Introducción a la Algorítmica y Programación (3300)

Prof. Ariel Ferreira Szpiniak - aferreira@exa.unrc.edu.ar
Departamento de Computación
Facultad de Cs. Exactas, Fco-Qcas y Naturales
Universidad Nacional de Río Cuarto

Teoría 14

Estructuras de Datos

Estructuras dinámicas de datos

Listas encadenadas

Estructuras de datos

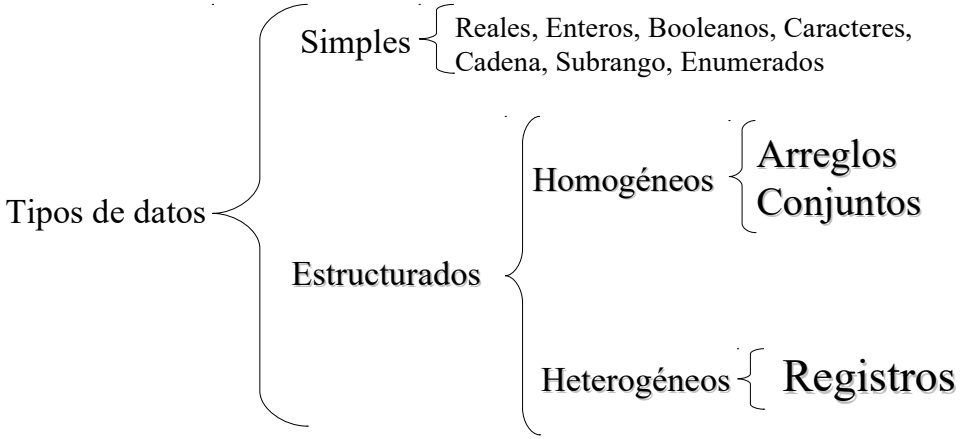
Una estructura de datos *es una colección de datos que pueden ser caracterizados por su organización y las operaciones que se definen en ella.*

Las estructuras de datos son de vital importancia para representar de mejor manera los objetos de la vida real.

Dentro de las estructuras de datos encontramos distintos tipos. Hay varios enfoques o puntos de vista para abordar este tema. A continuación repasaremos las diferentes taxonomías propuestas para clasificar los tipos de datos y las estructuras de datos. Posteriormente nos centraremos en una de ellas para desarrollar la temática de estructuras dinámicas.

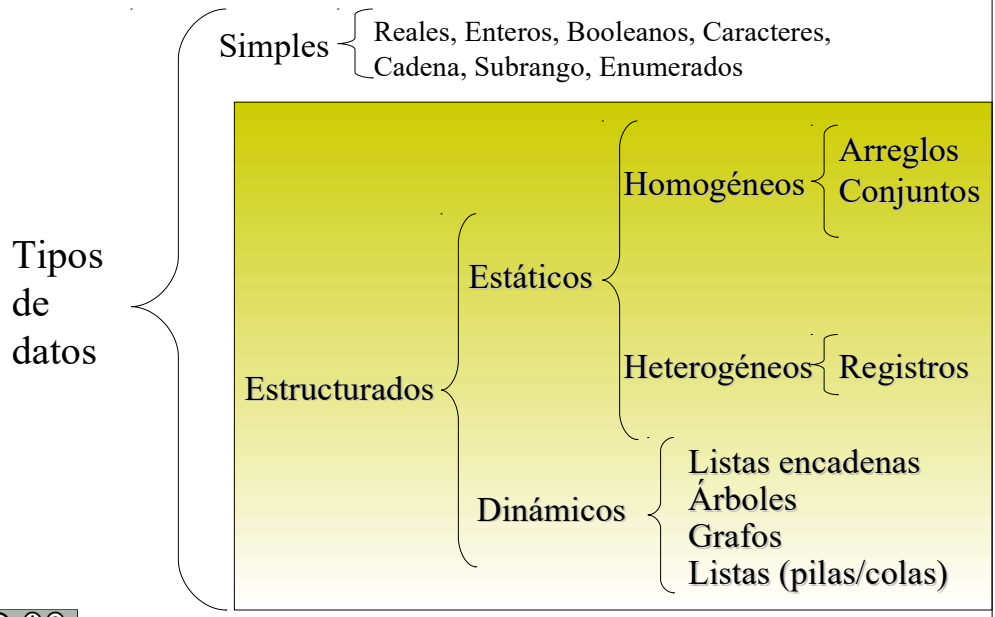
Tipos de datos

Hasta el momento nos hemos centrado en la siguiente taxonomía para tipos de datos:



Tipos de datos

Una clasificación más completa sería:



Estructuras de datos

Las estructuras dinámicas son homogéneas, al igual que los arreglos. En particular, en todo lo referido a estructuras de datos homogéneas (estáticas y dinámicas) hay otras taxonomías:

- Arreglos: uni y bidimensionales
- Listas encadenadas: simples y dobles
- Árboles: binarios, no binarios, V, etc.
- Grafos: dirigidos, no dirigidos.
- Pilas
- Colas

Estructuras de datos

Otra taxonomía:

- Secuencias:
  - Arreglos: uni y bi
  - Listas encadenadas: simples y dobles
  - Pilas
  - Colas
- Árboles: binarios, no binarios, V, etc.
- Grafos: dirigidos, no dirigidos.

Estructuras de datos

La taxonomía que utilizaremos nosotros:

Lineales:

- **Arreglos: uni y bidimensionales**
- **Listas encadenadas: simples y dobles.**
  * También las denominamos *estructuras encadenadas*.

No lineales:

- Árboles: binarios, no binarios, V, etc.
- Grafos: dirigidos, no dirigidos.

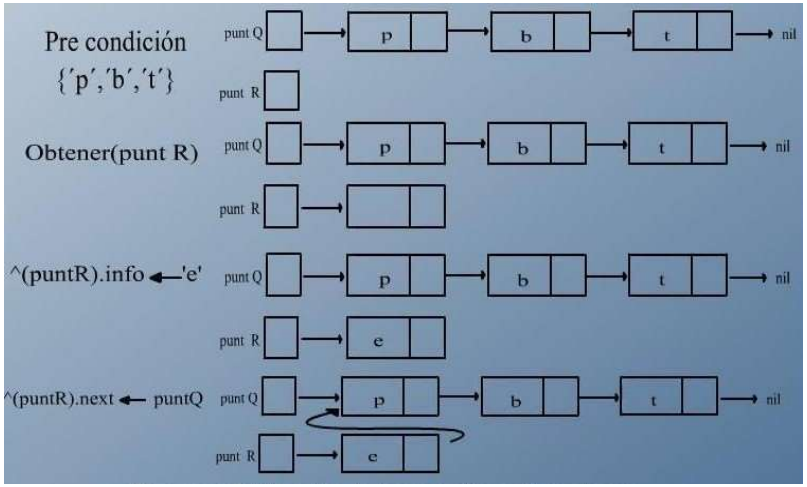
En nuestra materia abordaremos las estructuras ***Lineales*** únicamente. Las estructuras ***No Lineales*** se verán en materias más avanzadas (2do año).

Estructuras de datos

Dentro de las **estructuras de datos lineales** nos encontramos con los arreglos y las listas encadenadas. Los *arreglos* se denominan *estructuras de datos estáticas* mientras que las *listas encadenadas* se denominan *estructuras de datos dinámicas*.

Dentro de las **estructuras de datos no lineales** nos encontramos con los árboles y los grafos. Ambas estructuras son denominadas dinámicas. Sin embargo no las estudiaremos debido a que escapan el alcance de este curso.

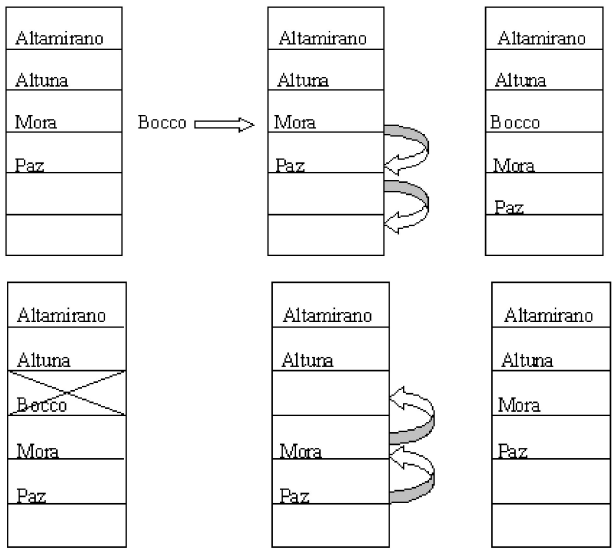
Estructuras de datos dinámicas
Listas encadenadas



Estructuras de datos dinámicas
Listas encadenadas

Las estructuras dinámicas de datos son de gran utilidad para modelar datos del mundo real que cambian constantemente. Por ejemplo, si tenemos datos de personas almacenados en un arreglo, los cuales están ordenados alfabéticamente, y necesitamos insertar los datos de una nueva persona sería necesario insertarlos en el orden que le corresponda, y para ello se deberían mover los elementos (uno por uno) de un lugar a otro para dejar el lugar libre.

Estructuras de datos dinámicas



Esto siempre y cuando haya lugar en el arreglo. Algo similar ocurre si tenemos que eliminar una persona.

Para superar estos problemas nada mejor que el uso de estructuras dinámicas de datos, de allí su importancia.

Estructuras de datos dinámicas
Punteros

Las variables estáticas se guardan en lugares concretos de la memoria y podemos tener acceso a ellas por medio de un identificador (nombre).
Para nuestros programas, la memoria de la computadora es solamente una sucesión casilleros (lugares) de 1 byte cada uno.
Cada casillero tiene una dirección única mediante la cual se puede acceder a su contenido.
La memoria de computadora puede ser comparada con una calle:

- En una calle todas las casas se numeran consecutivamente con un identificador único, tal que si hablamos del número 270 de la calle Nicaragua, podremos encontrar el lugar sin problemas, puesto que debe haber solamente una casa con ese número.
- De acuerdo al tamaño de la casa, ésta puede ocupar uno o más terrenos.
- Lo mismo sucede con las variables y su tipo respectivo. Por ejemplo una variable de tipo carácter ocupará menos espacio que una variable de tipo arreglo de 5 caracteres.

Estructuras de datos dinámicas
Punteros

En la siguiente figura podemos observar de manera esquemática como se almacenaría en la memoria el valor **'a'** en la variable **car** de tipo caracter y los valores **p**, **b** y **t** en la variable **pal** de tipo arreglo de 5 caracteres.

variables

car ε Caracter

pal ε Arreglo [1..5] de Caracter

	Dirección	Memoria
car	0000	'a'
pal[1]	0001	'p'
pal[2]	0010	'b'
pal[3]	0011	't'
pal[4]	0100	reservado
pal[5]	0101	reservado
	0110	libre
	0111	libre
	1000	libre
	1001	libre
	1010	libre
	1011	libre

Estructuras de datos dinámicas
Punteros

Una variable de tipo puntero es una variable que posee la particularidad de almacenar una dirección de memoria. Con una variable de ese tipo es posible acceder a la información que se encuentra alojada en otra parte de la memoria.
De allí la importancia de saber manejar adecuadamente las mismas para evitar cometer errores que puedan afectar a los datos almacenados en la memoria y al funcionamiento de los programas.

Estructuras de datos dinámicas
Punteros

En la siguiente figura podemos observar de manera esquemática como se almacenaría en la memoria una variable **puntQ** de tipo puntero que apunte al valor que se encuentra almacenado en la variable **car**, es decir el valor que se encuentra ubicado en la dirección de memoria **0000**.

variables

car ε Caracter

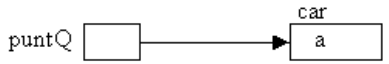
pal ε Arreglo [1..5] de Caracter

puntQ ε puntero a Caracter

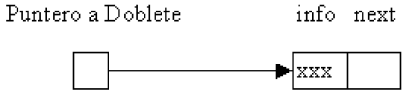
	Dirección	Memoria
car	0000	'a'
pal[1]	0001	'p'
pal[2]	0010	'b'
pal[3]	0011	't'
pal[4]	0100	reservado
pal[5]	0101	reservado
puntQ	0110	0000
	0111	libre
	1000	libre
	1001	libre
	1010	libre
	1011	libre

Estructuras de datos dinámicas
Punteros

A los efectos de hacer más sencilla la interpretación de punteros, se utilizará una representación gráfica en lugar de una tabla. La variable **puntQ** que apunta a la variable **car** se representa de la siguiente manera:



Se dice que la primera variable (puntQ) apunta a la segunda (car).
En el ejemplo anterior la variable es de tipo “puntero a carácter”, pero en lugar de puntero a caracter podría ser de tipo puntero a cualquier tipo, tanto simple como compuesto. De hecho, como vemos mas adelante, el uso de puntero cobra importancia cuando se lo utiliza para apuntar a datos de tipo compuesto, es decir, registro.



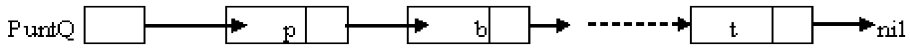
Listas encadenadas

Definición
Denominaremos lista encadenada al conjunto de elementos dispersos en memoria que representan una secuencia de datos.

Lista Simplemente Encadenada (LSE)
Es una estructura encadenada donde:

a) se conoce la dirección por el primer elemento.
b) cada elemento conoce la dirección de memoria donde está ubicado su sucesor.
c) es posible determinar cuando un elemento no tiene sucesor, es decir, es el último elemento.

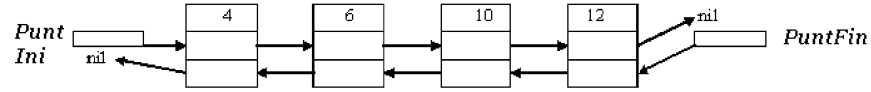
Gráficamente la representaremos de esta manera:



Listas encadenadas

Lista Doblemente Encadenada (LDE)
Es una estructura encadenada donde:
a) se conoce la dirección de memoria del primer y del último elemento.
b) cada elemento conoce la dirección de memoria donde está ubicado su predecesor y su sucesor.
c) es posible determinar cuando un elemento no tiene sucesor, es decir, es el último elemento.

Gráficamente la representaremos de esta manera:

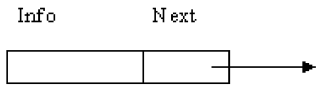


Esta estructura es un caso especial de la secuencia donde la misma puede recorrerse de adelante para atrás o a la inversa y en donde podemos retrocedes si es necesario.

Listas encadenadas

Dobletes
Denominaremos de esta manera a cada elemento que compone una lista **simplemente** encadenada. Un doblete se define mediante un registro formado por **dos campos:** uno para almacenar la información y otro para almacenar la dirección del siguiente elemento (sucesor).
El tipo de primer elemento puede ser cualquiera, tanto simple como compuesto, dependiendo de la información a almacenar. El segundo campo es de tipo puntero puesto que su función es almacenar la dirección de memoria donde se encuentra el siguiente elemento. De manera general podríamos definirlo y graficarlo de la siguiente forma:

Telem =< info ε tipo de la información, **next** ε puntero a Telem>



Listas encadenadas

Dobletes
Si los elementos fueran del tipo caracter, por ejemplo la definición sería:

TelemCar =< **info** ε Caracter, **next** ε puntero a TelemCar >

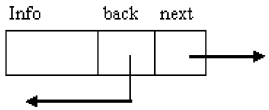
En cambio si los elementos fueran de tipo compuesto como por ejemplo de tipo auto, compuesto de marca, modelo y año, la definición sería la siguiente:

TAuto= < **marca** ε Cadena , **modelo** ε Cadena , **año** ε Entero >
TelemAuto = < **info** ε TAuto , **next** ε puntero a TelemAuto >

Es interesante observar que el campo next debe ser el mismo tipo del elemento, es decir TelemCar en el primer caso y TelemAuto en el segundo, puesto que las estructuras deben ser homogéneas, al igual que los arreglos.

Listas encadenadas

Tripletes
Denominaremos de esta manera a cada elemento que compone una lista **doblemente** encadenada.
Un triplete se define mediante un registro formado por **tres campos**: uno para almacenar la información, otro para almacenar la **dirección del elemento anterior** (predecesor) y un tercer campo para almacenar la **dirección del elemento siguiente** (sucesor).
El tipo del primer campo puede ser cualquiera, tanto simple como compuesto, dependiendo de la información a almacenar. El segundo y tercer campo son de tipo puntero puesto que su función es almacenar la dirección de memoria donde se encuentra el elemento anterior y el elemento siguiente.
Telem= <**info** ε tipo de la información, **back** ε puntero a Telem, **next**: puntero a Telem>



Listas encadenadas

Tripletes
Si los elementos fueran de tipo caracter, por ejemplo la definición sería :

TelemCar= < **info** ε Caracter, **back** ε puntero a TelemCar ,**next** ε puntero a TelemCar >

En cambio si lo elementos fueran de tipo compuesto con por ejemplo de tipo auto, compuesto por marca, modelo y año, la definición sería:

TAuto= < **marca** ε Cadena, **modelo** ε Cadena, **año** ε Entero >

TelemAuto= < **info** ε TAuto , **back** ε puntero a TelemAuto, **next** ε puntero a TelemAuto >

Es interesante observar que los campos **back** y **next** deben ser punteros del mismo tipo que el elemento, es decir **TelemCar** en el primer caso y **TelemAuto** en el segundo, puesto que las estructuras deben ser homogéneas.

Listas encadenadas

Operaciones y Notaciones
A efecto de poder manipular las estructuras dinámicas, mencionaremos la **notación algorítmica** a utilizar para las operaciones básicas.
Para facilitar su comprensión utilizaremos solamente los **dobletes,** ya que para **tripletes** es similar con el agregado de un campo más. Mencionaremos también las equivalencias con el **lenguaje C** para facilitar la traducción de los algoritmos a este lenguaje de programación.
Debido a que las estructuras dinámicas deben ser homogéneas y los elementos que la componen deben estar definidos, es decir tener un tipo determinado, cada estructura dinámica a utilizar debe definirse previamente. Se hace necesario por ello tener definido el tipo del doblete y declarada una variable cuyo tipo sea puntero al tipo del doblete deseado.
Por ejemplo, si deseamos trabajar con una estructura dinámica para almacenar números enteros, debemos definir como tipo del doblete un registro con dos campos uno de tipo entero y otro de tipo puntero a al doblete en cuestión.
Además necesitamos declarar una variable de tipo puntero al tipo del doblete:

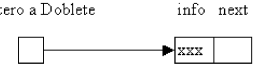
TdobleteEntero = < info ε Entero, **next** ε puntero a TdobleteEntero >
punteroADoblete ε puntero a **TdobleteEntero**

Listas encadenadas

Crear un doblete
La operación para crear un doblete se denomina **Obtener.** Esta operación tiene un parámetro que indica el nombre de la variable que apuntará al nuevo doblete. Dicha variable debe estar declarada previamente. Por ejemplo, para poder crear un nuevo doblete de tipo **TdobleteEntero**, es necesaria tener definida una variable de tipo **puntero a TdobleteEntero**. Supongamos que la variable es la definida en la página anterior, cuyo nombre es **punteroAdoblete**:

Obtener(punteroADoblete)
En este caso la operación **Obtener** obtiene un nuevo espacio de memoria del tipo de **TdobleteEntero** y asigna la dirección de ese doblete a la variable **punteroAdoblete.** De esta manera mediante la variable **punteroAdoblete** es posible acceder al doblete creado sin necesidad de saber en que lugar de la memoria se encuentra.

En C: `malloc(cantidad de memoria en bytes)`



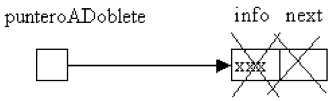
Listas encadenadas

Eliminar un doblete

Los dobletes no se destruyen automáticamente cuando finaliza el bloque donde fueron creados. Eso solo ocurre con las variables de tipo puntero que apuntan a los dobletes, debido a que son variables estáticas. Si no eliminamos cada doblete que dejamos de usar, corremos el riesgo de dejar parte de la memoria inutilizada. Cuando el doblete deja de ser apuntado es imposible volver a acceder al mismo.
Se debe utilizar una operación que se encarga de devolver la memoria ocupada por el doblete. Esta operación se denomina **Liberar** y lleva como parámetro el nombre de la variable que apunta al doblete.

En notación algorítmica: Liberar(nombrePuntero)
En C: `free(puntero a la memoria que se desea liberar)`

Ejemplo: **Liberar(punteroAdoblete)**



Liberar devuelve el espacio de memoria del tipo **TdobleteEntero**.

Listas encadenadas

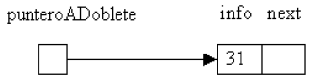
Guardar información en un doblete

Una vez creado el doblete estamos en condiciones de agregar la información deseada al mismo. La información debe ser almacenada en el campo correspondiente, "info" en nuestro caso, y el tipo de la información a almacenar debe coincidir con el tipo de dicho campo.
Como los dobletes son registros, el campo al cual deseamos acceder se selecciona de la misma manera que en un *registro* con la única diferencia que debemos indicar el lugar de memoria donde está ubicado dicho registro, o sea el doblete. Esto se logra mediante la información que tiene almacenada el puntero que apunta a dicho doblete.

En notación Algorítmica: (^nombrePuntero).info <- <expresión>
En C: nombrePuntero->info = <expresión>

Ejemplo: consideremos el doblete y la variable definidos previamente. Almacenemos el número **31** en el campo **info** del doblete.

(^punteroADoblete).info <- 31



Listas encadenadas

Acceder a información de un doblete

A los efectos de acceder a información almacenada en un doblete es necesario recuperar lo que está almacenado en el campo **info**. Como los dobletes son registros, el campo del cual deseamos obtener información se selecciona de la misma manera que en un registro, con la única diferencia que debemos indicar el lugar de memoria en donde está ubicado dicho registro. Este se logra mediante la información que tiene almacenada el puntero que apunta a dicho doblete.

En notación algorítmica: (^nombrePuntero).info
En C: nombrePuntero->info

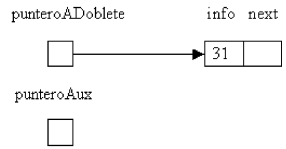
Ejemplo: almacenemos en la variable **val** el valor almacenado en **info**, incrementemos uno a **val**, y finalmente mostremos su nuevo valor.

val <- (^punteroADoblete). info
val <- val + 1
Salida: (^punteroADoblete).info
Salida: val

Listas encadenadas

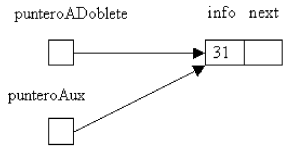
Asignación de direcciones entre punteros

Muchas veces es necesario que más de una variable (de tipo puntero) apunte a una dirección determinada de memoria donde se encuentra alojado un doblete; o que un puntero cambien la dirección a la cual esta apuntando. Para realizar este tipo de tareas deben realizarse asignaciones. Estas asignaciones involucran variables de tipo puntero y campo de tipo puntero de algún doblete (next en nuestro ejemplo). No se trata de nada nuevo sino de asignaciones entre variables. Lo único de tener en cuenta es que debemos asignar direcciones de memoria.
Supongamos declarada otra variable **punteroAux**.



punteroAux ε puntero a TdobleteEntero

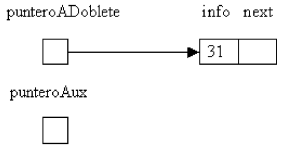
punteroAux <- punteroADoblete producirá:



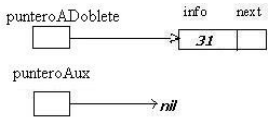
Listas encadenadas

La dirección nil

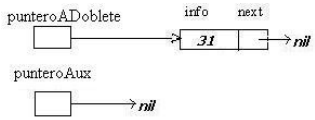
Nil representa una dirección especial de memoria, que significa **"ninguna parte"**, es decir es un elemento especial del tipo puntero que se utiliza para indicar que un puntero o el campo de tipo puntero de un doblete no apunta a nadie. Generalmente se utiliza para representar la finalización de un encadenamiento de datos.
Veamos a continuación un ejemplo:



punteroAux <-nil producirá:



(^punteroADoblete).next <- nil producirá:



Listas encadenadas
Ejercicios

1) Usando LSE, construir una lista vacía. Suponga declarado tipo **TelemCar**, el puntero **sec**, y todos los que necesite.

Algoritmo Ej1
Lexico
TelemCar =< info ε Caracter, next ε puntero a TelemCar >
sec ε puntero a TelemCar
Inicio

?

Fin

Listas encadenadas

2) Usando LSE, construir algorítmicamente la LSE para alojar la palabla "HOLA" (un carácter por doblete). Colocar pre-condición, pos-condición, y estados intermedios. Suponga declarado tipo **TelemCar**, el puntero **sec** y todos los que necesite.

Algoritmo Ej2
Lexico
TelemCar =< info ε Caracter, next ε puntero a TelemCar >
sec ε puntero a TelemCar
Inicio

?

Fin

Listas encadenadas

3) Dada una LSE, y las variables de tipo puntero r, t, s y q. Describa gráficamente los estados intermedios y el estado final, partiendo de una LSE donde ya está la palabra HOLA (4 dobletes) y el siguiente segmento de programa.

Algoritmo Ej3
Lexico
TelemCar =< info ε Caracter, next ε puntero a TelemCar >
q, r, w, x ε puntero a TelemCar
Inicio
Obtener(r)
(^r).info ← 'C'
(^r).next ← q
q ← r
Obtener(w)
(^w).info ← 'A'
(^w).next ← q
q ← w
Obtener(x)
(^x).info ← 'A'
(^x).next ← q
q ← x
Fin

Listas encadenadas en C

```
//Telem =< info ε tipo de la información, next ε puntero a Telem>
struct Telem {
            tipo de la información info;
            struct Telem *next;
                };

struct Telem *p;
p=(struct Telem *) malloc(sizeof(struct Telem));

//o como un sinónimo para el tipo de dato
typedef struct Telem *Lista;
Lista p;
p=(Lista)malloc(sizeof(struct Telem));

// TelemCar =< info ε Caracter, next ε puntero a TelemCar >
struct TelemCar {
            char info;
            struct TelemCar *next;
                    };
```


Listas encadenadas en C

```
// TAuto= < marca ε Cadena , modelo ε Cadena , año ε Entero >
//TelemAuto = < info ε TAuto , next ε puntero a TelemAuto >
struct TAuto {
            char marca[30];
            char modelo[30];
            int anio;
           };
struct TelemAuto {
                struct TAuto info;
                struct TelemAuto *next;
              };

// TelemCar= < info ε Caracter, back ε puntero a TelemCar ,next ε puntero a
TelemCar >
struct TelemCar {
            char info;
            struct TelemCar *back;
            struct TelemCar *next;
           };
```


Listas encadenadas en C

```
//TAuto= < marca ε Cadena, modelo ε Cadena, año ε Entero >
//TelemAuto= < info ε TAuto , back ε puntero a TelemAuto, next ε puntero a
TelemAuto >
struct TAuto {
            char marca[30];
            char modelo[30];
            int anio;
           };
struct TelemAuto {
                struct TAuto info;
                struct TelemAuto *back;
                struct TelemAuto *next;
              };

//TdobleteEntero = < info ε Entero, next ε puntero a TdobleteEntero >
//punteroADoblete ε puntero a TdobleteEntero
struct TdobleteEntero {
                    int info;
                    struct TdobleteEntero *next;
                   };
struct TdobleteEntero *punteroADoblete;
```


Listas encadenadas en C

```
//Obtener(punteroADoblete)
punteroADoblete=(struct TdobleteEntero *) malloc(sizeof(struct TdobleteEntero));

// Liberar(punteroAdoblete)
free(punteroADoblete);

// (^punteroADoblete).info <- 31
punteroADoblete->info=31

// val <- (^punteroADoblete).info
//val <- val + 1
//Salida: (^punteroADoblete).info
//Salida: val

val=punteroADoblete->info;
val=val+1;
printf("%d",punteroADoblete->info);
printf("%d",val);
```

Listas encadenadas en C

```
//punteroAux ε puntero a TdobleteEntero
//punteroAux <- punteroADoblete
struct TdobleteEntero *punteroAux;
punteroAux=punteroADoblete;

// punteroAux <-nil
//(^punteroADoblete).next <- nil
punteroAux=NULL;
punteroADoblete->next=NULL;
```


Listas encadenadas en C

```
//Ej 1
#include <stdio.h>

struct TelemCar {
                char info;
                struct TelemCar *next;
              };

int main() {
  struct TelemCar *sec;

  sec = (struct TelemCar *) NULL;
  return(0);
}
```


Listas encadenadas en C

```
//Ej 2
#include <stdio.h>
struct TelemCar {
                char info;
                struct TelemCar *next;
              };
int main() {
  struct TelemCar *nuevo, *sec;
  nuevo = (struct TelemCar *) NULL;
  sec = (struct TelemCar *) NULL;
  nuevo = (struct TelemCar *) malloc (sizeof(struct TelemCar));
  nuevo->info='A';
  nuevo->next=NULL;
  sec =nuevo;
  nuevo = (struct TelemCar *) malloc (sizeof(struct TelemCar));
  nuevo->info='L';
  nuevo->next=sec;
  sec=nuevo;
  nuevo = (struct TelemCar *) malloc (sizeof(struct TelemCar));
  nuevo->info='O';
  nuevo->next=sec;
  sec=nuevo;
  nuevo = (struct TelemCar *) malloc (sizeof(struct TelemCar));
  nuevo->info='H';
  nuevo->next=sec;
  sec=nuevo;
  return(0);
}
```


Listas encadenadas en C

```
//Ej 3
#include <stdio.h>
tructTelemCar {
                char info;
                struct TelemCar *next;
              };
void anadir_palabra_hola(struct TelemCar **primero) {
  struct TelemCar *nuevo;
  nuevo = (struct TelemCar *) malloc (sizeof(struct TelemCar));
  if (nuevo==NULL) printf( "No hay memoria disponible!\n");
  else {
    nuevo->info='A';
    nuevo->next=NULL;
    *primero=nuevo;
    nuevo = (struct TelemCar *) malloc (sizeof(struct TelemCar));
    nuevo->info='L';
    nuevo->next=*primero;
    *primero=nuevo;
    nuevo = (struct TelemCar *) malloc (sizeof(struct TelemCar));
    nuevo->info='O';
    nuevo->next=*primero;
    *primero=nuevo;
    nuevo = (struct TelemCar *) malloc (sizeof(struct TelemCar));
    nuevo->info='H';
    nuevo->next=*primero;
    *primero=nuevo;
  }
}
```

Listas encadenadas en C

```
//Ej 3continuacion

void mostrar_lista(struct TelemCar *primero) {
  struct TelemCar *auxiliar; /* lo usamos para recorrer la lista */
  int i;

  i=0;
  auxiliar=primero;
  printf("\nMostrando la lista completa:\n");
  while (auxiliar!=NULL) {
    printf("%c",auxiliar->info);
    auxiliar=auxiliar->next;
    i++;
  }
  if (i==0) printf( "\nLa lista esta vacia!\n" );
}
```


Listas encadenadas en C

```
//Ej 3 continuacion

int main() {
  struct TelemCar *q, *r, *w, *x;

  r = (struct TelemCar *) NULL;
  w = (struct TelemCar *) NULL;
  x = (struct TelemCar *) NULL;
  q = (struct TelemCar *) NULL;

  anadir_palabra_hola(&q);
  r=(struct TelemCar *) malloc(sizeof(struct TelemCar));
  r->info='C';
  r->next=q;
  q=r;
  w=(struct TelemCar *) malloc(sizeof(struct TelemCar));
  w->info='B';
  w->next=q;
  q=w;
  x=(struct TelemCar *) malloc(sizeof(struct TelemCar));
  x->info='A';
  x->next=q;
  q=x;
  mostrar_lista(q);
  return(0);
}
```


Listas encadenadas en C

Ejemplo de Agenda de Contactos en C

Ejercicio

- Descargar el archivo C en la sección Materiales -> Software
- Usar el programa (libertad 0 del software libre).
- Estudiar cómo funciona el programa y adaptarlo a tus necesidades (libertad 1 del software libre).
- Compartirlo con tus compañeros (libertad 2 del software libre).
- Mejorar el programa completando la opción **"2.- Borrar elementos"**, y compartirlo con tus compañeros (libertad 3 del software libre).

Bibliografía

– Scholl, P. y Peyrin, J.-P. "Esquemas Algorítmicos Fundamentales: Secuencias e iteración".

– Biondi, J. y Clavel, G. "Introducción a la Programación. Tomo 1: Algorítmica y Lenguajes".

– Clavel, G. y Biondi, J. "Introducción a la Programación. Tomo 2: Estructuras de Datos".