


Repaso de Memoria Dinámica

Ejercicio 1


Defina la estructura de una lista enlazada de enteros. Implemente las siguientes funciones:

- ▶ Inicializar la lista
 - ▶ Eliminar todos los elementos de la lista
 - ▶ Agregar un elemento al principio de la lista
 - ▶ Agregar un elemento al final de la lista
 - ▶ Calcular la cantidad de elementos de la lista
 - ▶ Imprimir todos los elementos separados por coma
 - ▶ Acceder al elemento i -ésimo de la lista
- 

Memoria Dinámica – Repaso

Las implementaciones de los siguientes ejercicios están pensadas para discutir con los alumnos en la clase de repaso

Los objetivos de la clase son:

- ▶ Trabajar el manejo de memoria con listas enlazadas
 - ▶ Trabajar el manejo de memoria con vectores dinámicos
 - ▶ Trabajar el manejo de memoria con vectores de vectores dinámicos
 - ▶ Reforzar conceptos de encapsulamiento y abstracción
- 

Ejercicio 1 – Estructura

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

// Definicion de la estructura de un nodo de la lista enlazada
struct Nodo {
    int dato;
    struct Nodo* siguiente;
};

typedef struct Nodo* Lista;
```

Ejercicio 1 – Inicializar

```
// Inicializar la lista
void lista_inicializar(Lista* lista) {
    *lista = NULL;
}
```

Ejercicio 1 – Imprimir

```
// Imprimir todos los elementos de la lista separados por coma
void lista_imprimir(Lista lista) {
    Lista actual = lista;

    while (actual != NULL) {
        printf("%d", actual->dato);
        if (actual->siguiente != NULL) {
            printf(", ");
        }
        actual = actual->siguiente;
    }
    printf("\n");
}
```

Ejercicio 1 – Vaciar

```
// Eliminar todos los elementos de la lista
void lista_vaciar(Lista* lista) {
    Lista actual = *lista;
    Lista siguiente;

    while (actual != NULL) {
        siguiente = actual->siguiente;
        free(actual);
        actual = siguiente;
    }

    *lista = NULL;
}
```

Ejercicio 1 – Agregar Adelante

```
// Funciion para agregar un elemento al principio de la lista
void lista_agregar_adelante(Lista* lista, int nuevoDato) {
    Lista nuevo = (Lista)malloc(sizeof(struct Nodo));
    nuevo->dato = nuevoDato;
    nuevo->siguiente = *lista;
    *lista = nuevo;
}
```


Ejercicio 1 – Agregar al Final

```
// Agregar un elemento al final de la lista
void lista_agregar_al_final(Lista* lista, int nuevoDato) {
    Lista nuevo = (Lista)malloc(sizeof(struct Nodo));
    Lista ultimo = *lista;

    nuevo->dato = nuevoDato;
    nuevo->siguiente = NULL;

    if (*lista == NULL) {
        *lista = nuevo;
        return;
    }

    while (ultimo->siguiente != NULL) {
        ultimo = ultimo->siguiente;
    }

    ultimo->siguiente = nuevo;
}
```

Ejercicio 1 – Contar Elementos

```
// Calcular la cantidad de elementos de la lista
int lista_contar(Lista lista) {
    int contador = 0;
    Lista actual = lista;

    while (actual != NULL) {
        contador++;
        actual = actual->siguiente;
    }

    return contador;
}
```

Ejercicio 1 – Acceder a i-esimo elemento

```
// Obtener el elemento i-esimo de la lista
int lista_elemento(Lista lista, int indice) {
    int contador = 0;
    Lista actual = lista;

    while (actual != NULL) {
        if (contador == indice) {
            return actual->dato;
        }
        contador++;
        actual = actual->siguiente;
    }

    // Si el indice esta fuera de rango, se retorna -1
    return -1;
}
```

Ejercicio 2

Redefina la estructura de la lista enlazada de enteros para optimizar algunas funciones para garantizar $O(1)$ en su ejecución. Adapte todas las funciones a la nueva estructura de la lista y garantice $O(1)$ en siguientes funciones:

- ▶ Agregar un elemento al final de la lista
- ▶ Calcular la cantidad de elementos de la lista

Ejercicio 2 – Estructura

```
// Estructura de un nodo de la lista enlazada
struct Nodo {
    int dato;
    struct Nodo* siguiente;
};

// Álias al puntero de nodo, lista simple
typedef struct Nodo* ListaPtr;

// Definicion de la estructura de la lista, para optimizaciones
typedef struct {
    ListaPtr primero;
    ListaPtr ultimo;
    int cantidad;
} Lista;
```

Ejercicio 2 – Inicializar

```
// Funcion para inicializar la lista
void lista_inicializar(Lista* lista) {

    lista->primero = NULL; // puntero al primero
    lista->ultimo = NULL;  // puntero al ultimo
    lista->cantidad = 0;    // cantidad de elemntos
}
```

Ejercicio 2 – Imprimir

```
// Imprimir todos los elementos de la lista separados por coma
void lista_imprimir(Lista lista) {

    ListaPtr actual = lista.primeros;

    while (actual != NULL) {
        printf("%d", actual->dato);
        if (actual->siguiente != NULL) {
            printf(", ");
        }
        actual = actual->siguiente;
    }
    printf("\n");
}
```

Ejercicio 2 – Vaciar

```
// Eliminar todos los elementos de la lista
void lista_vaciar(Lista* lista) {
    ListaPtr actual = lista->primero;
    ListaPtr siguiente;

    while (actual != NULL) {
        siguiente = actual->siguiente;
        free(actual);
        actual = siguiente;
    }

    lista->primero = NULL;
    lista->ultimo = NULL;
    lista->cantidad = 0;
}
```


Ejercicio 2 – Acceder a i-esimo elemento

```
// Obtener el elemento i-esimo de la lista
int lista_elemento(Lista lista, int indice) {
    if (indice < 0 || indice >= lista.cantidad) {
        printf("Error: Indice fuera de rango\n");
        return -1;
    }

    ListaPtr actual = lista.primeros;
    int contador = 0;

    while (contador < indice) {
        actual = actual->siguiente;
        contador++;
    }

    return actual->dato;
}
```

Ejercicio 2 – Agregar Adelante

```
// Agregar un elemento al principio de la lista
void lista_agregar_adelante(Lista* lista, int nuevoDato) {

    ListaPtr nuevo = (ListaPtr)malloc(sizeof(struct Nodo));
    nuevo->dato = nuevoDato;
    nuevo->siguiente = lista->primero;
    lista->primero = nuevo;

    if (lista->ultimo == NULL) {
        lista->ultimo = nuevo;
    }

    lista->cantidad++;
}
```

Ejercicio 2 – Agregar al Final

```
// Agregar un elemento al final de la lista
void lista_agregar_al_final(Lista* lista, int nuevoDato) {

    ListaPtr nuevo = (ListaPtr)malloc(sizeof(struct Nodo));
    nuevo->dato = nuevoDato;
    nuevo->siguiente = NULL;

    if (lista->ultimo == NULL) {
        lista->primero = nuevo;
    } else {
        lista->ultimo->siguiente = nuevo;
    }

    lista->ultimo = nuevo;
    lista->cantidad++;
}
```

Ejercicio 2 – “Contar” Elementos

```
// “Calcular” la cantidad de elementos de la lista
int lista_contar(Lista lista) {
    return lista.cantidad;
}
```

Ejercicio 3

Redefina la estructura de la lista enlazada de enteros anterior para garantizar $O(1)$ al acceder a los elementos. Adapte todas las funciones a la nueva estructura de la lista y garantice $O(1)$ en siguientes funciones (aunque las demás pierdan $O(1)$):

- ▶ Acceder al elemento i -ésimo de la lista
- ▶ Eliminar todos los elementos de la lista

Ejercicio 3 – Estructura

```
// Definicion de la estructura de la lista
typedef struct {
    int *elementos;    // puntero para almacenar los elementos
    int cantidad;      // dimension logica
    int capacidad;     // dimension fisica
    int crecimiento;   // cantidad de elementos para crecer la lista
} Lista;
```

Ejercicio 3 – Inicializar

```
// Inicializar la lista
void lista_inicializar(Lista *lista, int crecimiento) {

    lista->elementos = NULL; // inicialmente no hay elementos
    lista->cantidad = 0;      // cantidad inicial de elementos
    lista->capacidad = 0;     // capacidad inicial
    lista->crecimiento = crecimiento; // para crecer la lista
}
```

Ejercicio 3 – Imprimir

```
// Imprimir todos los elementos de la lista separados por coma
void lista_imprimir(Lista lista) {
    for (int i = 0; i < lista.cantidad; i++) {
        printf("%d", lista.elementos[i]);
        if (i < lista.cantidad - 1) {
            printf(", ");
        }
    }
    printf("\n");
}
```


Ejercicio 3 – Vaciar

```
// Eliminar la lista y liberar memoria
void lista_limpiar(Lista *lista) {

    free(lista->elementos);
    lista->elementos = NULL;
    lista->cantidad = 0;
    lista->capacidad = 0;
}
```

Ejercicio 3 – Acceder a i-esimo elemento

```
// Obtener el elemento i-esimo de la lista
int lista_elemento(Lista lista, int indice) {

    if (indice < 0 || indice >= lista.cantidad) {
        return -1;
    }
    return lista.elementos[indice];
}
```

Ejercicio 3 – Agregar al Final

```
// Agregar un elemento a la lista
void lista_agregar_al_final(Lista *lista, int nuevoDato) {
    // Verificar si la capacidad actual es suficiente
    if (lista->cantidad >= lista->capacidad) {
        // Si no es suficiente, aumentar la capacidad
        lista->capacidad += lista->crecimiento;
        lista->elementos = realloc(lista->elementos, lista->capacidad*sizeof(int));
    }

    // Agregar el nuevo dato
    lista->elementos[lista->cantidad] = nuevoDato;
    lista->cantidad++;
}
```

Ejercicio 3 – Agregar Adelante – v1

```
// Agregar un elemento al principio de la lista
void lista_agregar_adelante2(Lista *lista, int nuevoDato) {
    // Si no es suficiente, aumentar la capacidad
    if (lista->cantidad >= lista->capacidad) {
        lista->capacidad += lista->crecimiento;
        lista->elementos = realloc(lista->elementos, lista->capacidad*sizeof(int));
    }

    // Desplazar elementos hacia la derecha para hacer espacio para nuevo dato
    for (int i = lista->cantidad; i > 0; i--) {
        lista->elementos[i] = lista->elementos[i - 1];
    }

    lista->elementos[0] = nuevoDato; // Agregar el nuevo dato al principio
    lista->cantidad++;
}
```

Ejercicio 3 – Agregar Adelante – v2

```
void lista_agregar_adelante(Lista *lista, int nuevoDato) {
    int *elementos;
    // Verificar si la capacidad actual es suficiente
    if (lista->cantidad >= lista->capacidad) {
        lista->capacidad += lista->crecimiento;
        elementos = (int*) malloc((lista->capacidad) * sizeof(int));
    } else {
        elementos = lista->elementos;
    }
    // Copia segura de lista->elementos en elementos -1
    memmove(elementos + 1, lista->elementos, lista->cantidad * sizeof(int));
    elementos[0] = nuevoDato; // Agregar el nuevo dato al principio
    lista->cantidad++;

    if (lista->elementos != elementos) {
        free(lista->elementos);
        lista->elementos = elementos;
    }
}
```


Para evitar copiar 2 veces
los elementos conviene
utilizar
Malloc + free
Para copiar los elementos
de manera eficiente y
segura usamos
memmove (string.h)

Ejercicio 4 – Matrices Ralas

- ▶ Una matriz rala (sparse matrix) tiene la mayoría de sus elementos en cero. Normalmente, tiene muy pocos elementos no nulos en comparación con el número total de elementos
- ▶ Se utiliza en áreas, como el álgebra lineal numérica, el procesamiento de señales, el aprendizaje automático
- ▶ Debido a que pueden tener tamaños extremadamente grandes es importante manejarlas eficientemente
- ▶ Ventajas:
 - **Ahorro de espacio de almacenamiento:** no almacenar explícitamente los ceros, se reduce significativamente el espacio requerido
 - **Eficiencia de cálculo:** los algoritmos adaptados para matrices ralas evitan operaciones innecesarias con los ceros, acelerando cálculos

Ejercicio 4 – Matrices Ralas

Defina la estructura eficiente para implementar una matriz rala de números reales. Implemente las siguientes funciones:

- ▶ crear la matriz
 - ▶ Destruir la matriz
 - ▶ Acceder al elemento de una (fila, columna)
 - ▶ Modificar un elemento de una (fila, columna)
- 

Ejercicio 4 – Estructura

```
// Estructura para nodos de lista enlazada de indice de columna y valor
typedef struct _NodoColValor {
    int columna;
    double valor;
    struct _NodoColValor* siguiente;
} NodoColValor;

// Estructura para nodos de lista enlazada de Índices de fila
typedef struct _NodoFila {
    int fila;
    NodoColValor* valores;
    struct _NodoFila* siguiente;
} NodoFila;

// Estructura para la matriz rala
typedef struct {
    int valor_defecto;
    int cant_filas;
    int cant_cols;
    NodoFila* filas;
} MatrizRala;
```


Ejercicio 4 – Crear Matriz

```
// Crear una matriz rala
```

```
void matriz_crear(MatrizRala* matriz, int valor_ini, int filas, int cols) {  
  
    matriz->valor_defecto = valor_ini;  
    matriz->cant_filas = filas;  
    matriz->cant_cols = cols;  
    matriz->filas = NULL;  
}
```

Ejercicio 4 – Destruir Matriz

```
// Destruir una matriz rala
void matriz_destruir(MatrizRala* matriz) {
    NodoFila* tempFila;
    NodoColValor* tempColValor;

    while (matriz->filas != NULL) {
        tempFila = matriz->filas;
        matriz->filas = matriz->filas->siguiente;

        while (tempFila->valores != NULL) {
            tempColValor = tempFila->valores;
            tempFila->valores = tempFila->valores->siguiente;
            free(tempColValor);
        }

        free(tempFila);
    }
}
```

Ejercicio 4 – Ver Elemento

```
int matriz_ver_elemento(MatrizRala* matriz, int fila, int col) {
    NodoFila* tempFila = matriz->filas;
    // Busca fila, puede no existir
    while (tempFila != NULL && tempFila->fila < fila) {
        tempFila = tempFila->siguiente;
    }

    if (tempFila != NULL && tempFila->fila == fila) {
        NodoColValor* tempColValor = tempFila->valores;
        // Busca en lista de columnas, puede no existir
        while (tempColValor != NULL && tempColValor->columna < col) {
            tempColValor = tempColValor->siguiente;
        }
        // si encontro => retorna valor
        if (tempColValor != NULL && tempColValor->columna == col) {
            return tempColValor->valor;
        }
    }

    return matriz->valor_defecto; // no encontro => retorna valor por defecto
}
```

Ejercicio 4 – Modificar Elemento

```
void matriz_modificar_elemento(MatrizRala* matriz, int fila, int col, double valor) {
    NodoFila* tempFila = matriz->filas;
    NodoFila* prevFila = NULL;

    // Buscar la fila
    while (tempFila != NULL && tempFila->fila < fila) {
        prevFila = tempFila;
        tempFila = tempFila->siguiente;
    }

    // Si no se encuentra la fila, se debe crear
    if (tempFila == NULL || tempFila->fila != fila) {
        NodoFila* nuevaFila = (NodoFila*)malloc(sizeof(NodoFila));
        nuevaFila->fila = fila;
        nuevaFila->valores = NULL;
        nuevaFila->siguiente = tempFila;

        // Actualizar la referencia de la fila anterior o la cabeza de la lista
        if (prevFila != NULL) {
            prevFila->siguiente = nuevaFila;
        } else {
            matriz->filas = nuevaFila;
        }

        tempFila = nuevaFila;
    } // continua en siguiente diapositiva
```

Ejercicio 4 – modificar Elemento

```
// continua de diapositive anterior, en este punto encuentro la fila de entrada o la creo

// Buscar o insertar el valor en la columna correspondiente
NodoColValor* tempColValor = tempFila->valores; // apunta columnas (c/nodo tiene valor de columna)
NodoColValor* prevColValor = NULL;
// busca columna en lista de columnas
while (tempColValor != NULL && tempColValor->columna < col) {
    prevColValor = tempColValor;
    tempColValor = tempColValor->siguiente;
}
// Si la columna ya existe, se actualiza con el nuevo valor
if (tempColValor != NULL && tempColValor->columna == col) {
    tempColValor->valor = valor;
} else { // Si no existe, se inserta
    NodoColValor* nuevoColValor = (NodoColValor*)malloc(sizeof(NodoColValor));
    nuevoColValor->columna = col;
    nuevoColValor->valor = valor;
    nuevoColValor->siguiente = tempColValor;

    // Actualizar la referencia del valor anterior o la cabeza de la lista
    if (prevColValor != NULL) {
        prevColValor->siguiente = nuevoColValor;
    } else {
        tempFila->valores = nuevoColValor;
    }
}
```