Estructuras de Datos y Algoritmos

Código de Curso: CY330
Versión 2.2

Introducción al Curso

CY330 Estructura de Datos y Algoritmos

Instructor:

Por favor preséntese a los alumnos

- Nombre
- Experiencia en estructura de datos y algoritmos
- Metas que espera alcanzar

Descripción del Curso

- Este curso está diseñado para introducir a los estudiantes a los conceptos de estructuras de datos y algoritmos.

- El alcance de este curso es proporcionar los conceptos básicos relacionados a estructura de datos y algoritmos. Así como, la implementación de varias operaciones sobre las estructuras de datos.

Audiencia

- Estudiantes, profesionales, y desarrolladores que desean saber acerca de estructura de datos y algoritmos.

Prerrequisitos

- Conocimiento básicos de conceptos de programación.
- Conocimiento de las características generales de los lenguajes de programación.
- Resolución de problemas usando lenguajes de programación C.
- Experiencia práctica en el uso de características avanzadas de C, cómo punteros y archivos.
- Estos conocimientos se pueden obtener completando el curso **CY320 - Introducción a la programación con C.**

Objetivos del Curso

- Definir tipos de datos y estructuras de datos
- Discutir la necesidad de listas enlazadas
- Explicar la diferencia entre la implementación de una lista a través de un arreglo y una lista enlazada
- Explicar la estructura de datos de pila
- Listar las aplicaciones de pilas

Objetivos del Curso

- Definir el tipo de dato abstracto cola e implementar una cola usando un arreglo
- Definir grafos y sus aplicaciones
- Definir árbol como una estructura de datos y discutir acerca de árboles binarios.
- Explicar las diversas técnicas de ordenamiento
- Listar las diferentes técnicas de búsqueda, y describir las tablas hash como una posible estructura de datos

Agenda

Cada Unidad de este curso es de dos horas de duración

Resumen

- Hemos visto los objetivos generales del curso y la agenda
- Empecemos ahora

Volumen 1:

Estructuras de Datos Simples

Unidad 1:

Fundamentos de Estructura de Datos y la Estructura de Datos Lista

Objetivos del Aprendizaje

- Explicar el rol de las estructuras de datos y algoritmos como bloques de construcción en programas de computadora
- Definir estructuras de datos
- Discutir las diferencias entre un tipo de dato y una estructura de dato
- Explicar el rol de las estructuras de datos en la resolución de problemas
- Describir el Tipo de Dato Abstracto lista
- Discutir la estructura de datos lista
- Explicar como implementar una lista como un arreglo

La Necesidad de Algoritmos y Estructuras de Datos

Dos elementos esenciales en la resolución de problemas:

- **Estructura de datos**:

  Mecanismo de almacenamiento usado para almacenar datos.

- **Algoritmos:**

  Método usado para resolver el problema

- Los algoritmos permiten que algunas de las operaciones básicas e importantes se realicen sobre las estructuras de datos.

Tipos de Datos

- Un tipo de dato es un conjunto de valores que una variable de ese tipo puede tomar, y las operaciones permitidas en ella.
- Las diferentes tipos de datos que un lenguaje de programación soporta son:
  - integer
  - float
  - char
  - boolean

Estructuras de Datos

- El término *estructura de dato* se usa normalmente para referirse a una colección de datos que están organizados de alguna forma u otra.
- Una estructura de datos es una colección de entidades pertenecientes a tipos de datos diferentes que resultan de la organización de piezas de datos interrelacionadas.
- Las estructuras de datos también tienen un conjunto de valores y operaciones definidas en estos valores.
- Las estructuras de datos son declaradas y definidas por los programadores.

Rol de las estructuras de datos para resolver problemas

- La selección de las estructuras de datos que pueden usarse para resolver un problema es uno de las principales fuentes de variabilidad en el diseño de algoritmos.
- La selección de una estructura de dato determina la facilidad con la cual un algoritmo pueda ser escrito.
- La selección de una estructura de dato puede incrementar o disminuir el requerimiento de espacio de almacenamiento y otros recursos en una computadora.

Rol de las estructuras de datos para resolver problemas

- La selección de una estructura de dato puede incrementar o disminuir el tiempo de ejecución de un algoritmo en una computadora.
- Las estructuras de datos y los algoritmos constituyen los principales bloques de construcción de los programas.

TDA

- TDA - Tipo de Dato Abstracto se refiere a entidades de almacenamiento de datos y las operaciones definidas en esas entidades.
- TDA es simplemente una colección de operaciones.
- Lista vista como un TDA
  - Ø Insertar un elemento
  - Ø Borrar un elemento
  - Ø Encontrar un elemento
  - Ø Mostrar elementos

Estructura de Datos Lista

- Una lista se puede definir como una serie de cero o más elementos, donde cada elemento pertenece a un tipo dado.

  $elem_1, elem_2, elem_3 \ldots elem_n$

- n se denomina la longitud de la lista
- $elem_1$, es el primer elemento de la lista
- $elem_n$ es el último elemento de la lista
- Una lista se dice que está vacía cuando el número de elementos, n, es cero.
- Las listas son estructuras de datos flexibles que se puede expandir o contraer

Lista como un TDA

- Una lista puede ser vista como un TDA al definir un conjunto de operaciones sobre ésta
  - nuevo(lista)
  - insertar(lista, elem)
  - borrar(lista, pos)
  - encontrar(lista, elem)
  - obtener(lista, pos)
  - esvacio(lista)

Lista como un TDA...Continuación

- Las operaciones nuevo (new) e insertar (insert) son dos operaciones básicas de la lista que generan todas las nuevas listas posibles.
- Estas operaciones se denominan *operaciones primitivas*
- Una operación primitiva es aquella que puede usarse para definir todas las otras operaciones de una estructura de datos.
- Todas las otras operaciones sobre la estructura de datos se pueden definir usando las operaciones primitivas

Definición de `esvacio` usando `nuevo` e `insertar`

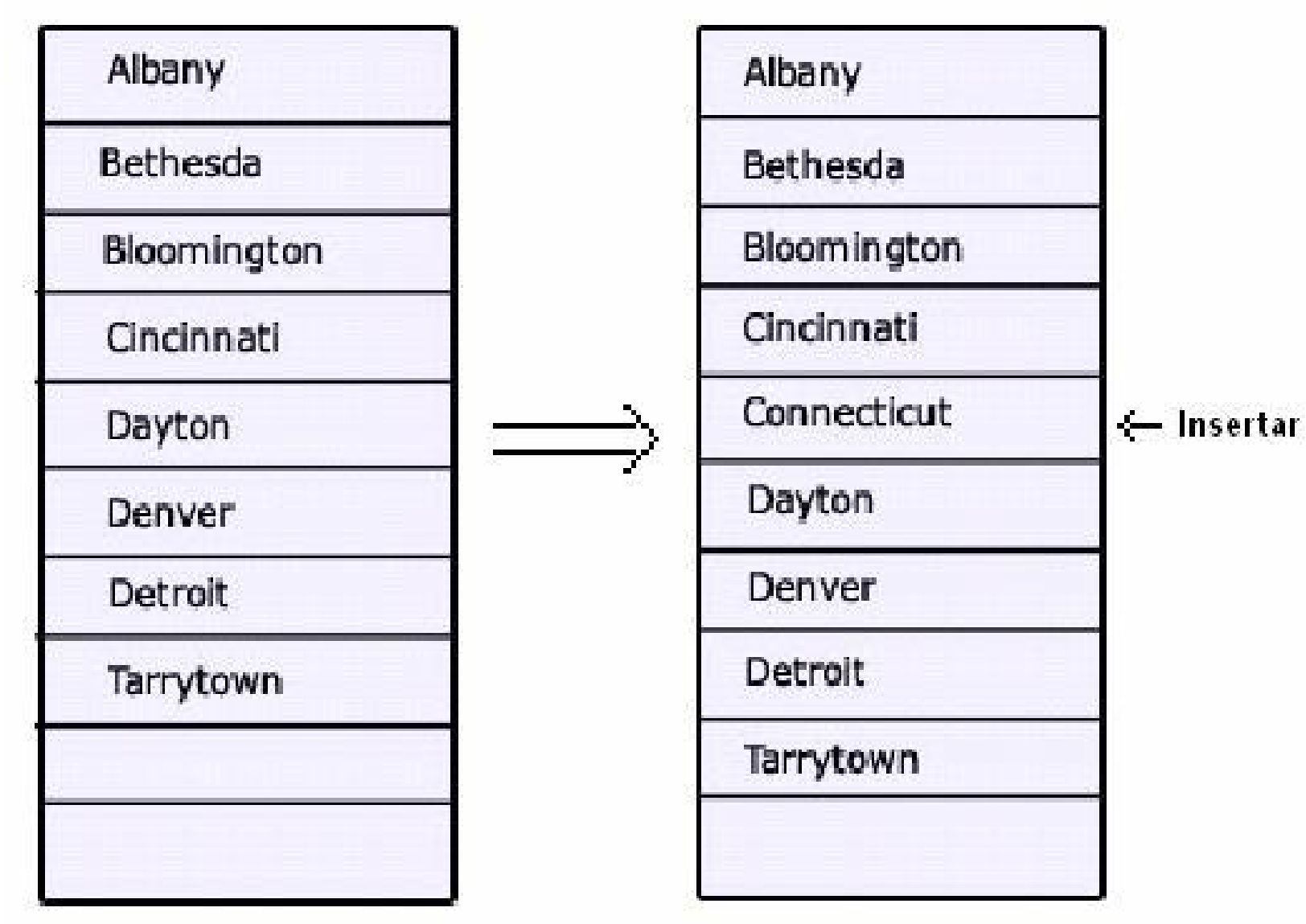
```
esvacio(nuevo) = verdadero
```

- La nueva lista retornada por la operación nuevo es una lista vacía, ya que ningún elemento se ha añadido hasta ahora.

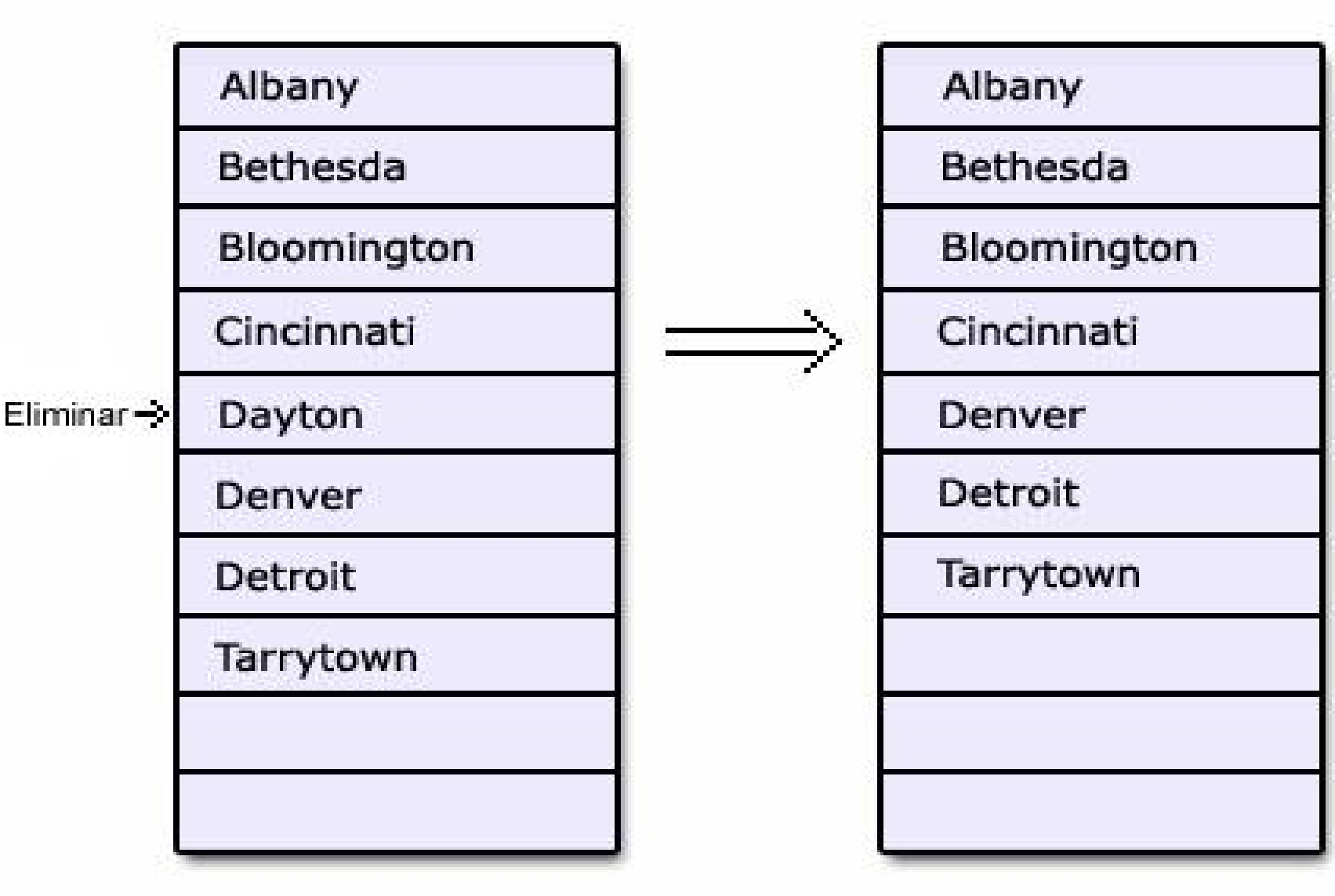
```
esvacio(insertar(lista,elem))=falso
```

- `lista` pasada a `esvacio` después de la adición de `elem` no es una lista vacía
- Las reglas que definen a `esvacio` usando operaciones primitivas se denominan axiomas.

Insertar en una Lista



Eliminar en una Lista



Operaciones sobre una Lista

- Usando un arreglo se implementan las operaciones:
  - Ø Crear una nueva lista
  - Ø Insertar un elemento en una lista
  - Ø Eliminar un elemento de una lista
  - Ø Buscar un elemento en una lista
  - Ø Encontrar el sucesor y predecesor de un elemento en una lista
  - Ø Determinar si una lista está vacía

Implementar una Lista como un Arreglo

```
#define LISTMAX 100 /* Limite máximo */
#define STRINGSIZE 30

/* Definición del Type para una lista */

typedef struct {
 int count; /* Nro de elementos en list */
 char list[LISTMAX][STRINGSIZE];
} List_Array;
```

Crear una nueva Lista

```
/* Crear una lista. */

void newList(List_Array *list_ptr){
  int k;
  list_ptr->count = 0;/* No hay elementos*/
  for(k = 0; k < LISTMAX; k++)

/* Inicializar con un carácter null */
     strcpy(list_ptr->list[k],"\0");
}
```

Insertar un elemento en una Lista

```
/* Para insertar un elemento en la lista */

int insertElement(List_Array *list_ptr, \
char *element) {
 if(list_ptr->count == LISTMAX)
  return (-1); /*Tamaño máximo alcanzado*/
  else {
   strcpy(list_ptr->list[list_ptr->count], \
          element);
   list_ptr->count++; /*Un elemento más*/
   return(1);         /* retorna éxito */
  }
}
```

Eliminar un elemento en una Lista

```
/* Eliminar un elemento en la lista dada  su
  posición en la lista*/

int deleteElement(List_Array *list_ptr, \
int pos) {
 int k;

/* Revisar si pos existe y errores */

 if (pos < 0 || pos > list_ptr->count-1)
     return(-1);    /* error */
```

Eliminar un elemento en una Lista

```
else {

/* Mueve todos los elementos hacia arriba */

      for (k = pos; k < list_ptr->count - 1;
                        k++)
      strcpy(list_ptr->list[k],
                  list_ptr-list[k+1]);
      list_ptr->count--;
      return(1); /* eliminación con éxito */
   }
}
```

Buscar un elemento en una Lista

```
/* Encontrar un elemento en la lista */

int find(List_Array *list_ptr, char *element){
   int k = 0;
   while (k <= list_ptr->count - 1)

/* Revisar por el nombre en k */

   if (!strcmp(list_ptr->list[k], element))
     return(k+1);
   else
     k++;
  return(-1); /* Elemento no encontrado */
}
```

Sucesor de un elemento en una Lista

```
/* Encuentra el elemento sucesor dada una
   posición en la lista */

char * succ(List_Array *list_ptr, int pos){

  /* Revisar errores */

 if (pos < 0 || pos > list_ptr->count-1) {
  printf("\nSuccessor - Input error: No\
  tal posición en la lista\n"); /* Error */
  return(NULL);
  }
```

Sucesor de un elemento en una Lista

```
else
  if (pos == list_ptr->count-1) {
   printf("No existe sucesor para el \
         último elemento en la lista!\n");
   return(NULL);
     }
     else
      return(list_ptr->list[pos + 1]);
}
```

Predecesor de un elemento en una Lista

```
/* Encuentra el elemento predecesor dada
una posición en la lista */

char * pred(List_Array *list_ptr, int pos){

  /* Revisar errores */
if (pos < 0 || pos > list_ptr->count-1){
printf("\nPredecessor - Input error: No\
existe esa posición en la lista\n");//Error
return(NULL);
}
```

Predecesor de un elemento en una Lista

```
else
 if (pos == 0) {
 printf("No existe predecesor para el \
        primer elemento en la lista!\n");
 return(NULL);
    }
     else
       return(list_ptr->list[pos - 1]);
}
```

Determinar si la Lista esta vacia

```
/* Determinar si la lista está vacía o no */

    int isEmpty(List_Array *list_ptr) {
  if (list_ptr->count == 0)
    return(1); /* lista está vacía */
   else
    return(0); /* lista no está vacía */
}
```

Aplicaciones usando Lista

- Representación y operaciones aritméticas en enteros súper largos
- Representación y operaciones sobre polinomios
- Diversidad de listas, por ejemplo, listas de votantes, lista de elementos de compra, etc.
- Planificación de tareas
- Simulación de colas usando listas
- Bioinformática

Aplicaciones usando Lista

- Aplicaciones de procesamiento de texto
- Implementaciones de conjuntos
- Aplicaciones en sistemas operativos para administrar listas de procesos y administración de paginación de memoria

Resumen

- Explicar el rol de las estructuras de datos y los algoritmos como bloques de construcción de los programas de computadoras
- Definir estructuras de datos
- Discutir las diferencias entre un tipo de dato y una estructura de datos
- Explicar el rol de las estructura de datos en la solución problemas
- Describir el Tipo de Dato Abstracto lista
- Discutir la estructura de datos lista
- Explicar cómo implementar una lista como un arreglo

Unidad 2:

Lista Enlazada

Objetivos del Aprendizaje

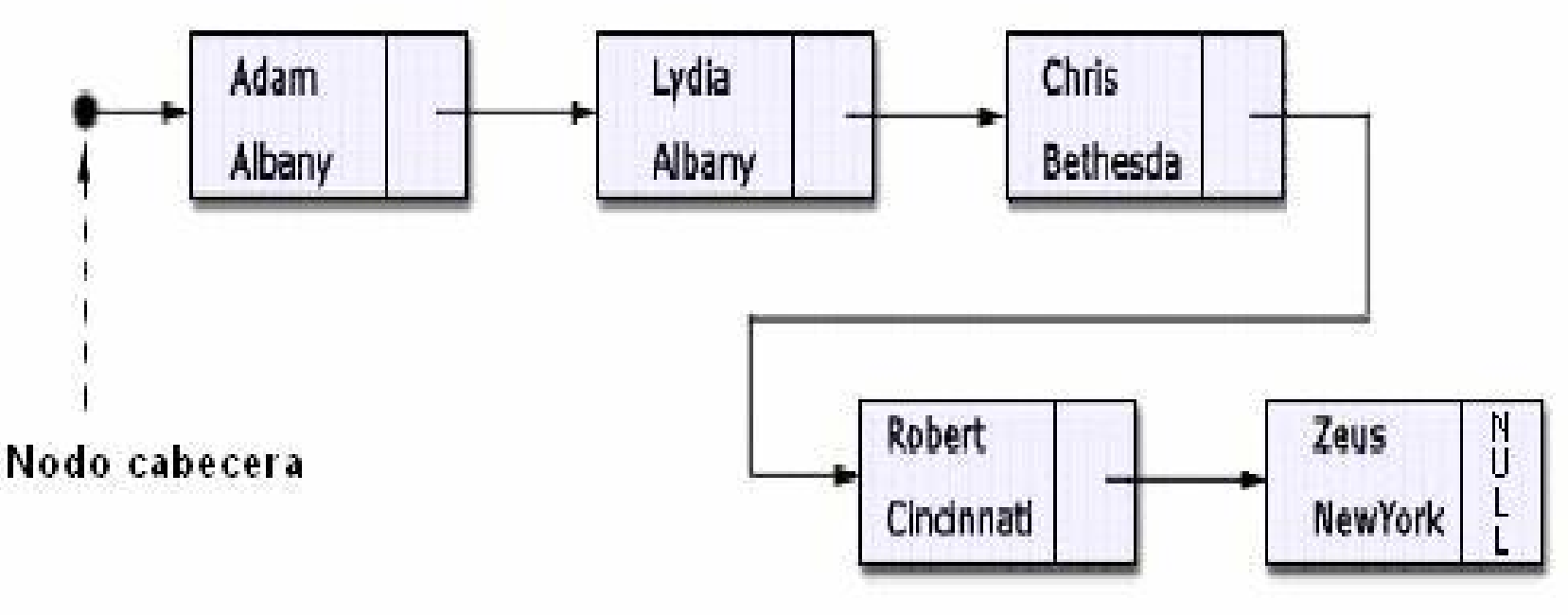
- Explicar la necesidad de las listas enlazadas
- Describir las diferentes operaciones sobre una lista enlazada
- Describir la implementación de listas enlazadas simples
- Definir listas doblemente enlazadas y listas circulares
- Comparar la implementación de una lista con arreglos y con lista enlazada

Introducción a Listas Enlazadas

- Una lista enlazada es una colección lineal de elementos de datos llamados *nodos*.
- Cada nodo consiste de dos partes:
  - Ø La parte de la Información
  - Ø La parte de Enlace, el cual contiene un puntero al siguiente nodo.

Introducción a Listas Enlazadas

- La implementación de la lista enlazada permite expandir y contraer la lista en tiempo de ejecución.
- Una lista enlazada en la cual cada nodo tiene un indicador al siguiente en la lista se denomina *lista enlazada simple*



Algunos ejemplos de Nodos en C

Ejemplo 1

```
typedef struct node {
    int data;
    struct node *next;
} Node_type;
```

Ejemplo 2

```
typedef struct node {
    int studentid;
    char studentname;
    struct node *next;
} Node_type;
```

Nodo cabecera en una Lista Enlazada

- En una lista enlazada, el último nodo se denota por un valor `null` en su campo `next`, por lo que es fácil determinar el final de la lista.
- Un puntero externo a la lista, referido como *cabecera* (`header`) , se usa para determinar el inicio de la lista
- Inicializar la cabecera con NULL, indica una lista vacía

```
header = NULL;
```

Nodo cabecera en una Lista Enlazada

- El valor en la variable `cabecera` (`header`) cambia en las siguientes instancias

Cuando el primer nodo se añade a la lista	La ubicación de memoria del nuevo nodo se asigna a la cabecera
Cuando un nuevo nodo va a ser insertado al inicio de la lista	La ubicación de memoria de la cabecera se asigna al nuevo nodo
Cuando el primer nodo en la lista es eliminado	Se le asigna a la cabecera el valor del nodo, siguiente al primer nodo
Cuando la lista contiene un solo nodo y el nodo es eliminado	Se le asigna a la cabecera `NULL`

Operaciones – Definición de Nodo

```
typedef int Element_type;
typedef struct LinkedList_node_tag {
    Element_type value;
    struct LinkedList_node_tag *next;
} Node_type;
```

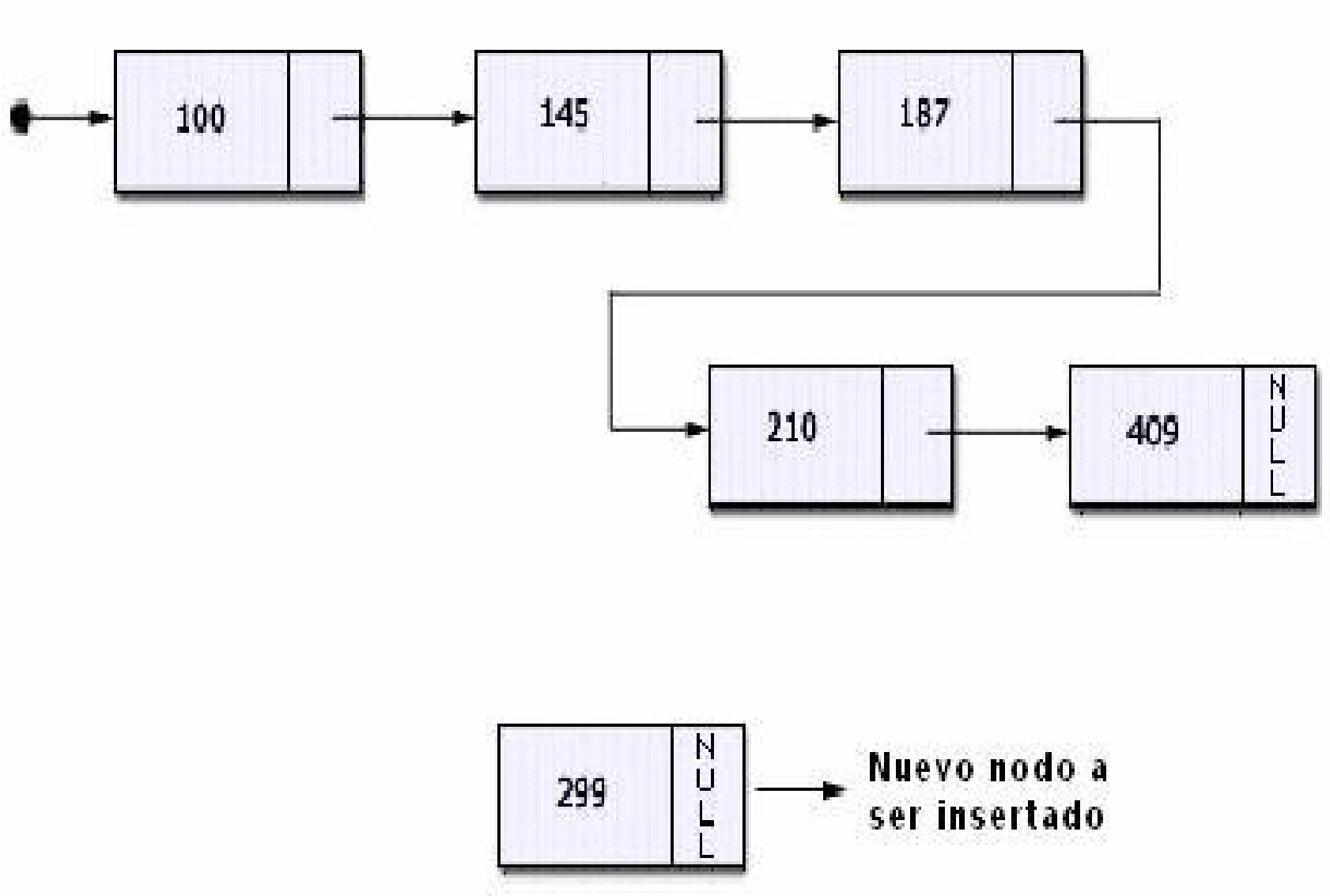
Crear un nuevo Nodo

```
/* Obtener un nodo dinámicamente */

Node_type *getNode(Element_type elem) {
   Node_type *node;
   node = (Node_type *) \
    malloc(sizeof(Node_type));
   if (node != NULL) {
     node->value = elem;
     node->next = NULL;
   }
   return node;
}
```

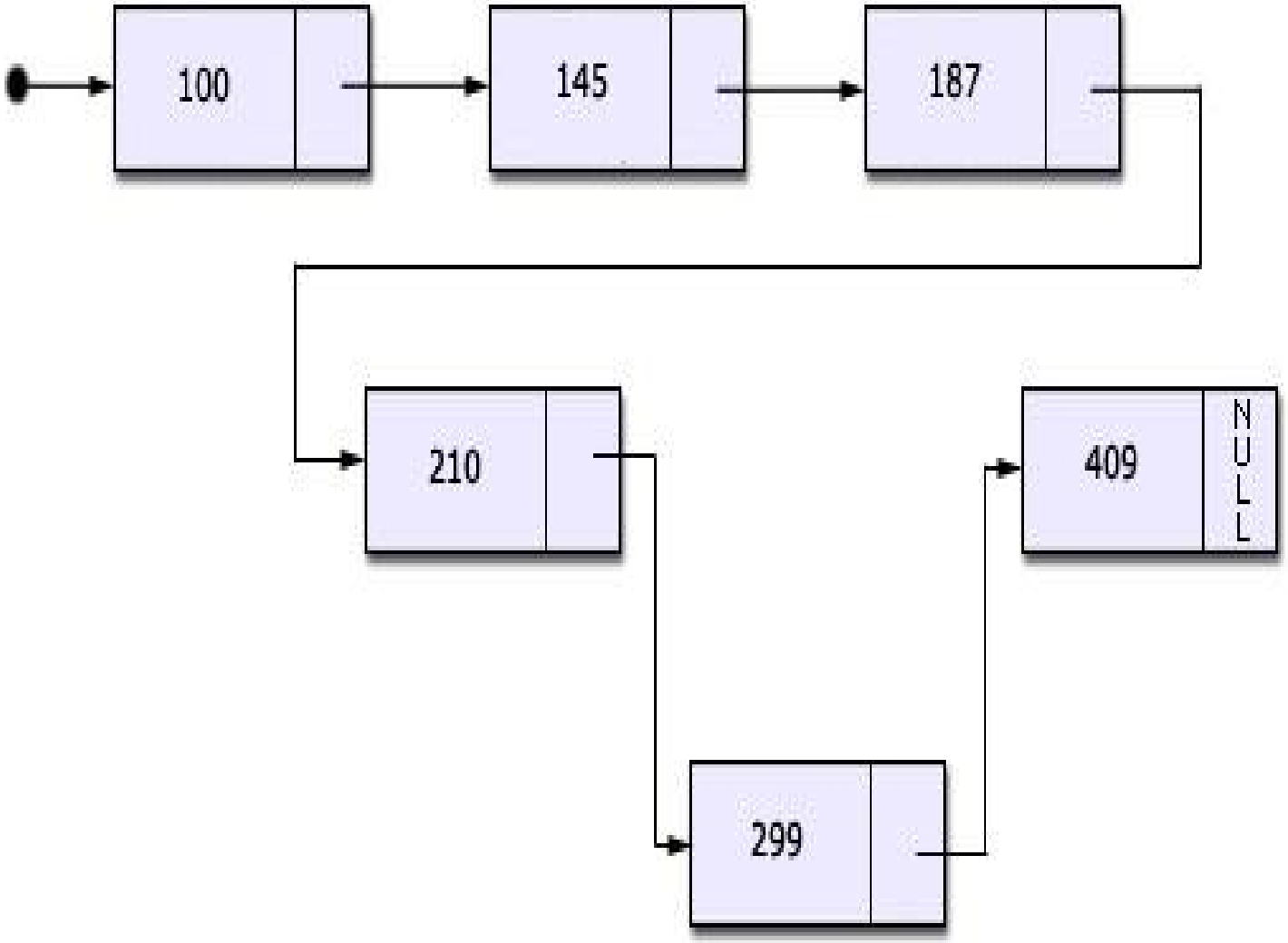
Recorrer una Lista Enlazada

```
/* Recorrer una lista dada la cabecera de la
    lista */

void traverseList(Node_type *head) {
    Node_type *ptr;
    for (ptr = head; ptr!= NULL; \
         ptr = ptr->next)
      printf("%d  ",ptr->value);
    printf("\n");
}
```

Insertar un nodo después de un nodo dado



Insertar un nodo después de un nodo dado



Insertar un nodo después de un nodo dado

```
/* Insertar un nodo después de un nodo dado*/

Node_type* insertAfter(Node_type *node,\
                       Element_type elem) {
    Node_type *ptr;
    if (node == NULL)
      return NULL;   /*No puede insertar*/
    ptr = getNode(elem);
    if (ptr != NULL) {
      ptr->next = node->next;
      node->next = ptr;
    }
    return ptr; //Retorna el nuevo nodo creado
}
```

Insertar un nodo antes de un nodo dado

```
/*Insertar un nodo antes de un nodo dado */

Node_type* insertBefore(Node_type **head,\
    Node_type *node, Element_type elem) {

   Node_type *ptr, *q;

   if (node == NULL)
      return NULL;
   if (head == NULL)
      return NULL;
   if (*head == NULL)
      return NULL;
   ptr = getNode(elem);
```

Insertar un nodo antes de un nodo dado

```
if (ptr == NULL)
     return NULL;
if (*head == node) {
     ptr->next = *head;
     *head = ptr;
    }
    else {
       for (q= *head; q->next != node;
           q = q->next);
           // Sólo recorrido de la lista
     ptr->next = q->next;
     q->next = ptr;
    }
    return ptr; //Retorna el nuevo nodo creado
}
```

Eliminar un nodo

```
/* Eliminar el nodo actual */

Node_type* deleteNode(Node_type *head,\
    Node_type *node) {
    Node_type *ptr;
    if (head == NULL || node == NULL)
      return NULL;
    if (head == node) {
      head = node->next; // Primer nodo
      free(node);
      return head;
    }
```

Eliminar un nodo

```
/* Moverse en la lista */

 for (ptr = head; ptr->next != node; \
      ptr = ptr->next)
     ptr->next = node->next;
 free(node);
 return ptr; /*Retorna el nodo previo al
                nodo de entrada*/
}
```

Operación Predecesor de un nodo

```
/* Encontrar el predecesor de un nodo en
 una lista */

Node_type *findPred(Node_type *head,\
    Node_type *node) {

    Node_type *ptr;
    /* Revisa por valores NULL de entrada */
    if (head == NULL) {   /* Error */
    printf("\nPredecesor- Error de entrada:\
            Head es NULL\n");
      return(NULL);
    }
```

Operación Predecesor de un nodo

```
if (node == NULL) {    /* Error */
    printf("\n Predecesor- Error de \
            entrada: Node es NULL\n");
    return(NULL);
   }
   else
    if (node == head) {
    /* Si es primer nodo en lista */
     printf("\n No existe para el primer \
             nodo en la lista!\n");
     return(NULL);
       }
```

Operación Predecesor de un nodo

```
else
 for (ptr = head; ptr->next && \
     ptr->next != node; ptr = ptr->next);
  if (ptr->next == NULL) {
    printf("\n Predecesor- Error de \
    entrada: Nodo de entrada no se \
   encuentra en la lista\n");
    return (NULL);
   }
   else
        return ptr;
}
```

Operación Sucesor de un nodo

```
/* Encontrar el sucesor de un nodo en
   una lista */

Node_type *findSucc(Node_type *head,\
    Node_type *node) {
    Node_type *ptr;

  /* Revisa por valores NULL de entrada */
    if (head == NULL) {          /* Error */
      printf("\n Sucesor- Error de entrada: \
                  Head es NULL\n");
      return (NULL);
    }
```

Operación Sucesor de un nodo

```
if (node == NULL) {          /* Error */
    printf("\n Sucesor- Error de entrada:\
            Node es NULL\n");
    return(NULL);
   }
else {
   for (ptr = head; ptr && \
        ptr != node; ptr = ptr->next);
        if (ptr)  /* Nodo válido en la
                     lista */
```

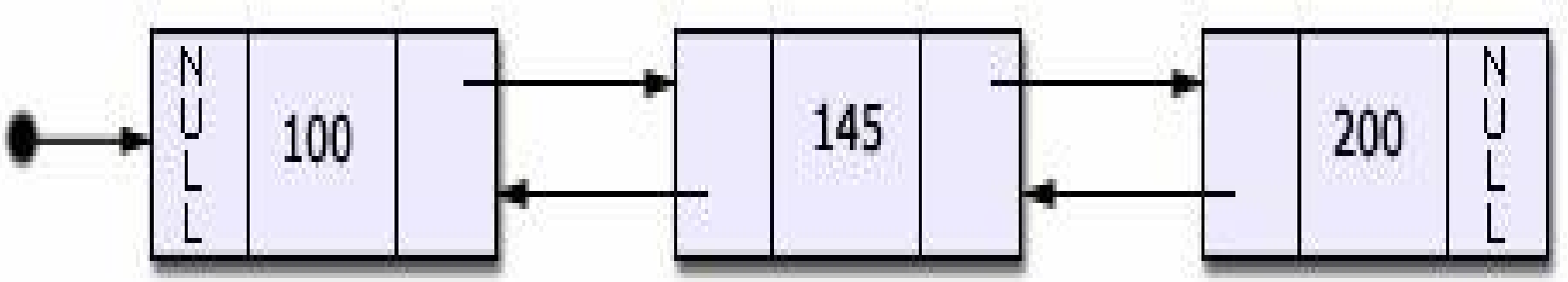
Operación Sucesor de un nodo

```
if (node->next == NULL) {
 printf("\n No existe sucesor para el \
        último nodo en la lista!\n");
 return(NULL);
}
          else
               return node->next;
    else {
  printf("\n Sucesor- Error de entrada: \
        Nodo de entrada no se encuentra \
        en la lista\n");
    return(NULL);
    }
 }
}
```

Lista Doblemente Enlazada

- Una lista doblemente enlazada también es una colección lineal de nodos, donde el nodo consiste de lo siguiente:
  - Un campo de información
  - Un puntero `next`, apuntando al siguiente nodo en la lista
  - Un puntero `prev`, apuntando al nodo anterior en la lista

Lista Doblemente Enlazada

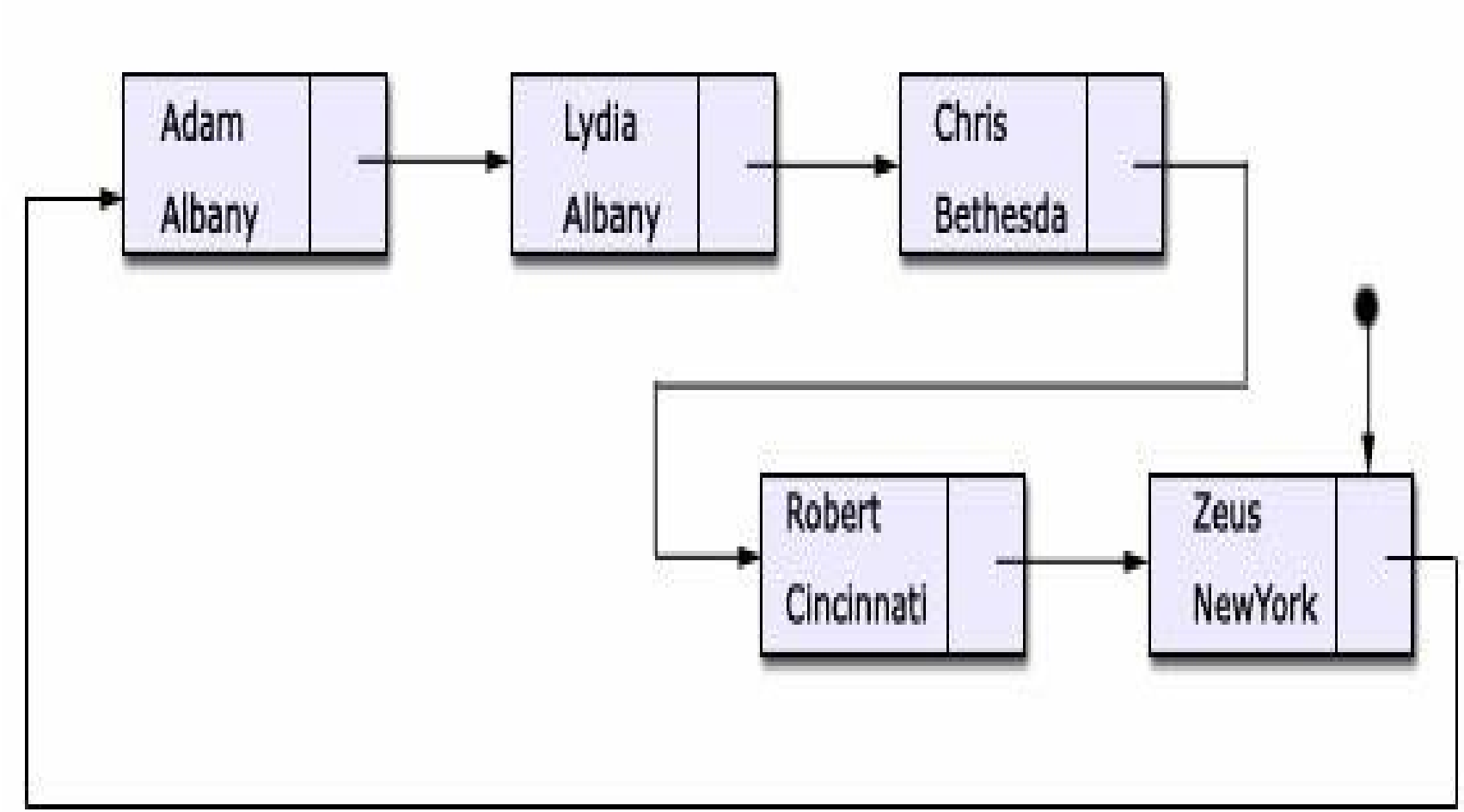



```
typedef struct DLinkedList_node_tag {
    Element_type value;
    struct DLinkedList_node_tag *next;
    struct DLinkedList_node_tag *prev;
} Node_type;
```

Lista Enlazada Circular

- En una lista circular, el último nodo es conectado a la cabeza de la lista
- En una lista circular, no hay cabeza ni último nodo
- El movimiento en una lista puede ser desde cualquier nodo a cualquier otro nodo
- Un puntero externo al último nodo en la lista permite tener un puntero a la cabeza de la lista también, ya que el puntero al siguiente (next) del último nodo apunta al primer nodo de la lista

Lista Enlazada Circular



Lista de Consideraciones

- Número de elementos en una lista (tamaño)
- Acceso arbitrario a cualquier elemento en la lista
- Frecuencia de búsqueda
- Frecuencia de operaciones de insertar y eliminar
- Frecuencia de ordenamiento
- Orden de acceso a elementos
- Tipo de elemento en una lista
- Combinación de dos listas
- Implementación de una lista FIFO
- Implementación de una lista LIFO (pila)
- Técnica eficiente de ordenamiento
- Memoria

Resumen

- Explicar la necesidad de las listas enlazadas
- Describir las diferentes operaciones una lista enlazada
- Discutir la implementación de una lista simplemente enlazada
- Definir listas doblemente enlazadas y listas enlazadas circulares
- Comparar la implementación de una lista con arreglos y con listas enlazadas

Unidad 3:

Laboratorio de Lista Enlazada

Unidad 4:

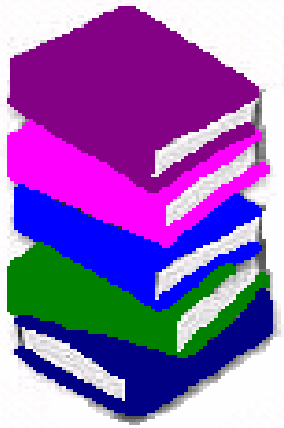
Pila

Objetivos del Aprendizaje

- Definir la estructura de datos de pila
- Explicar las diferentes operaciones definidas en la pila
- Describir las operaciones TDA en la pila
- Explicar la implementación de una pila usando un arreglo y una lista enlazada
- Discutir cómo las pilas se pueden usar para resolver problemas

Introducción a las Pilas

- Una *pila* permite que los elementos sean añadidos o eliminados de un sólo lado
- Las pilas son muy comunes en la vida real



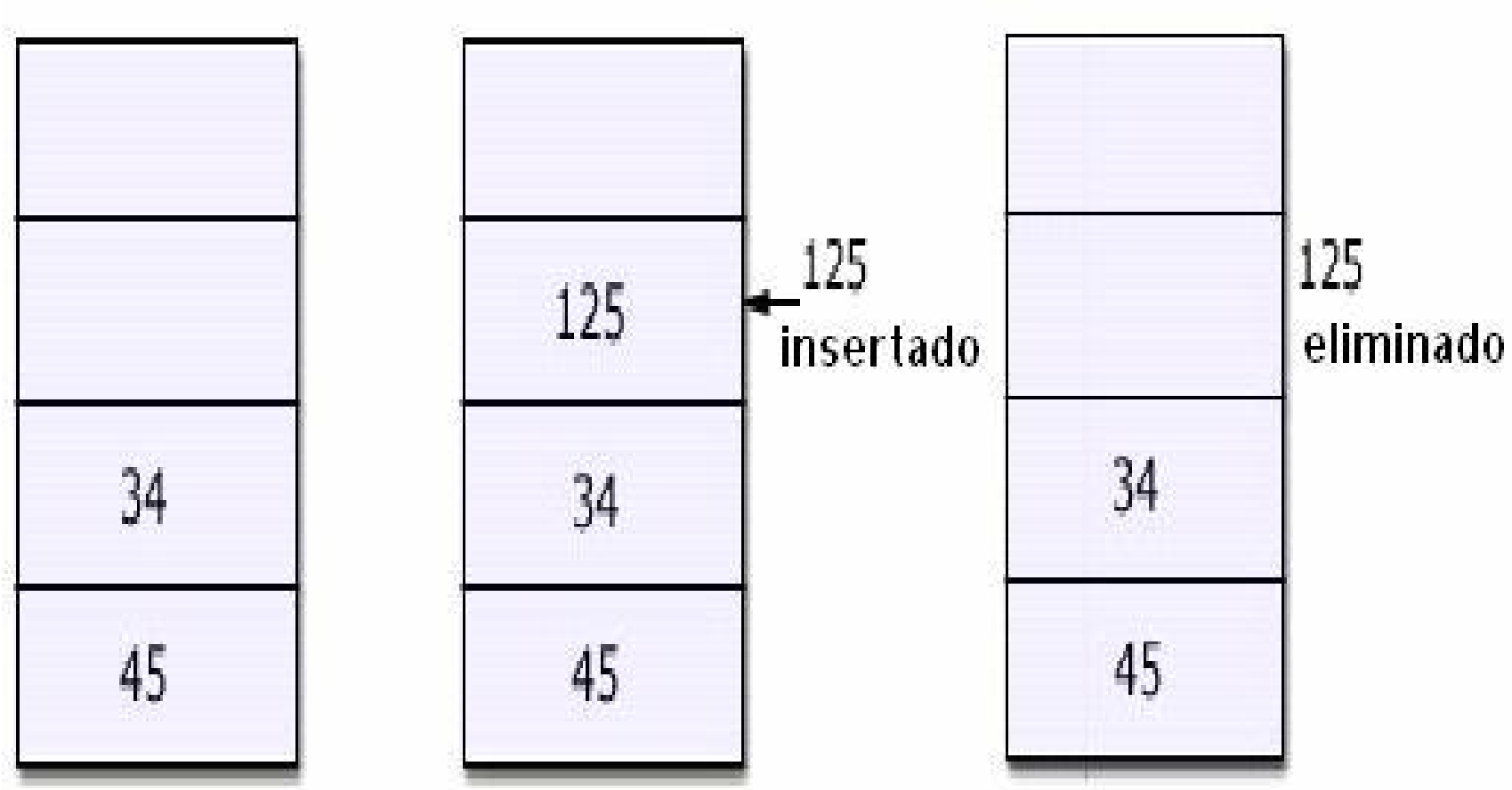
Pila de libros

Pila de periódicos

Introducción a las Pilas

- En una pila, los elementos que son añadidos de último son los primeros en ser removidos
- De aquí que, las pilas también son conocidas como listas Ultimo en Entrar, Primero en Salir (LIFO – Last-IN First-Out)
- El lado final donde los elementos son añadidos o removidos se conoce como ‘tope’ (top) de la pila
- La inserción en una pila se conoce como ‘*push*’
- La eliminación de una pila se conoce como ‘*pop*’

Ejemplo de inserción y eliminación en una Pila



TDA Pila

- `new(S)` – Crea una nueva pila S. La pila S está vacía cuando se crea.
- `push(S,elemento)` – Pone un elemento en el tope de la pila.
- `pop(S)` – Elimina el elemento del tope de la pila.
- `top(S)` – Retorna el elemento en el tope de la pila.
- `isempty(S)` – Retorna verdadero si la pila está vacía, de lo contrario retorna falso.

Definición de pop usando new y push

- Un elemento no puede ser removido de una pila vacía

  `pop(new) = error`

- El último elemento insertado en `S` (sea este `elemento`) es el que se saca

  `pop(push(S,elemento)) = S`

Definición de top usando new y push

- Un pila `new` no contiene elementos y por lo tanto, el elemento `top` no existe o no está definido

  `top(new) = error`

- Colocar `elemento` en la pila `S` significa que `elemento` se convierte en el `top` de la pila

  `top(push(S,elemento)) = elemento`

Definición de isempty usando new y push

- Una pila `new` está vacía

  `isempty(new) = true`

- Colocar un elemento en una pila S significa que hay al menos un elemento en la lista

  `isempty(push(S,elemento)) = false`

Implementar Pilas usando arreglos

- Las pilas pueden implementarse usando un arreglo
- El número de elementos en una pila se define como teniendo un máximo prefijado, llamado STACKMAX
- La pila se define para contener elementos de un tipo particular

```
#define STACKMAX 100
typedef int Element_type;
// Definición del Pila
typedef struct stacker {
 int top;
 Element_type stack[STACKMAX];
} Stack_type;
```

Implementación de newStakc

- Una función *newStack* ayuda a inicializar la pila para su uso
- La función *newStack* establece el valor de *top* a 0

```
/* New inicializa la pila al estado vacío */

  void newStack(Stack_type *st_ptr) {
     st_ptr->top = 0;
  }
```

Implementación de push

```
/* Operación Push */

void push(Stack_type *st_ptr,
                        Element_type elem){
  if(st_ptr == NULL)
     return;
  if (isFull(st_ptr))
     return;
  else {
     st_ptr->stack[st_ptr->top++] = elem;
     return;
  }
}
```

Implementación de pop

```
/* Operación Pop */

Element_type pop(Stack_type *st_ptr) {
  if (isEmpty(st_ptr))
     return -1;
  else
     return st_ptr->stack[--st_ptr->top];
}
```

Implementación de top

```
/* Operación Top */

Element_type top(Stack_type *st_ptr) {
      if (isEmpty(st_ptr))
      return(-1);
   else
      return st_ptr->stack[st_ptr->top-1];
}
```

Implementación de isEmpty

```
/* Determinar si la Pila está Vacía */

int isEmpty(Stack_type *st_ptr) {
  if(st_ptr == NULL)
     return(1);
  if(st_ptr->top == 0)
     return(1);
  else
     return(0);
}
```

Implementación de isFull

```
/* Determinar si la Pila está Llena */

int isFull(Stack_type *st_ptr) {
  if(st_ptr == NULL)
     return(1);
  if(st_ptr->top == STACKMAX)
     return(1);
  else
     return(0);
}
```

Ejemplo de uso de la Pila

```
void reverseAndPrint() {
  Element_type elem;

/*Una pila para contener los caracteres de
  entrada */
  Stack_type myStack;

  /* Crear una pila vacía */
  newStack(&myStack);

  /* Ingresar entrada */
  printf("Ingrese la cadena a  invertir: \n");
```

Ejemplo de uso de la Pila

```
/*Bucle para tomar la entrada y colocar cada
  carácter en la pila */

  // Ingresar primer caracter
  scanf("%c", &elem);
  while (elem != '\n') {

/* Colocar el carácter entrada en la pila */
     push(&myStack, elem);

  // Ingresar siguiente carácter
     scanf("%c", &elem);
  }
  /* En este punto los caracteres están en
  la pila */
```

Ejemplo de uso de la Pila

```
/* Imprimir la cadena invertida */
  printf("\nLa cadena invertida es:\n");

  /* Bucle para sacar elementos de la pila
   mientras la pila no está vacía */

  while (!isEmpty(&myStack)) {

/* Quitar el elemento tope de la pila */
     elem = pop(&myStack);

     /* Imprimir el valor del elemento */
     printf("%c",(char) elem);
  }

     printf("\n");
}
```

Ejemplo de uso de la Pila- Entrada y Salida

Ejemplo 1:

Ingrese la cadena a ser invertida:
Estructura de Datos y Algoritmos

La cadena invertida es:
somtiroglA y sotaD ed arutcurtsE

Ejemplo 2:

Ingrese la cadena a ser invertida :
Madam

La cadena invertida es :
madaM

Pila como una Lista Enlazada – Definiciones Básicas

```
typedef int Element_type;

// Definición para un nodo
typedef struct node {
  Element_type info;
  struct node *next;
} Node_type;

// Definir tope de una pila
typedef struct stacker {
     Node_type *top;
  } Stack_type;
```

Funciones createNode y newStack

```
/* Crear un nuevo nodo */

Node_type *createNode(Element_type,element) {
  Node_type *node;
  node = (Node_type *) \
           malloc(sizeof(Node_type));
  node->info = element;
  node->next = NULL;
  return node;
}

/* Crear una nueva Pila */
void newStack(Stack_type *st_ptr) {
  st_ptr->top = NULL;
}
```

Función isEmpty

```
/* Determinar si la Pila está Vacía */

int isEmpty(Stack_type *st_ptr) {
  if (st_ptr->top == NULL)
     return(1);
  else
     return(0);
}
```

Función top

```
/* Para obtener el elemento que está al
   frente de la pila */

Element_type top(Stack_type *st_ptr) {
   if (!isEmpty(st_ptr))
      return(st_ptr->top->info);
   else

   /* Devuelve -1 cuando la pila está
   vacía */
      return(-1);
}
```

Función push

```
/* Añadir un elemento a la pila */

void push(Stack_type *st_ptr,\
                  Element_type element) {
  Node_type *node;
  if(st_ptr == NULL)
     return;
  node = createNode(element);
  if (st_ptr->top == NULL)
     st_ptr->top = node;
  else {
     node->next = st_ptr->top;
     st_ptr->top = node;
  }
}
```

Función pop

```
/* Eliminar un elemento de la pila */

Element_type pop(Stack_type *st_ptr) {
  Node_type *q;
  Element_type element;
  if(st_ptr == NULL)  return (-1);
  if (isEmpty(st_ptr))

  /* Devuelve -1 cuando la pila está vacía
  no puede hacerse eliminaciones */
     return -1;
  q = st_ptr->top; element = q->info;
  st_ptr->top = q->next;
  free(q);
  return element;
}
```

Aplicaciones de Pilas

- Procesamiento de Cadenas
- Balanceo de Paréntesis en Expresiones Aritméticas
- Conversión de una Expresión en Notación Infija a Notación Postfija
- Evaluación de una Expresión en Notación Postfija
- Soporte a los Mecanismos de Llamada y Retorno de Función y Procedimiento
- Soporte para la Recursión
- Soporte para la Estrategia de Rastreo (Backtracking)

Resumen

- Definir la estructura de datos pila
- Explicar las diferentes operaciones definidas en la pila
- Describir las operaciones TDA en la pila
- Explicar la implementación de una pila usando un arreglo y una lista enlazada
- Discutir como las pilas pueden ser usados para resolver problemas

Unidad 5:

Laboratorio de Pilas

Unidad 6:

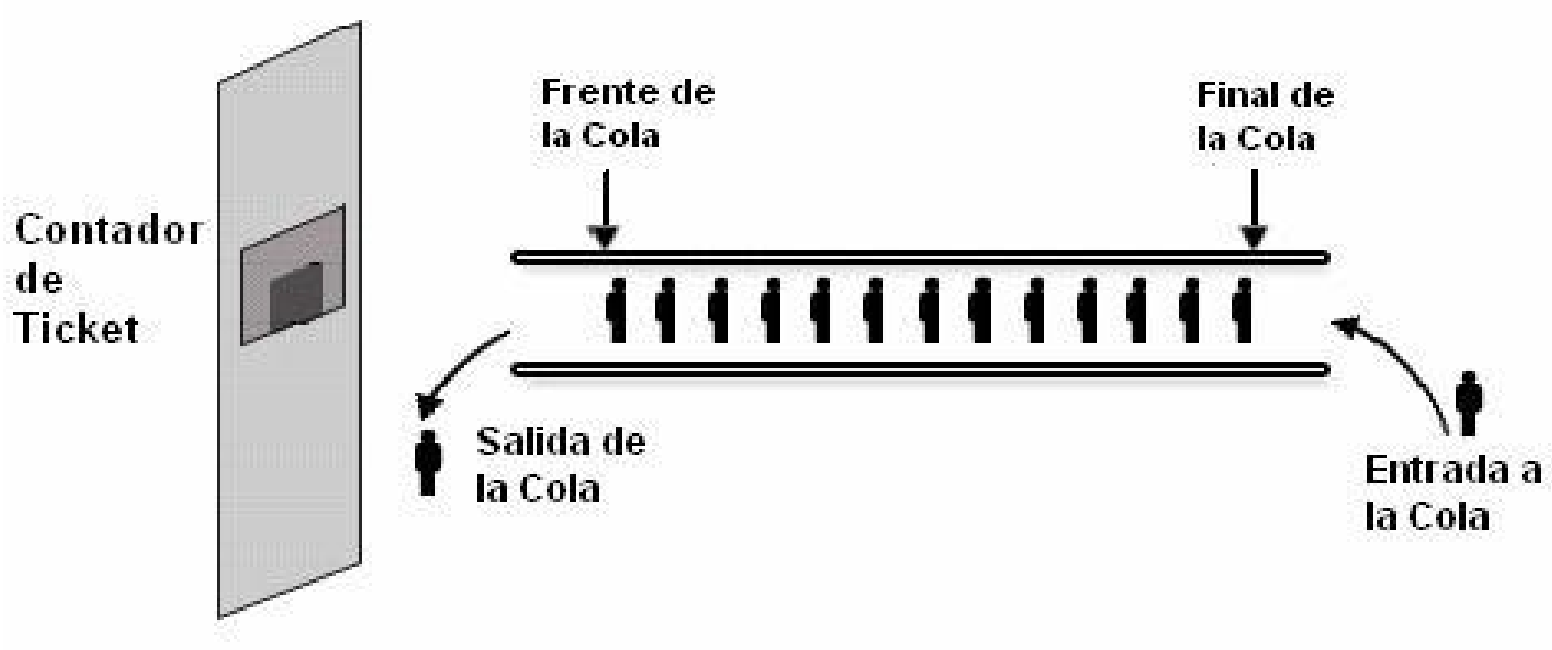
Colas

Objetivos del Aprendizaje

- Definir la estructura de datos de cola
- Explicar las operaciones TDA en colas
- Discutir la implementación de una cola como un arreglo
- Explicar las implementaciones de una cola como un arreglo circular
- Describir la implementación de una cola como una lista enlazada

Definición de Colas

- Una cola es una lista en la que los elementos se insertan en un extremo de la lista y se retiran en el otro extremo
- Una cola también se denomina una lista “Primero en Entrar, Primero en Salir” o simplemente FIFO (First-In First-Out).



Colas como un TDA

- `new (queue)` – Crea una nueva cola que está vacía
- `enqueue(queue, elemento)` – Inserta el elemento `elemento` en la parte posterior de la cola
- `dequeue(queue)` – Elimina el elemento de la cabeza o frente de la cola
- `front(queue)` – Devuelve el elemento de la cabeza o frente de la cola
- `isempty(queue)` – Devuelve verdadero si la cola está vacía, falso en caso contrario

Definición de dequeue usando new y enqueue

- Cuando la cola es new

```
dequeue(new(queue)) = error
```

- Si queue está vacía, la operación `enqueue(queue, elemento)` resultará en una cola con sólo `elemento`
- Si queue no está vacía, el `elemento` se agrega al final de la cola

```
  dequeue(enqueue(queue, elemento)) =
if isempty(queue) then new(queue)
  else enqueue(dequeue(queue, elemento))
```

Definición de front usando new y enqueue

- No puede devolver el elemento al frente de una cola vacía

```
front(new(queue)) = error
```

- En una cola vacía, `front(enqueue(queue, elemento))` devuelve `elemento`, ya que es el único elemento en la cola
- Para una cola no vacía, el resultado es el mismo que el resultado de la operación `front(queue)`

```
front(enqueue(queue, element)) =
  if isempty(queue) then element
  else front(queue)
```

Definición de isEmpty usando new y enqueue

- Una cola nueva está vacía

```
isempty(new) = true
```

- Cuando una cola está vacía, y un elemento se añade, la cola es no vacía
- Cuando una cola es no vacía, y un elemento se añade, el estado de la cola permanece no-vacío

```
isempty(enqueue(queue, elemento)) = false
```

Implementación de Colas como arreglos

- Siempre se debe mantener el control de ambos extremos de la cola
- Se puede mantener la cabeza de la cola siempre en la posición 0 del arreglo
- Agregar un elemento a la cola, significa incrementar el contador que muestra el final de la cola
- Eliminar un elemento de la cola es costoso, porque se deben mover hacia arriba en el arreglo los restantes elementos al eliminar el primer elemento del arreglo
- Cuando el tamaño de la cola es grande, este proceso será muy lento

Implementación de Colas como arreglos

```
typedef enum {false, true} boolean;
#define QUEUEMAX 100
typedef int Element_type;
typedef struct queuer {
  int front;
  int rear;
  Element_type queue[QUEUEMAX];
} Queue_type;
```

Crear una nueva Cola

```
/* New inicializa la cola a un estado vacío */

void newQueue(Queue_type *q_ptr) {
  q_ptr->front = 0;
  q_ptr->rear = -1;
}
```

Inserta en una Cola

```
/* Insertar un elemento en Q */

void enqueue(Queue_type *q_ptr, \
                  Element_type elem){
  if (isFull(q_ptr))
     return;
  else {
     q_ptr->rear++;
     q_ptr->queue[q_ptr->rear]= elem;
     return;
  }
}
```

Eliminar de una Cola

```
/* Eliminar un elemento de Q */

Element_type dequeue(Queue_type *q_ptr) {
  Element_type element;
  int i;
  if (isEmpty(q_ptr))
     return -1;
  else {
    element = q_ptr->queue[q_ptr->front];
    q_ptr->rear--;
    for (i = 0; i <= q_ptr->rear; i++)
     q_ptr->queue[i]=q_ptr->queue[i + 1];
    return element;
  }
}
```

Encontrar la cabeza o frente de la Cola

```
/* Operación front */

Element_type front(Queue_type *q_ptr) {
  if (isEmpty(q_ptr))
     return -1;
  else
     return q_ptr->queue[q_ptr->front];
}
```

Condición Empty y Full de una Cola

```
/* Para verificar si Q está vacía */

int isEmpty(Queue_type *q_ptr) {
  if(q_ptr->rear == -1)
     return(1);
  else
     return(0);
}
/* Para verificar si Q está llena */
int isFull(Queue_type *q_ptr) {
  if(q_ptr->rear == QUEUEMAX - 1)
     return(1);
  else
     return(0);
}
```

Operación de Eliminación – Otra Solución

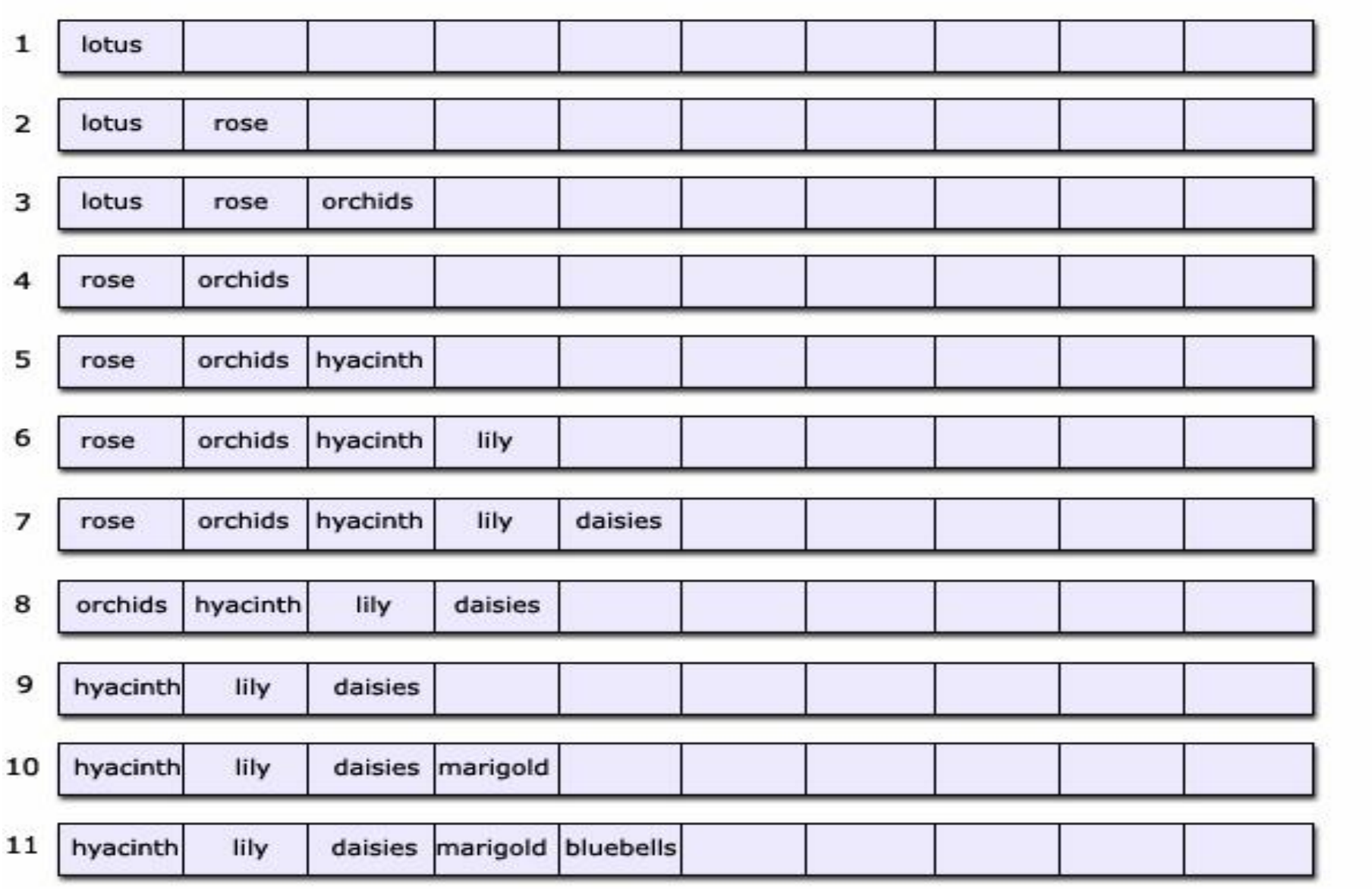
- En la primera solución, los elementos se desplazan hacia arriba en el arreglo cuando un elemento se elimina de la cola

- Para evitar tener que empujar los elementos hacia arriba en la cola por cada eliminación, se puede cambiar el valor de `front` cada vez que ocurra una eliminación, manteniendo vacantes las posiciones en la cabeza desde donde fueron retirados los elementos

Ejemplo de operaciones de Cola

```
Insertar: lotus
Insertar: rose
Insertar: orchids
Eliminar
Insertar: hyacinth
Insertar: lily
Insertar: daisies
Eliminar
Eliminar
Insertar: marigold
Insertar: bluebells
```

Primer Método

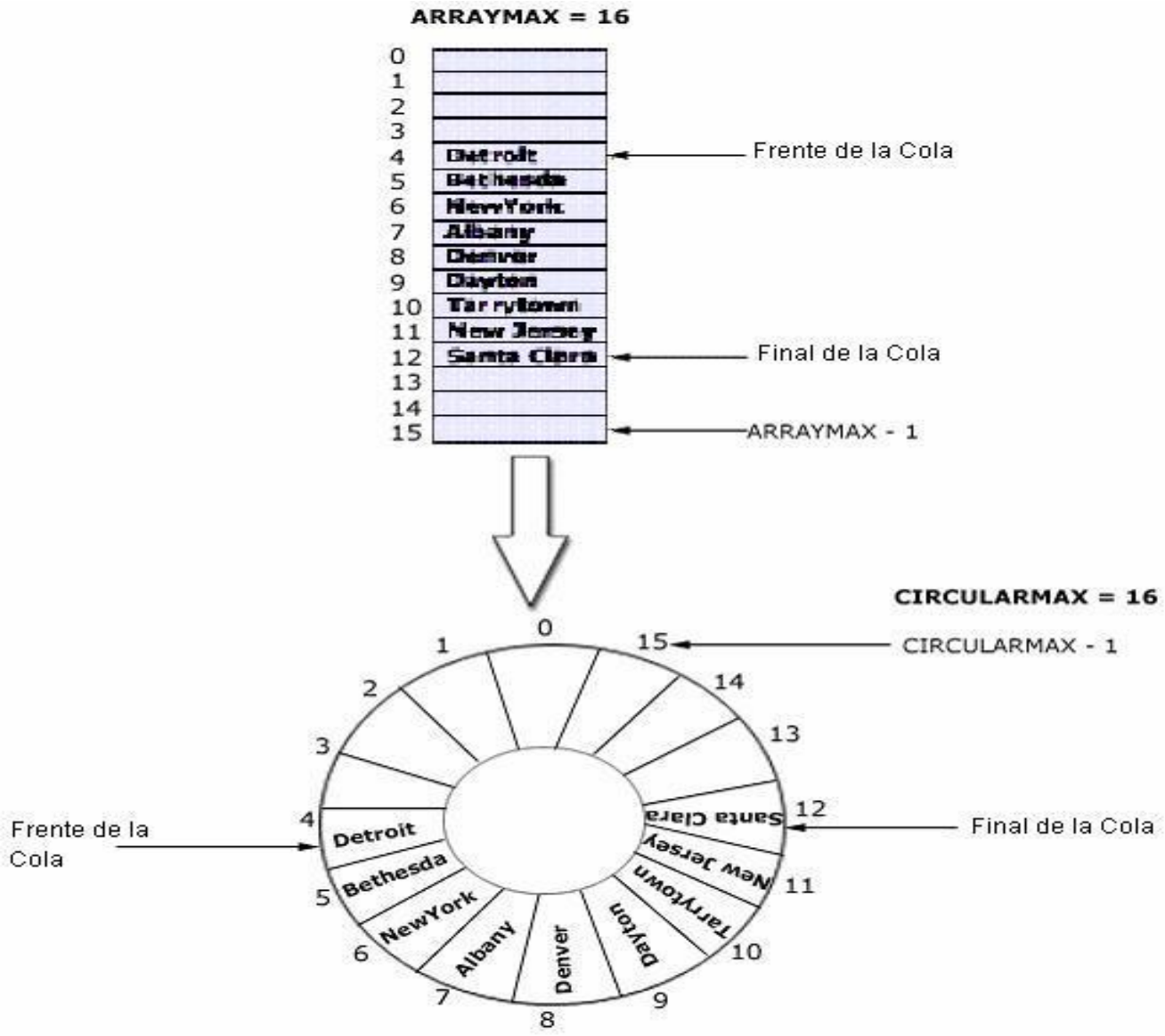


Segundo Método

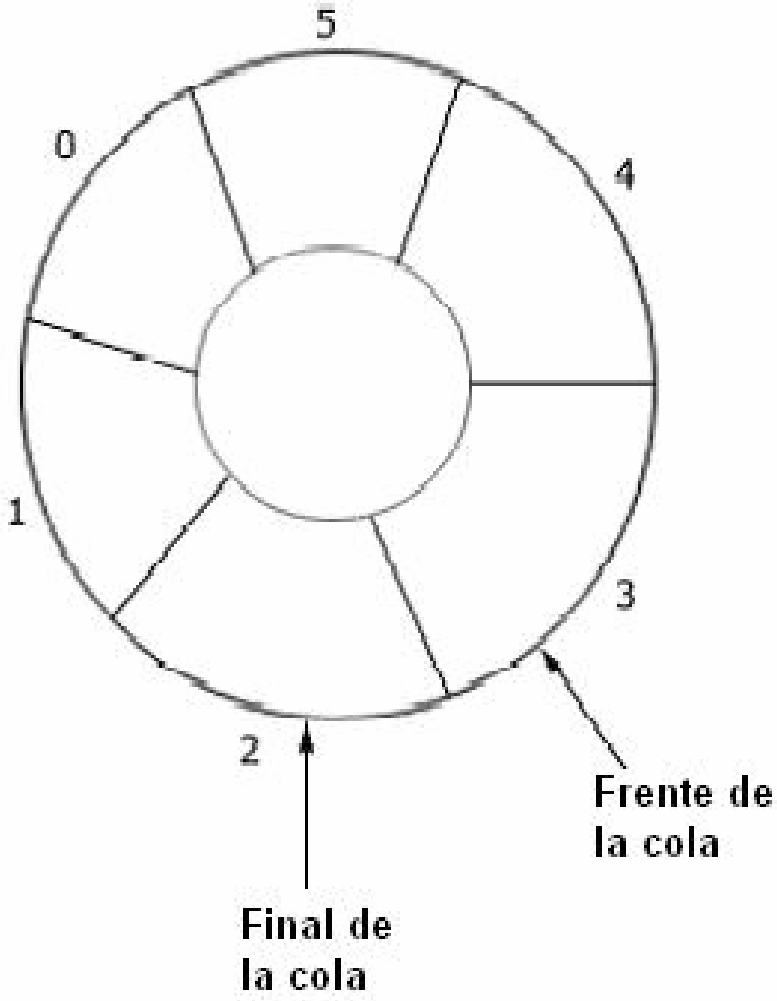
1	lotus									
2	lotus	rose								
3	lotus	rose	orchids							
4		rose	orchids							
5		rose	orchids	hyacinth						
6		rose	orchids	hyacinth	lily					
7		rose	orchids	hyacinth	lily	daisies				
8			orchids	hyacinth	lily	daisies				
9				hyacinth	lily	daisies				
10				hyacinth	lily	daisies	marigold			
11				hyacinth	lily	daisies	marigold	bluebells		

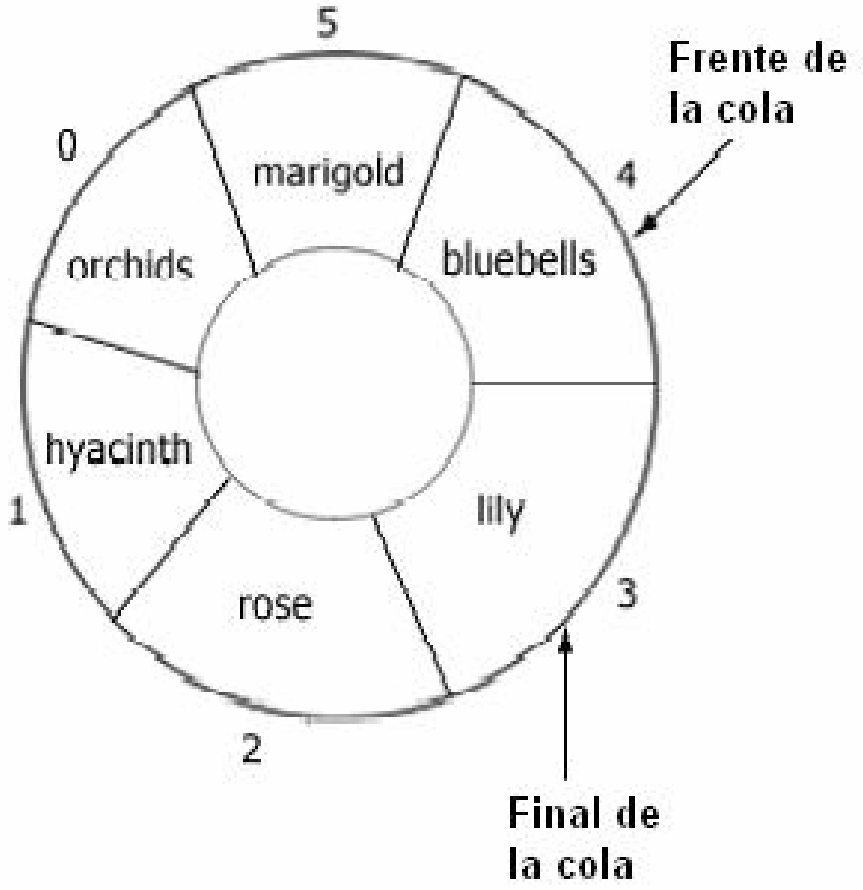
Usando arreglos circulares

- Un arreglo circular es uno en el que el primer elemento del arreglo es adyacente al último elemento del arreglo



Usando arreglos circulares





Usando arreglos circulares

- Para distinguir entre una cola vacía y una cola llena, se puede adoptar una de las siguientes soluciones:
  - Ø Tener variables Booleanas que indiquen si la cola está llena o vacía.
  - Ø Dejar explícitamente una celda en blanco entre `front` y `rear`.
  - ØUsar números negativos para representar `front` y/o `rear` para indicar que una cola está vacía.

Colas como Listas Enlazadas - Definición

```
typedef enum {falso,verdadero} boolean;

/* Tipo de dato definido por el usuario,
   para la cola */

typedef int Element_type;
 typedef struct node {
   Element_type info;
   struct node *next;
} Node_type;
 typedef struct queuer {
   Node_type *front;
   Node_type *rear;
} Queue_type;
```

Implementar createNode()

```
/* Crear un nuevo nodo */

Node_type          *createNode(Element_type
  element) {
  Node_type *node;
  node = (Node_type *)
  malloc(sizeof(Node_type));
  node->info = element;
  node->next = NULL;
  return node;
}
```

Implementar newQueue()

```
/* Crear de una nueva cola */

void newQueue(Queue_type *queue) {
  queue->front = NULL;
  queue->rear = NULL;
}
```

Implementar isEmpty()

```
/* Para verificar si la cola está vacía */

boolean isEmpty(Queue_type *queue) {
  if (queue->front == NULL)
     return verdadero;
  else
     return falso;
}
```

Implementar front()

```
/*  Para obtener el elemento que está al
    frente de la cola */

Element_type front(Queue_type *queue) {
    if (!isEmpty(queue))
        return((queue->front)->info);
    else
// Devuelve -1 cuando la cola está vacía
        return(-1);
}
```

Implementar enqueue()

```
/* Agrega un elemento a la cola */

void enqueue(Queue_type *queue, \
             Element_type element) {
  Node_type *node;
  node = createNode(element);

 /*en una Q cuyo frente es NULL*/
  if (queue->front == NULL) {
     queue->front = node;
     queue->rear = node;
  }
  else {/* Agregar al final Q */
     queue->rear->next = node;
     queue->rear = node;
  }
}
```

Implementar dequeue()

```
/* Eliminar un elemento de la cola */

Element_type dequeue(Queue_type *queue) {
  Node_type *q;
  Element_type elem;
  if (isEmpty(queue))

  /* Cuando la cola está vacía */
     return -1;
  if (queue->front == queue->rear) {
     elem = (queue->front)->info;
     free(queue->front);
     queue->front = NULL;
     queue->rear = NULL;
  }
```

Implementar dequeue()

```
else {
      q = queue->front;
      elem = q->info;
      queue->front = (queue->front)->next;
      free(q);
      }
  return elem;
}
```

Aplicaciones de las Colas

- Implementar las Colas de Impresión
- Datos del Buffer
- Simulación de Modelos de Líneas de Espera
- Recorrido Breadth First de Árboles
- Sistemas Operativos
- Administración del Tráfico de la Red
- Aplicaciones Comerciales en Línea

Resumen

- Definir la estructura de datos cola
- Explicar las operaciones TDA sobre colas
- Implementar una cola como un arreglo
- Explicar las implementación de una cola como un arreglo circular
- Describir la implementación de una cola como una lista enlazada

Unidad 7:

Laboratorio de Colas