UAEM | Universidad Autónoma del Estado de México

Estructura de Datos

Centro Universitario UAEM Valle de México

Centro Universitario UAEM Valle de México

Unidad II. Estructuras de Datos Lineales - Listas

Licenciatura en Ingeniería en Computación

Ph. D. Victor Manuel Landassuri Moreno
vmlandassurim@uaemex.mx
landassuri@gmail.com

Índice de Contenidos

- Visión global de la Unidad de Aprendizaje (UDA)
- Descripción de la unidad de aprendizaje
- Unidad 2. Estructuras de Datos Lineales - Listas
- Guión Explicativo
- Resumen
- Ejercicios Propuestos
- Bibliografía

PROGRAMA DE ESTUDIO POR COMPETENCIAS

ESTRUCTURAS DE DATOS

I. IDENTIFICACIÓN DEL CURSO

<table>
<tr><td colspan="7">ESPACIO EDUCATIVO: Facultad de Ingeniería</td></tr>
<tr><td colspan="3">LICENCIATURA: Ingeniería en Computación

AÑO DE APROBACIÓN POR EL CONSEJO UNIVERSITARIO:</td><td colspan="4">ÁREA DE DOCENCIA: Programación e Ingeniería de Software</td></tr>
<tr><td colspan="2">APROBACIÓN POR LOS HH. CONSEJOS ACADÉMICO Y DE GOBIERNO</td><td>FECHA:</td><td colspan="2">PROGRAMA ELABORADO POR:
Mtro. Eduardo Trujillo Flores
Mtra. Mireya Salgado Gallegos

FECHA DE ELABORACIÓN :
Mayo de 2007</td><td colspan="2">PROGRAMA REVISADO POR:
Integrantes de la Academia de Programación e Ingeniería de Software

Centro Universitario Valle de Chalco

FECHA DE REVISIÓN :
Mayo 2011</td></tr>
<tr><td>CLAVE</td><td>HORAS DE TEORÍA</td><td>HORAS DE PRÁCTICA</td><td>TOTAL DE HORAS</td><td>CRÉDITOS</td><td>TIPO DE CURSO</td><td>NÚCLEO DE FORMACIÓN</td></tr>
<tr><td>L41054</td><td>3</td><td>2</td><td>5</td><td>8</td><td>Curso</td><td>Sustantivo</td></tr>
<tr><td colspan="3">UNIDAD DE APRENDIZAJE ANTECEDENTE
Programación estructurada</td><td colspan="4">UNIDAD DE APRENDIZAJE CONSECUENTE
Organización de Archivos</td></tr>
<tr><td colspan="7">PROGRAMAS EDUCATIVOS O ESPACIOS ACADÉMICOS EN LOS QUE SE IMPARTE:
Licenciatura en Ingeniería en Computación (Facultad. de Ingeniería, Centros Universitarios: Atlacomulco, Ecatepec, Texcoco, Valle de Chalco, Valle de México, Valle de Teotihuacán, Zumpango)</td></tr>
</table>

Descripción de la unidad de aprendizaje

Identificación del Curso

Licenciatura en Ingeniería en Computación

Horas de Teoría: 3 hrs.

Horas de Práctica: 2 hrs.

Créditos: 8

Unidad de Aprendizaje Antecedente:
Programación Estructurada

Unidad de Aprendizaje Consecuente:
Organización de Archivos

Presentación

El estudio de las estructuras de datos, sin duda es uno de los más importantes dentro de las carreras relacionadas con la Computación, ya que en el conocimiento eficiente de las estructuras de datos suele ser imprescindible en la formación de los alumnos debido a la trascendencia que un aprendizaje teórico-práctico de las misma supondrá en su carrera.

Lineamientos

Del Profesor

- Cumplir en tiempo y contenido el programa de la unidad de aprendizaje
- Establecer tolerancia para el inicio de clases
- Proponer y respetar formas de evaluación
- Generar en sus alumnos una visión integradora de la unidad de aprendizaje
- Respetar el número de horas teóricas y practicas de la unidad de aprendizaje
- Cada sesión deberá concluir con una serie de ejercicios de repaso que permitirán reafirmar los conocimientos del curso
- Preferentemente las fechas de exámenes y entrega del proyecto final deberán establecerse desde el principio

Del Alumno

- Contar con el 80% de asistencia para presentar examen ordinario
- Contar con el 60% de asistencia para presentar examen extraordinario
- Contar con el 30% de asistencia para presentar examen a titulo de suficiencia
- Utilizar un lenguaje estructurado para la elaboración de programas
- Tener sentido de responsabilidad en los trabajos clase y extraclase
- Entregar en tiempo y forma los trabajos clase y extraclase
- Tener sentido de interrogación y participación dentro del salón de clases
- El alumno deberá entregar todas y cada una de las series de ejercicios.

Propósito

Que el alumno identifique las herramientas teóricas fundamentales para la representación y manipulación de información en la computadora haciendo énfasis en el tipo de datos dinámicos.

Competencias genéricas

Desarrollar programas analizando y diseñando soluciones a problemas reales del entorno a través del uso de herramientas lógicas

Ámbito de desempeño

Empresas públicas y privadas del sector industrial, educativo, comercial y de servicios.

Estructura

Unidad 1.	• Reconocer y manejar las variables dinámicas
Unidad 2.	• Aplicar las principales estructuras de datos lineales
Unidad 3.	• Aplicar la estructura de datos de árbol
Unidad 4.	• Aplicar la estructura de datos de grafo

Estructura por unidad

- **Unidad 1.** Reconocer y manejar las variables dinámicas
  - *Estructuras de datos*: Definición, Tipos.
  - *Abstracción*: Definición. TAD.
  - *Variables Dinámicas*: Apuntadores, Operaciones básicas.

Estructura por unidad

- **Unidad 2.** Aplicar las principales estructuras de datos lineales
  - *Pilas*: Representación. Operaciones (Inserción, Eliminación, Pila Llena, Pila Vacía). Tratamiento de expresiones aritméticas: notación infija, prefija, posfija. aplicaciones
  - *Colas*: Representación. Operaciones (Inserción, Eliminación, Cola Llena, Cola Vacía). Cola circular. Aplicaciones.
  - *Listas*. Representación. Operaciones (Inserción, Eliminación, Recorrido, Búsqueda). Listas doblemente ligadas. Aplicaciones

Estructura por unidad

- **Unidad 3.** Aplicar la estructura de datos de árbol
  - *Recursividad Directa*: Definición. Funcionamiento
  - *Árboles*: Usos. Características.
  - *Árboles binarios de Expresiones*: Características. Evaluación de expresiones aritméticas.
  - *Árboles Binarios de Búsqueda*: Características. Operaciones (Inserción, eliminación, búsqueda)

Estructura por unidad

- **Unidad 4.** Aplicar la estructura de datos de grafo
  - *Grafos*: Características. Tipos. Representación y construcción. Operaciones Librería estándar, interfaz con el sistema operativo
  - *Grafos Dirigidos*: Algoritmos para la obtención del camino más corto.
  - *Grafos No dirigidos*: Algoritmos para la obtención de costo mínimo

Procedimientos de Evaluación

La calificación ordinaria se obtiene como la suma del Trabajo semestral y el Proyecto final.
Para tener derecho a presentar el Proyecto final se debe obtener un promedio mínimo de 6.0/10.0 en el trabajo semestral.
Trabajo semestral:

- Programas producto y actividades clase y extra clase **30%**
- Exámenes parciales escritos **20%**
- Proyecto Final **50%**

Se podrá exentar el Proyecto final siempre y cuando se tenga un mínimo de 80% de asistencias, se aprueben los exámenes parciales y el promedio del Trabajo semestral sea de al menos 8.0/10.0
Extraordinaria y a Título de Suficiencia:

- Examen escrito **60%**
- Proyecto **40%**

Unidad 2.

Estructuras de Datos Lineales - Listas

Contenido

2.1 Representación de Listas

2.2 Operaciones

- 2.2.1 Inserción
- 2.2.2 Eliminación
- 2.2.3 Recorrido
- 2.2.4 Búsqueda

2.3 Listas doblemente ligadas

2.4 Aplicaciones

2.1 Representación de Listas

Una lista es una estructura lineal de datos que se define:

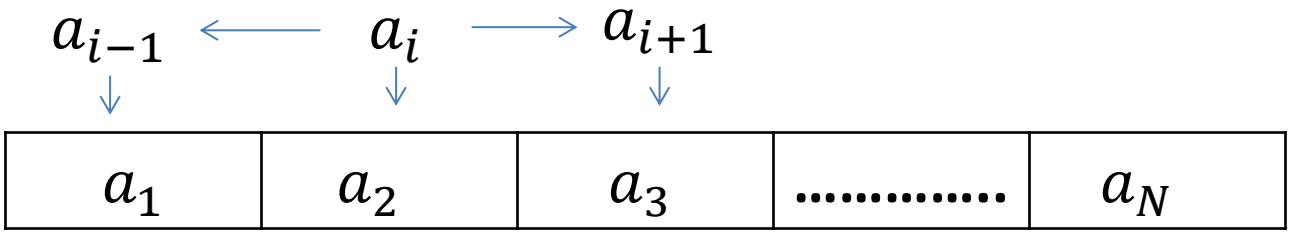
- En su forma general, como un conjunto de ***N*** elementos ($L = \{a_1, a_2, a_3, \ldots, a_N\}$) que la conforman.

a_1	a_2	a_3	.....	a_N

- En su forma especial, donde no existe ningún elemento, se le conoce como lista nula o vacía ($L_\emptyset$).
- El primer elemento de la lista es a_1 y el último es a_N

2.1 Representación de Listas

- Para cualquier lista, excepto para la nula, se dice que a_{i+1} es sucesor de a_i donde *(i < N)* y que a_{i-1} es predecesor de a_i donde *(i > 1).*



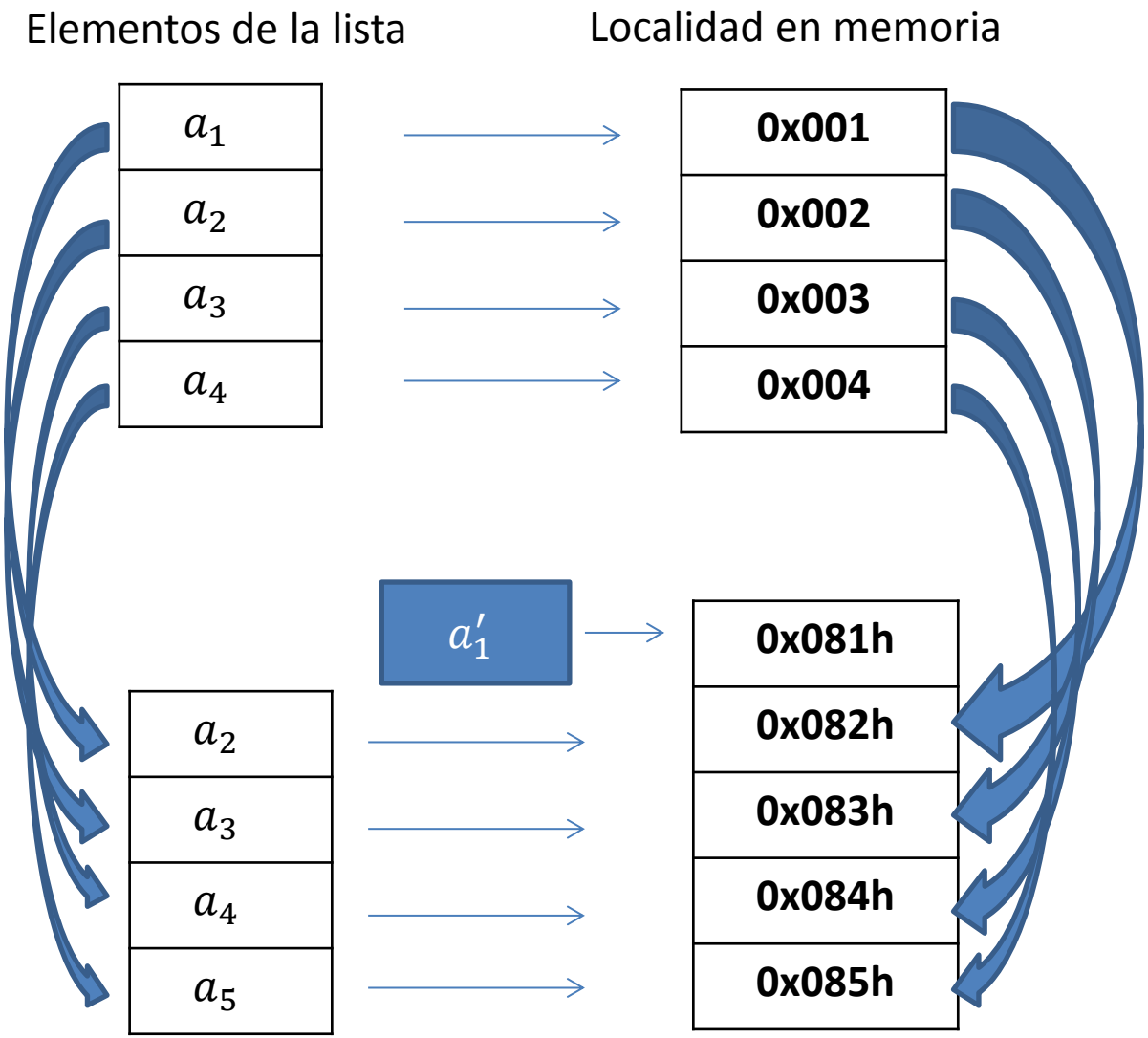
- No se define ni el predecesor de a_1 ni el sucesor de a_N.
- La posición del elemento a_i en una lista es *i.*

2.1 Representación de Listas

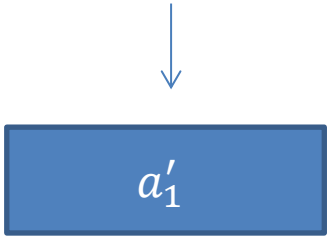
Una **FORMA SIMPLE** de **IMPLEMENTAR** una **LISTA** es mediante un **ARREGLO**, donde previamente se declara la cantidad de elementos que tendrá y se reserva en la memoria registros contiguos. **SIN EMBARGO**, las **OPERACIONES** como **INSERCIÓN** y **ELIMINACIÓN** resultan computacionalmente **COSTOSAS**.

Por ejemplo, la inserción en la posición 0 requiere empujar primero todo el arreglo a una posición para hacer espacio, mientras que eliminar el primer elemento requiere desplazar una posición hacia adelante todos los elementos de la lista, asi que en el peor de los casos estas operaciones son *O(n).*

2.1 Representación de Listas



Nuevo elemento a insertar la principio de la lista



Reservación de un nuevo arreglo en memoria *N+1* y se copian los *N* elementos

2.1 Representación de Listas

A fin de evitar el costo lineal de inserción y eliminación, necesitamos asegurar que la lista no se almacene contiguamente, ya que de otra forma necesitarán moverse partes completas de la lista.

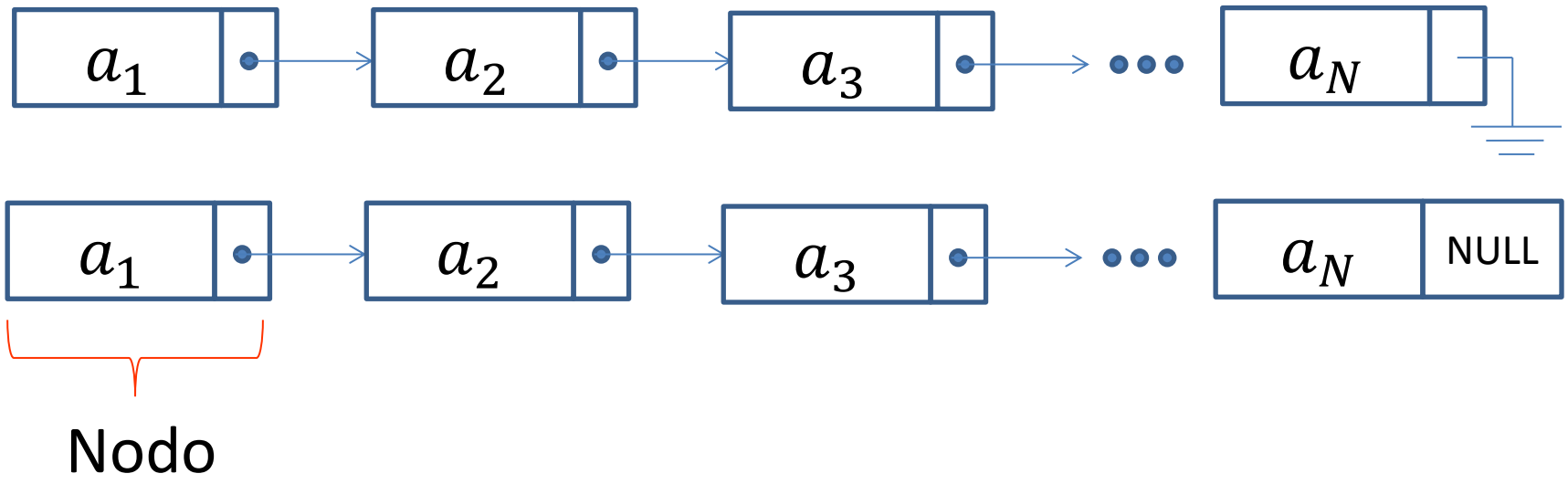
La **LISTA ENLAZADA** consta de una serie de registros que no necesariamente son adyacentes a la memoria.

Cada registro contiene el elemento y un apuntador que contiene su sucesor, a éste se le llama apuntador *siguiente*.

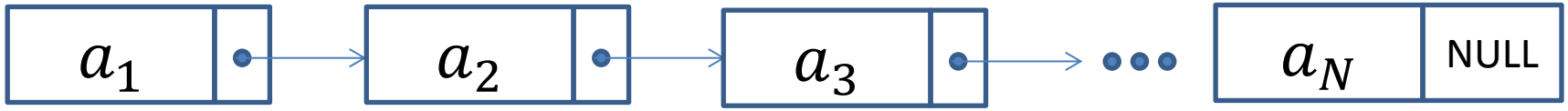
El apuntador *siguiente* de la ultima celda se apunta a *null,* es decir, no contiene una dirección de memoria.

2.1 Representación de Listas

La representación gráfica más extendida es aquella que utiliza una caja o rectángulo con dos secciones en su interior. En la primera sección se escribe el elemento o valor del dato y en la segunda sección, el enlace o puntero mediante una flecha que sale de la caja y apunta al nodo siguiente.

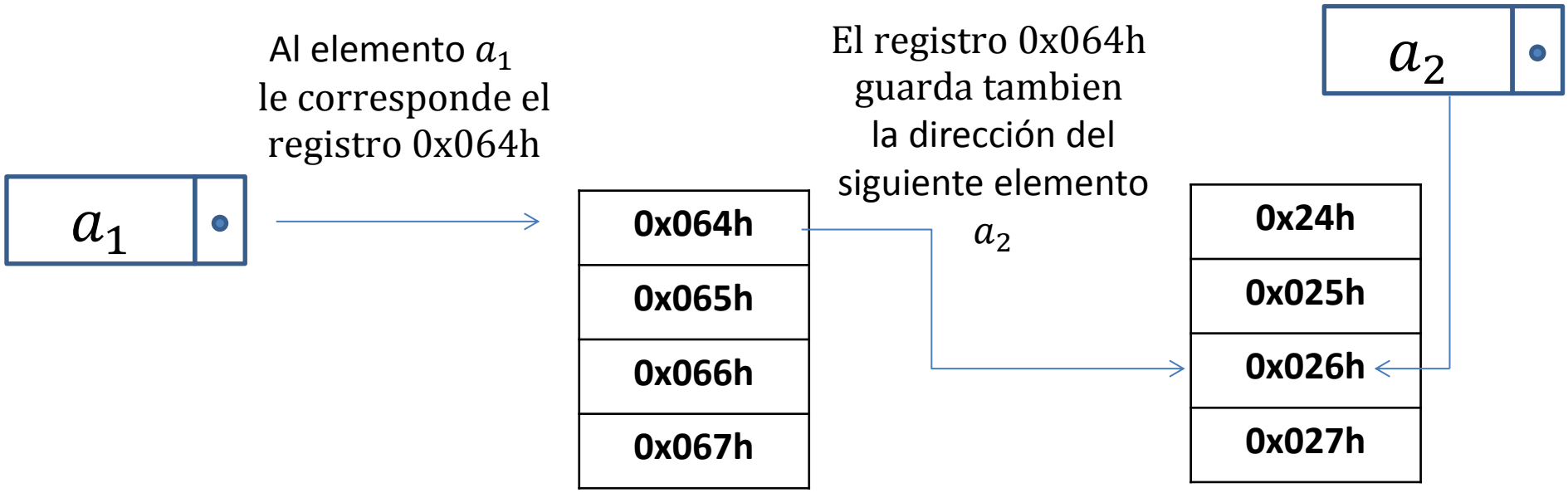


2.1 Representación de Listas



Al elemento a_1 le corresponde el registro 0x064h

El registro 0x064h guarda tambien la dirección del siguiente elemento a_2

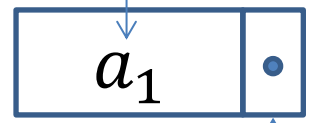


Localidad en memoria

2.1 Representación de Listas

```
struct Nodo
{
    int dato;
    Nodo *siguiente;
}
```

El elemento puede ser de cualquier tipo como float, char, double, double float, struct, etc.



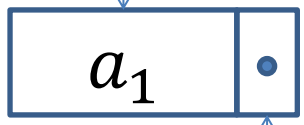
Guarda la dirección del siguiente nodo o elemento de la lista.

Apuntador a otra estructura tipo Nodo.

2.1 Representación de Listas

El elemento puede ser de cualquier tipo como float, char, double, double float, class, etc.

```
class Nodo {

public:
      int dato;
      Nodo *siguiente;
       //constructor
}
```



Guarda la dirección del siguiente objeto de la lista.

Apuntador a otra objeto tipo Nodo.

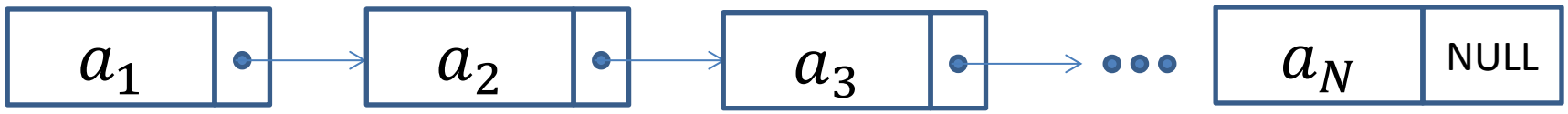
2.1 Representación de Listas

Las listas se pueden dividir en cuatro categorías:

- Listas simplemente enlazadas.
- Listas doblemente enlazadas.
- Lista circular simplemente enlazada.
- Lista circular doblemente enlazada.

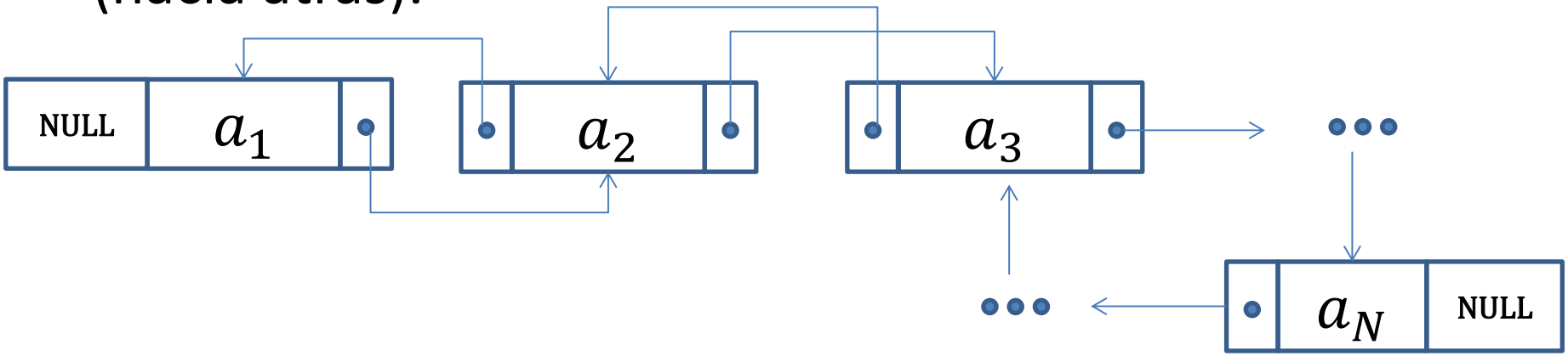
2.1 Representación de Listas

Listas simplemente enlazadas. Cada nodo (elemento) contiene un único enlace que conecta ese nodo siguiente o sucesor. La lista es eficiente en recorridos directos (hacia adelante).



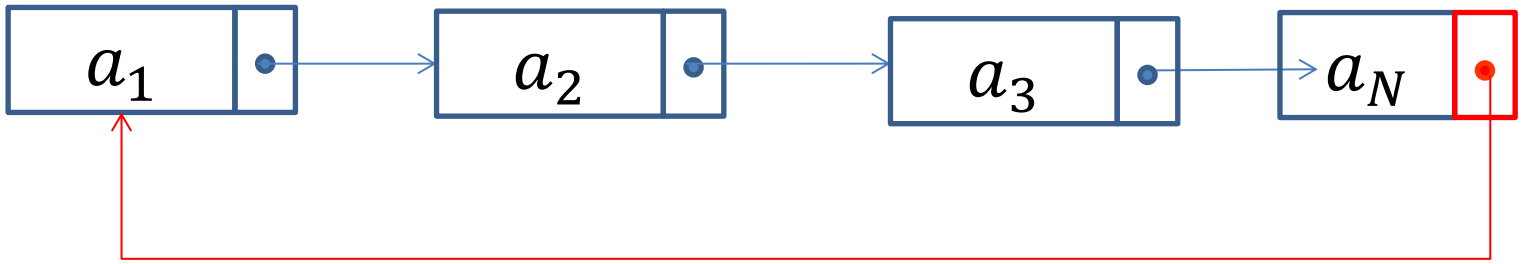
2.1 Representación de Listas

Listas doblemente enlazadas. Cada nodo contiene dos enlaces, uno a su nodo predecesor y el otro a su sucesor. La lista es eficiente tanto en recorrido directo (hacia adelante) como recorrido inverso (hacia atrás).



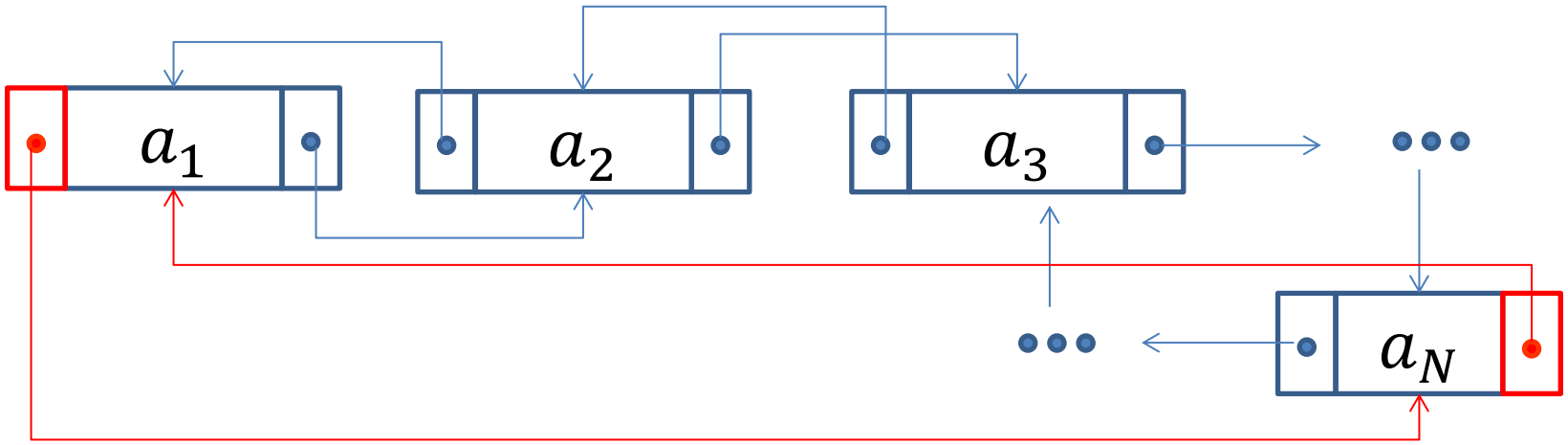
2.1 Representación de Listas

Listas circular simplemente enlazada. Una lista enlazada simplemente en la que el último elemento (cola) se enlaza al primer elemento (cabeza) de tal modo que la lista puede ser recorrida de modo circular (anillo)



2.1 Representación de Listas

Listas circular doblemente enlazada. Una lista doblemente enlazada en la que el último elemento se enlaza al primer elemento y viceversa. Esta lista se puede recorrer de modo circular (en anillo) tanto en dirección directa (adelante) como viceversa (atrás)



2.1 Representación de Listas

Por cada uno de estos cuatro tipos de estructuras de listas, se puede elegir una implementación basada en arrays o una implementación basada en apuntadores. Como ya se ha comentado, estas implementaciones difieren en el modo en que se asigna la memoria para los datos de los elementos, cómo se enlazan juntos los elementos y cómo se accede a dichos elementos.

2.1 Representación de Listas

De forma más específica, las implementaciones pueden hacerse con cualquiera de estas asignaciones:

- Fija de memoria mediante array
- Dinámica de memoria mediante punteros

Dado que la asignación fija de memoria mediante arrays es más eficiente. En este curos utilizaremos la asignación de memoria mediante apuntadores.

2.2 Operaciones en Listas

Una lista enlazada requiere controles para la gestión de los elementos contenidos en ellas. Estos controles se manifiestan en forma de operaciones que tendrán las siguientes tareas:

- Inicialización o creación, con declaración de los nodos.
- Insertar elementos en una lista.
- Eliminar elementos de una lista.
- Búsqueda de elementos de una lista (comprobar la existencia de elementos).
- Recorrer una lista enlazada.
- Comprobar si la lista esta vacía.

2.2.1 Inserción

El algoritmo empleado para añadir o insertar un elemento en una lista enlazada varía dependiendo de la posición en que se desea insertar el elemento. La posición de inserción puede ser:

- En la cabeza (elemento primero) de la lista.
- En el final de la lista (elemento último).
- Antes de un elemento especificado.
- Después de un elemento especificado.

2.2.1 Inserción

INSERTAR un nuevo elemento en la CABEZA de una LISTA.

Aunque normalmente se insertan nuevo datos al final de una estructura de datos, es más fácil y más eficiente insertar un elemento nuevo en la cabeza de una lista. El proceso de inserción se puede resumir en el siguiente algoritmo:

2.2.1 Inserción

1. Asignar un nuevo nodo apuntador *eltolista*, que es una variable puntero local que apunta al nuevo nodo que se va a insertar en la lista.

2. Situar el nuevo elemento en el campo de *elemento* del nuevo nodo.

3. Hacer que el campo de enlace *resto* del nuevo nodo **APUNTE A LA CABEZA** (primer nodo) de la lista original.

4. Hacer que el nodo *lista* apunte al nuevo nodo que se ha creado

2.2.1 Inserción

Para ejemplificar la implementación de una lista, se utilizará el siguiente código que crea una lista de números del 1 al 10.

2.2.1 Inserción

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

struct ElementoLista
{
        int elemento;
        ElementoLista *resto;
};

void crearVacia(ElementoLista **);
void Insertar(ElementoLista **,int);
```

```
int main()
{

    ElementoLista *lista;

    int i;

    crearLista(&lista);

    for(i=0;i<10;i++)
        Insertar(&lista,i);


    return 0;
}
```

2.2.1 Inserción

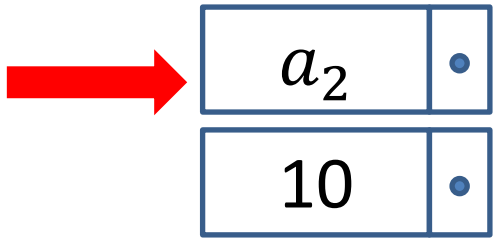
```
void crearLista(ElementoLista **lista)
{
      *lista = NULL;

}
```

```
void Insertar(ElementoLista **lista,int elemento)
{
        struct ElementoLista *eltolista;

        eltolista=(struct ElementoLista*)malloc(sizeof(struct ElementoLista));

        eltolista -> resto = NULL;
        eltolista -> elemento = elemento;

        eltolista -> resto = *lista;
        *lista = eltolista;
}
```

2.2.1 Inserción

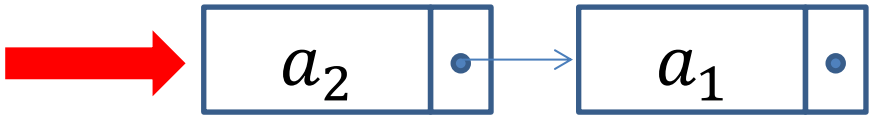
Paso 1. Se crea el primer nodo

Paso 2. Se crea un nuevo nodo (apuntador local) y se asigna el dato

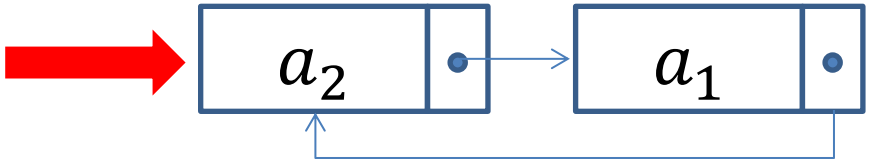


$a_2 = 10$

Paso 3. Hacer que el nuevo nodo apunte al primer nodo



Paso 4. El primer nodo apunta al nuevo nodo creado



2.2.1 Inserción

INSERTAR un nuevo elemento al FINAL de una LISTA.

1. Asignar un nuevo nodo apuntador *eltolista*, que es una variable puntero local que apunta al nuevo nodo que se va a insertar en la lista.

2. Situar el nuevo elemento en el campo de *elemento* del nuevo nodo.

3. Si la lista está vacía, asignar al primer nodo el nuevo nodo creado

4. Si la lista no está vacía, recorrer la lista hasta encontrar un nodo en el que el campo *resto* sea nulo.

2.2.1 Inserción

```
int esVacia(ElementoLista **lista)
{
      if(*lista == NULL)
            return 1;

      else
            return 0;

}
```

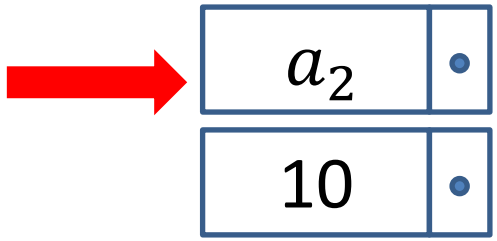
2.2.1 Inserción

```
void Insertar(ElementoLista **primero,int elemento)
{
        struct ElementoLista *eltolista;
        struct ElementoLista *lista;

        eltolista=(struct ElementoLista*)malloc(sizeof(struct ElementoLista));

        eltolista -> resto = NULL;
        eltolista -> elemento = elemento;

        if(esVacia(primero))
                *primero = eltolista;
        else

        {
                lista = *primero;
                while(lista->resto!=NULL)
                        lista = lista->resto;

                lista->resto = eltolista;

        }

}
```

2.2.1 Inserción

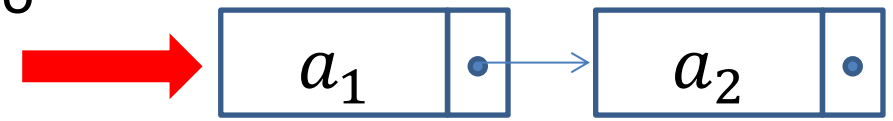
Paso 1. Se crea el primer nodo

Paso 2. Se crea un nuevo nodo (apuntador local) y se asigna el dato

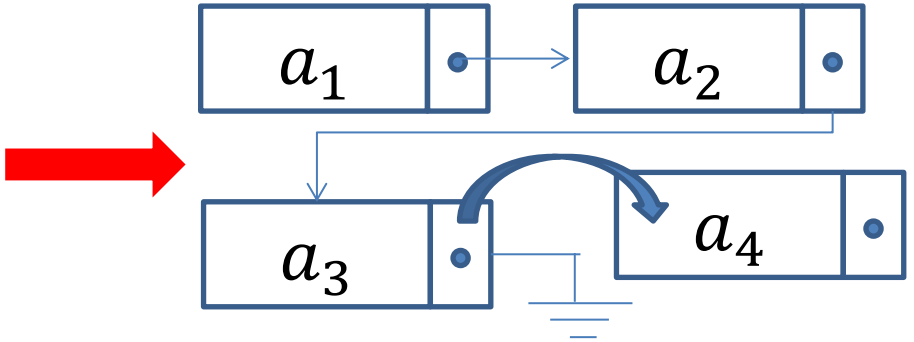


$a_2 = 10$

Paso 3. Hacer que el primer nodo apunte al nuevo nodo

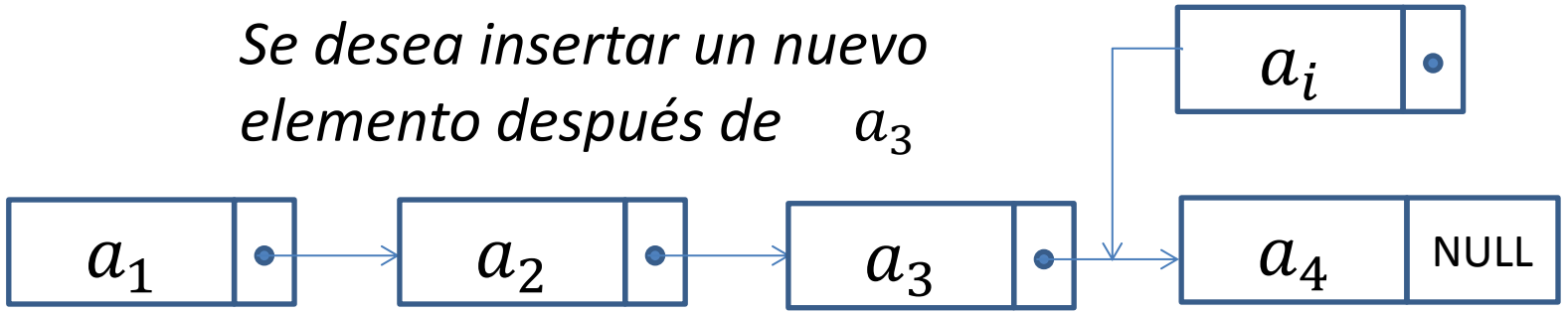


Paso 4. Recorrer la lista hasta encontrar un nodo con el *resto* vacío.

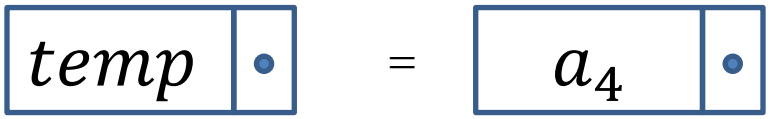


2.2.1 Inserción

INSERTAR un nuevo elemento en la ANTES o DESPUÉS de un elemento de la LISTA.

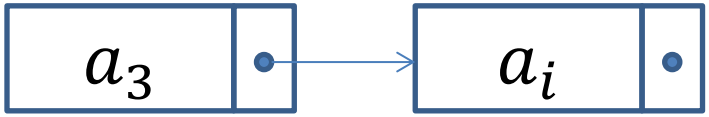


Paso 1. Asignar a un apuntador local la dirección de enlace al siguiente elemento del nodo en el que se va realizar la inserción.

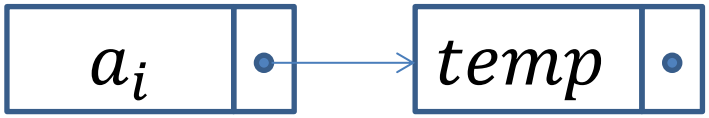


2.2.1 Inserción

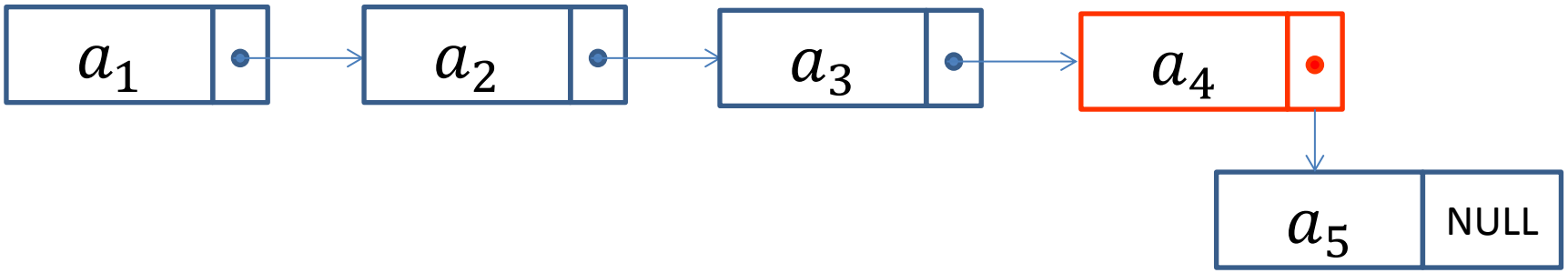
Paso 2. Asignar la dirección del nuevo nodo creado al apuntador de enlace del nodo actual.



Paso 3. Asignar la dirección del apuntador temporal *(tmp)* al apuntador de enlace del nuevo nodo insertado

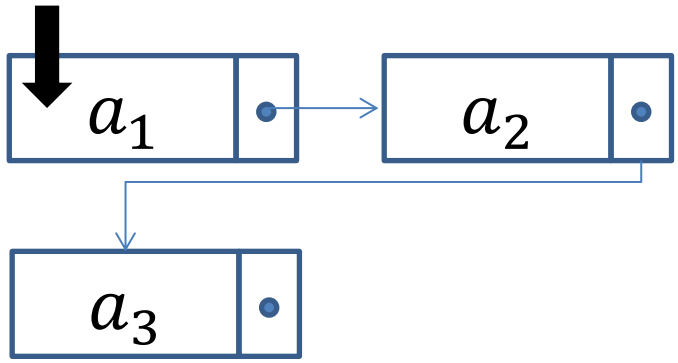


Paso 4. Se reacomodan los índices de la lista.

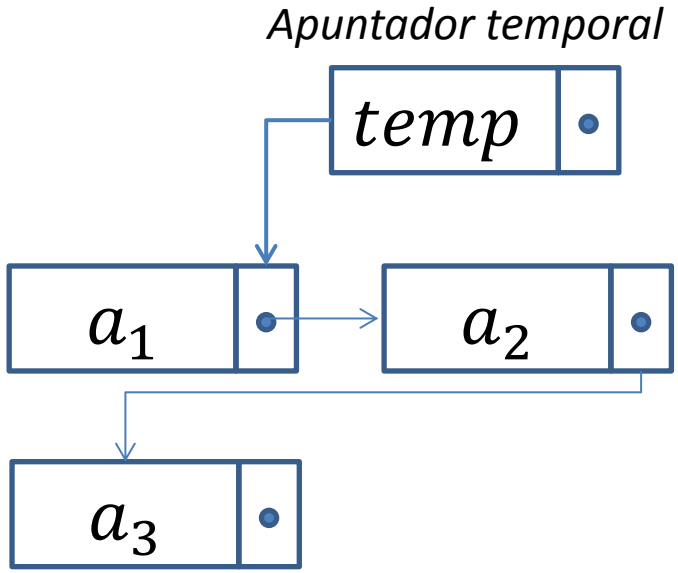


2.2.2 Eliminación

Paso 1. Posicionamiento en el nodo de cabecera

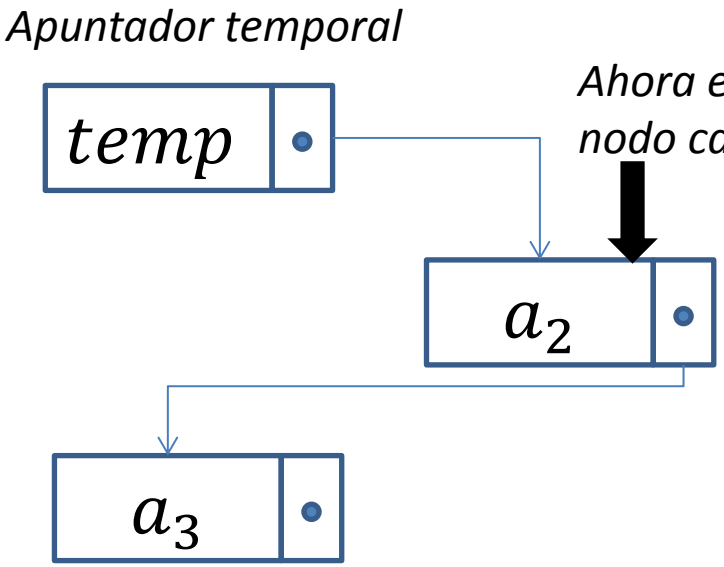


Paso 2. Asignar a un apuntador local la dirección del nodo de enlace del nodo de cabecera



2.2 Eliminación

Paso 3. Liberar el nodo cabecera



Paso 4. Asignar el nodo cabecera el nodo al que apunta el apuntador local

2.2.2 Eliminación

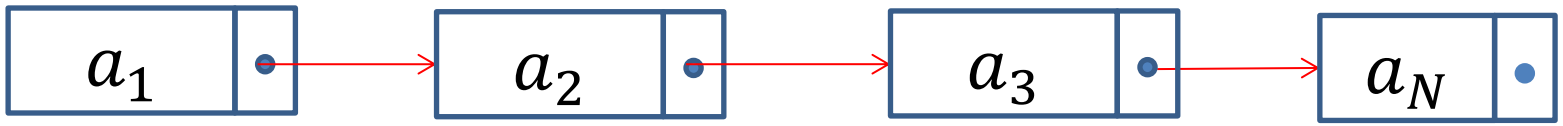
```
void Borrar(ElementoLista **lista)
{
        struct ElementoLista *tmp;
        struct ElementoLista *primero;

        primero = *lista;

        while(primero!=NULL)
        {
                tmp = primero->resto;
                free(primero);
                primero = tmp;
        }

}
```

2.2.3 Recorrido

El recorrido de una lista se realiza en sentido directo *(hacia adelante)* o, en algunos casos, en sentido inverso *(hacia atrás).*

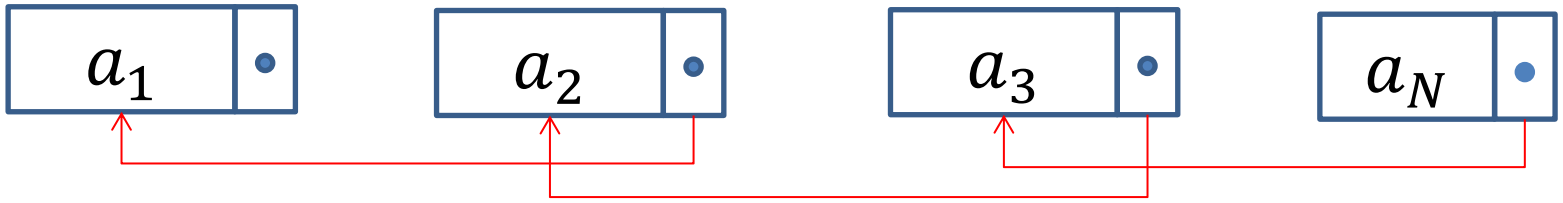
$L = \{a_1, a_2, a_3, \ldots\ldots, a_N\}$

hacia adelante



$L = \{a_N, \ldots, a_3, a_2, a_1\}$

hacia atrás



2.2.3 Recorrido

```
void Recorrido(ElementoLista **lista)
{
        while((*lista)->resto!=NULL)
        {
                printf("%d",(*lista)->elemento);
                printf("\n");
                *lista = (*lista)->resto;
        }
}
```

2.2.4 Búsqueda

El nodo o elemento se especifica por su posición en la lista; para ello se considera la pocisión 1, la correspondiente al nodo de cabeza *(header);* posición 2, la correspondiente al siguiente nodo, así sucesivamente.

El algoritmo de búsqueda del elemento comienza con el recorrido de la lista mediante un apuntador *índice* que comienza apuntando al nodo que encabeza la lista. Un bucle mueve el *índice* hacia adelante el número correcto de sitios (lugares).

2.2.4 Búsqueda

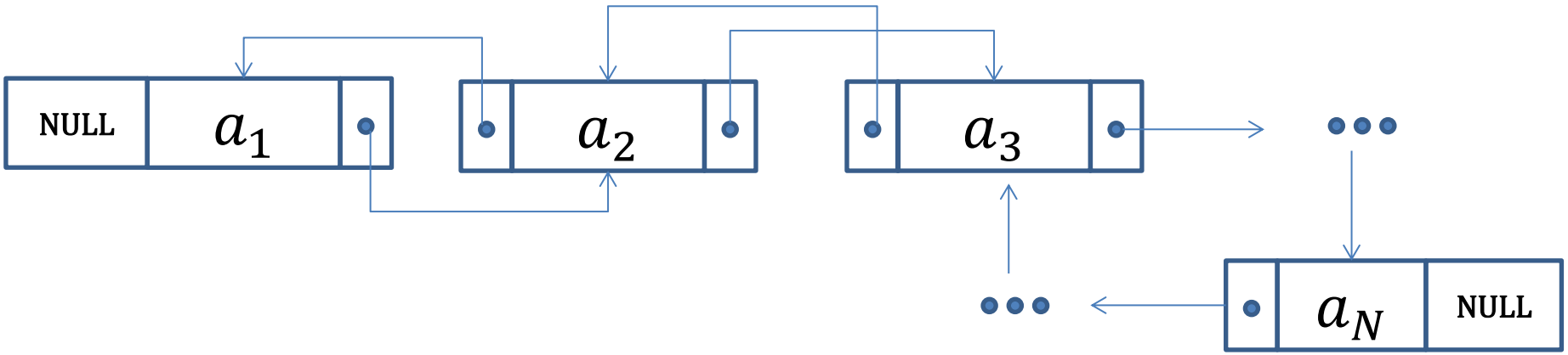
A cada iteración del bucle se mueve el apuntador *índice* un nodo hacia adelante. El bucle termina cuando se alcanza la posición deseada e *índice* apunta al nodo correcto. El bucle también se puede terminar si *índice* apunta a un valor nulo, lo que indicará que la posición solicitada era más grande que el número de nodos de la lista.

2.2.4 Búsqueda

```
ElementoLista * Búsqueda(ElementoLista **lista,ElementoLista **nodo_buscado)
{
        while((*lista)->resto!=NULL)
        {
                if((*lista)->elemento == (*nodo_buscado)->elemento)
                        return *lista;
                else
                        *lista = (*lista)->resto;
        }

        return NULL;
}
```

2.3 Listas doblemente ligadas

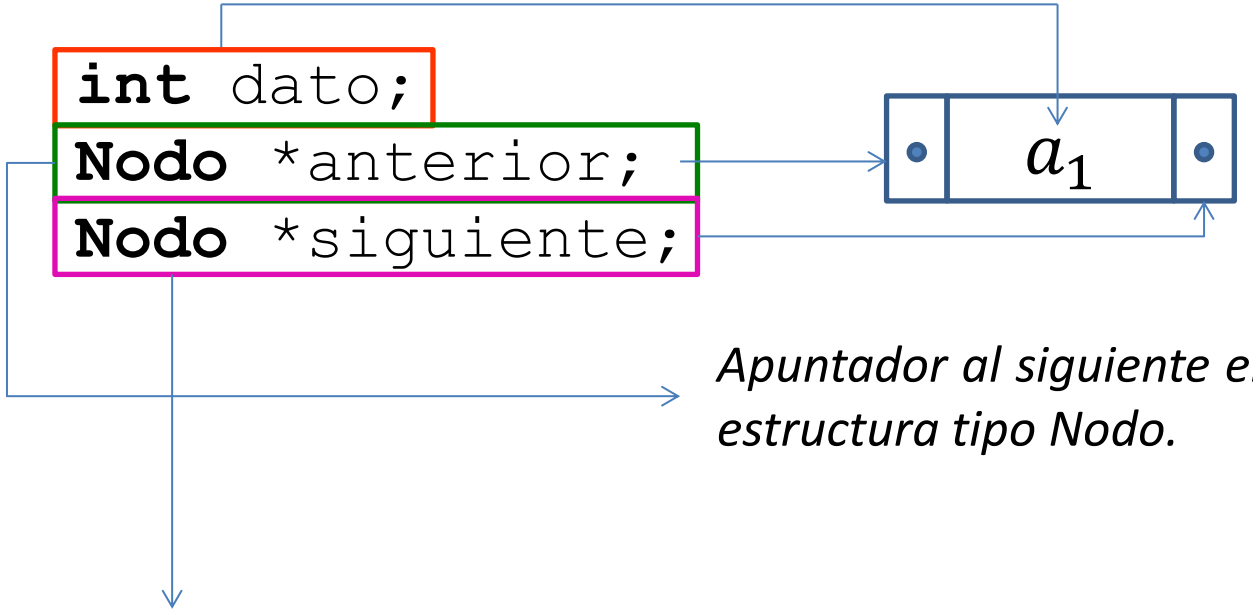
Existen numerosas aplicaciones en las que es conveniente poder acceder a los elementos o nodos de una lista en cualquier orden, esto es, hacia adelante o hacia atrás. En este caso se recomienda el uso de una **lista doblemente enlazada**. En tal lista, cada elemento contiene dos apuntadores, aparte del valor almacenado en el elemento. Un apuntador apunta al siguiente elemento de la lista y el otro apuntador al elemento anterior.



2.3 Listas doblemente ligadas

El elemento puede ser de cualquier tipo como float, char, double, double float, struct, etc.

```
struct Nodo
{
    int dato;
    Nodo *anterior;
    Nodo *siguiente;
}
```



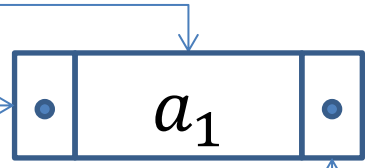
Apuntador al siguiente elemento estructura tipo Nodo.

Apuntador al anterior elemento estructura tipo Nodo.

2.3 Listas doblemente ligadas

El elemento puede ser de cualquier tipo como float, char, double, double float, class, etc.

```
class Nodo {

public:
      int dato;
      Nodo *anterior;
      Nodo *siguiente;

      //constructor
}
```

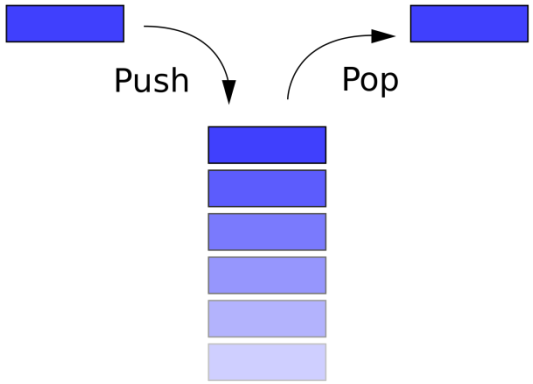


Guarda la dirección del anterior objeto de la lista.

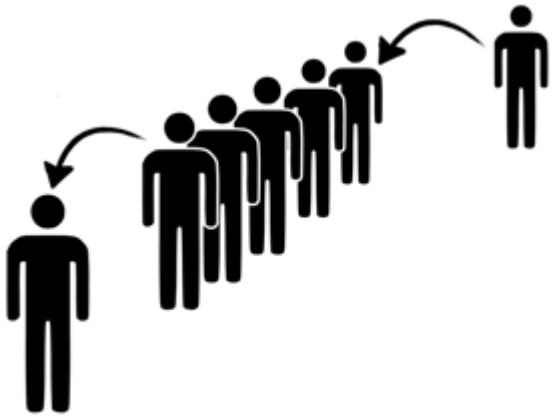
Apuntador al siguiente elemento tipo Nodo.

2.4 Aplicaciones

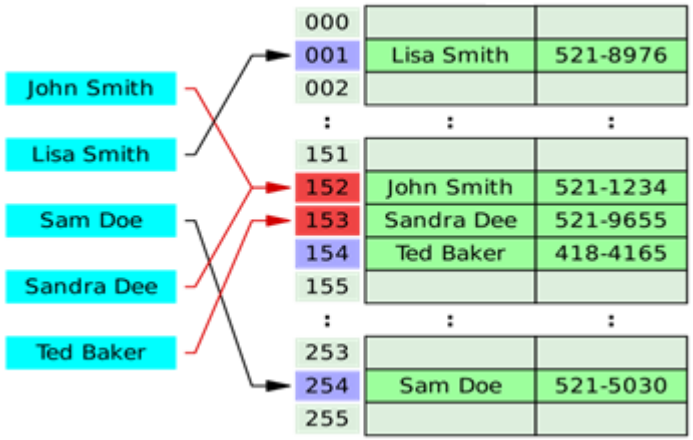
Pilas



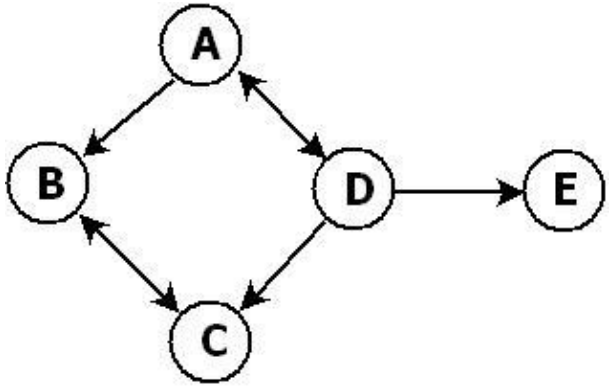
Colas



Tablas Hash



Grafos

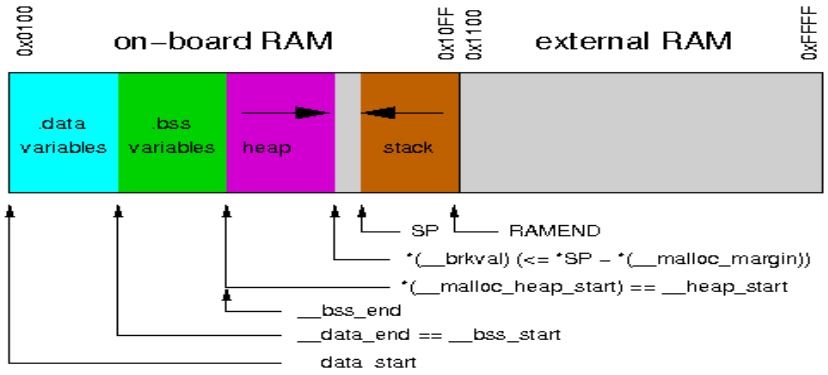


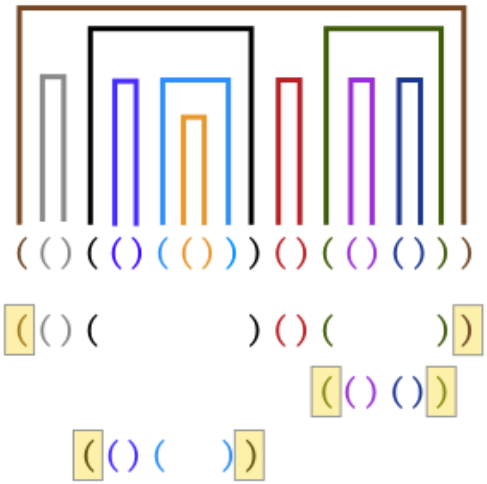
2.4 Aplicaciones

Representación de Matrices

$$A = \begin{bmatrix} a_0 & a_{-1} & a_{-2} & \cdots & \cdots & a_{-n+1} \\ a_1 & a_0 & a_{-1} & \ddots & & \vdots \\ a_2 & a_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{-1} & a_{-2} \\ \vdots & & \ddots & a_1 & a_0 & a_{-1} \\ a_{m-1} & \cdots & \cdots & a_2 & a_1 & a_0 \end{bmatrix}$$

Manejo dinámico de la memoria





Balance de paréntesis

Representación de polinomios

$$-3x^2y + 4x^3y^2 - 3xy$$

Guión explicativo

- Este juego de diapositivas debe leerse en el orden que aparece.
- Anterior a este juego, se debió haber revisado el manejo de variables dinámicas.
- Se puede leer en orden indistinto las secciones de Pilas, Colas así como esta presentación.
- Posterior a este juego de diapositivas se debe revisar el material de la Unidad 3 – aplicar la estructura de datos de árbol

Resumen

- Una lista lineal es una lista en la que cada elemento tiene un único sucesor.
- Existen cuatro operaciones típicas asociadas con listas lineales: inserción, eliminación, búsqueda y recorrido.
- Una **lista enlazada** es una colección ordenada de datos en los que cada elemento contiene la posición (dirección) del siguiente elemento. Es decir, cada elemento (nodo) de la lista contiene dos partes: datos y enlace(apuntador)

Resumen

- Una **lista simplemente enlazada** contiene sólo un enlace a un sucesor único a menos que sea el último, en cuyo caso no se enlaza con ningún otro modo.
- Cuando se desea insertar un elemento en una lista enlazada, se deben considerar cuatro casos: añadir a lista vacía, añadir al principio, añadir en el interior y añadir al final.
- Si se desea borrar un nodo de una lista se deben considerar dos casos: borrar el primer nodo y borrar cualquier otro nodo

Resumen

- El recorrido de una lista enlazada significa ir por la lista (visitar) nodo a nodo y procesar cada una.
- Una **lista doblemente enlazada** es una lista en la que cada nodo tiene un apuntador a su sucesor y otro de predecesor.
- Una **lista enlazada circularmente** es una lista en la que el enlace del último nodo apunta al primero de la lista.

Ejercicios Propuestos

1. Escribir una función que calcule el número de nodos de una lista enlazada.
2. Escribir una función que elimine de una lista *L* el *n-ésimo* elemento.
3. Evaluar la longitud de una lista enlazada.
4. Escribir el código que muestra si una lista enlazada está vacía.
5. Escribir el segmento de código que cree la lista enlazada con los datos del 1 al 20.
6. Escribir una función que cuente el número de veces que una determinada clave se repite en una lista secuencial

Ejercicios Propuestos

7. Escribir un algoritmo que añada dos listas enlazadas juntas.
8. Escribir un algoritmo que lea una lista de enteros del teclado, que construya una lista enlazada con ellos e imprima el resultado.
9. Escribir un algoritmo que acepte una lista enlazada, la recorra y devuelva el dato del nodo con el valor menor.

Ejercicios Propuestos

8. Escribir un programa que intercambie dos nodos de una lista enlazada. Los nodos se identifican por número y se pasan como parámetros. Por ejemplo, para intercambiar los nodos 5 y 8, se debe llamar a la función **Intercambiar(5,8)**. Si el intercambio se realiza con éxito, se devuelve un valor verdadero; si se encuentra un error, tal como el número de un nodo inválido, se devuelve falso.

Bibliografía Básica

- Aguilar, Joyanes Luis, Programación en C++, Algoritmos, estructuras de datos y objetos, Ed. McGraw Hill,2002.
- Weiss. Allen Mark, Estructuras de datos y Algoritmos, Addison-Wesley Iberoamericana, S.A., 1995.
- A.,Euán Jorge I., B. Cordero Luis G., Estructura de datos, UNAM, 1984.
- Cairó, Osvaldo y Guardati, Silvia. Estructura de datos. Ed. McGraw Hill,2006.
- Criado, Ma. Asunción. Programación de lenguajes estructurados. AlfaOmega Ra-Ma, 2006.
- López, Leobardo. Programación estructurada. Un enfoque algorítmico. AlfaOmega,2004.

Bibliografía Complementaria

- Drozdeck,Adam. Estrcuturas de datos y algortimos en Java. Ed. Thomson, 2007.
- Joyanes, Luis M. Fernández, L. Sánchez, I. Zahonero. Estructuras de datos en C. Ed. McGraw Hill,2005.
- Koffman, Elliot y wolfang, Paul. Estrcutura de datos con C++. Objetos, abstracciones y diseño. Ed. McGraw Hill,2008.
- Nyhoff, Larry. TADs, Estructuras de datos y resolución de problemas en C++. Ed. Pearson-Prentice Hall, 2006.