/ptrtut13/blob/master/md/pointers.md
- Mark Allen Weiss. Estructuras de Datos y Algoritmos. Editorial: Addison-Wesley Iberoamericana.
- Joyanes Aguilar, Luis. Programación en Pascal. 4ª Edición. Editorial: McGraw-Hill/Interamericana de España, S.A.U.
- Cristóbal Pareja Flores, Manuel Ojeda Aciego, Ángel Andeyro Quesada, Carlos Rossi Jiménez. Algoritmos y Programación en Pascal.
- Joyanes Aguilar, Luis. Fundamentos de la Programación. Algoritmos, Estructuras de Datos y Objetos. 3ª Edición. Editorial: McGraw-Hill.
- Luis Joyanes Aguilar, Ignacio Zahonero Martinez. Programación en C. Metodología, algoritmos y estructura de datos. Editorial: McGraw-Hill.



COLAS. IMPLEMENTACIÓN CON PUNTEROS.

Algoritmos y Estructuras de Datos II

Lic. Ana María Company

QUÉ ES UNA COLA?

- Es una lista en la que las inserciones se realizan por un extremo **(final)** y las eliminaciones se realizan por el otro extremo **(principio de la lista o frente)**.

- También se llaman listas **FIFO**, del inglés First-In-First-Out, es decir, el primero en entrar es el primero en salir.

Los elementos se insertan por el final y se eliminan por el principio

DECLARACIÓN

```
typedef struct nodo {
        tDatos datos;
        struct nodo * siguiente;
}tNodo;


 typedef struct {
        tNodo * principio;
        tNodo * final;
 }tCola;
```

OPERACIONES BÁSICAS SOBRE COLAS

- Crear o inicializar
- Cola vacía
- Añadir un elemento al final
- Suprimir el primer elemento
- Consulta del primer elemento
- /* Recorrer la cola */

COLAS - OPERACIONES

- Inicializar o crear una cola vacía.
  - Consiste en apuntar NULL los punteros principio y final

- Comprobar si la cola está vacía.
  - Una función que retorne true en el caso que ambos punteros apunten a NULL y false en caso contrario

AÑADIR ELEMENTO EN UNA COLA

- Los nodos se insertan por el final de la cola, para ellos:
  1. Crear un nodo y asignar memoria al nuevo nodo
  2. Asignar valores a la parte de datos del nodo y al campo siguiente (siempre NULL)
  3. Si la cola está vacía estaríamos insertando el primer nodo entonces
     - ambos punteros deben apuntar al nuevo nodo
  4. Caso contrario significa que ya hay elementos en la cola, y
     - solo se actualiza el puntero del final (primero el que corresponde al nodo y luego el puntero final de la cola)

SUPRIMIR ELEMENTO EN UNA COLA

- Se debe eliminar el elemento que se encuentra en el principio de la cola, para ello tener en cuenta lo siguiente:

  1) Si hay elementos en la cola, se puede quitar

  2) Resguardar el nodo del principio, que es el que se va a quitar

  3) Tener en cuenta si la cola es unitaria para inicializarla, debido a que se quita el último elemento. Caso contrario, significa que la cola tiene más elementos, se debe quitar el del principio, solo se debe actualizar el puntero del principio

  4) Liberar la memoria del nodo eliminado

CONSULTA DEL PRIMER ELEMENTO

- Consiste en retornar el nodo del principio

RECORRER UNA COLA

- Consiste en visitar cada nodo de la cola, a partir de posicionarse en el nodo del principio de la cola

    1. Se deberá utilizar una variable auxiliar para recorrer la cola

    2. Verificar que la cola no esté vacía: si no está vacía entonces se puede recorrer.

REFERENCIAS

- A Tutorial on Pointers and Arrays in C. Ted Jensen. Disponible en: https://github.com/jflaherty/ptrtut13/blob/master/md/pointers.md
- Mark Allen Weiss. Estructuras de Datos y Algoritmos. Editorial: Addison-Wesley Iberoamericana.
- Joyanes Aguilar, Luis. Programación en Pascal. 4ª Edición. Editorial: McGraw-Hill/Interamericana de España, S.A.U.
- Cristóbal Pareja Flores, Manuel Ojeda Aciego, Ángel Andeyro Quesada, Carlos Rossi Jiménez. Algoritmos y Programación en Pascal.
- Joyanes Aguilar, Luis. Fundamentos de la Programación. Algoritmos, Estructuras de Datos y Objetos. 3ª Edición. Editorial: McGraw-Hill.
- Luis Joyanes Aguilar, Ignacio Zahonero Martinez. Programación en C. Metodología, algoritmos y estructura de datos. Editorial: McGraw-Hill.