Universidad católica Andrés Bello
Escuela de Ingeniería en Informática
Cátedra: Algoritmos y Programación II

TAD: Lista XOR

Docente:
Larez, Jesús

Integrantes:
Patiño, Angel
González, Leonardo

Ciudad Guayana, Enero del 2023

Análisis del Problema:

Premisas:

1. Los nodos de la lista solamente tendrán la dirección del nodo anterior y el nodo siguiente únicamente abarcando un simple campo o espacio en memoria.
2. El operador XOR será utilizado para la asignación de las direcciones en los nodos y para varias operaciones en la lista.
3. En todo momento se tendrá la referencia al primer elemento (Cabeza) y último elemento (Final) de la lista gracias a la existencia de un nodo que guardará apuntadores a ambas secciones.
4. Las funciones **sacar…** eliminaran un elemento en cierta posición de la lista, pero además guardara dicho elemento en una variable.

Requerimientos:

1. Para la compilación del programa, se deberá usar el fichero makefile, el cual ya estará previamente definido en la solución. Únicamente se deberá introducir el comando **make** en la terminal, y todos los archivos serán compilados a la vez. Además, se espera que el equipo del usuario ya tenga instalado todo lo necesario para el uso de **make** en Linux.

2. Recomendado el uso de la terminal nativa de los distros de Linux, sobre todo para mejor visualización del menú del programa.

Análisis del Problema:

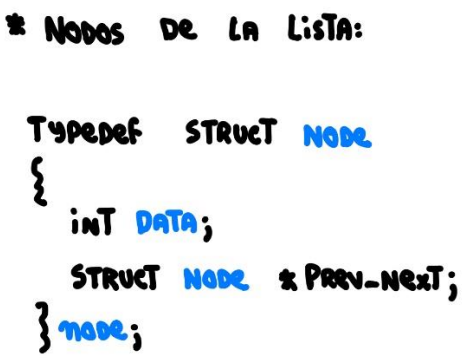
ENTRADAS	PROCESOS	SALIDAS
Operación a realizar sobre la lista: 0 <= Operación <= 12	1. Creación de una nueva lista. 2. Acceso al menú principal de la aplicación. 3. Seleccionar entre las 12 operaciones disponibles sobre la lista. 4. Mostrar la salida.	* Resultados obtenidos por cada caso de operación especificado en el menú del programa. * Cierre del programa por decisión del usuario.

Diseño de la Solución:

Descripción:

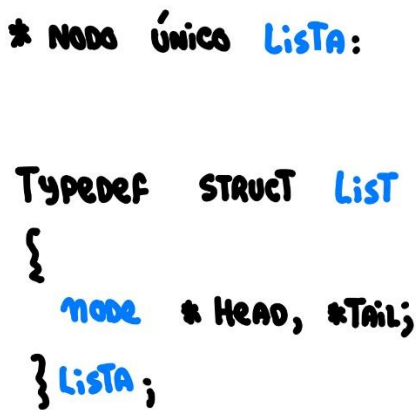
Una lista doblemente enlazada es una lista en la cual cada nodo contiene 2 campos de direcciones en los cuales se guardará la referencia al nodo anterior y al nodo siguiente. Para efectos del proyecto, se solicita una implementación que guarde la referencia de dichos nodos, sin embargo, permitiendo solo el uso de un único campo de dirección en el nodo además del campo donde se guarda el valor o elemento respectivo del nodo.

Por ende, cada nodo de la lista será implementado mediante el uso de la siguiente estructura:



Data: Guardar el valor o elemento respectivo del nodo.
Prev_next: Guardara la dirección del nodo anterior y siguiente en un único campo.

Además, para la creación de la lista, es recomendado el crear un nodo único que guardará la dirección del primer elemento (Cabeza) y del ultimo elemento (Final) de la lista, lo cual permitirá una implementación mas eficaz de las operaciones pertenecientes al TAD Lista.



Head: Apuntador hacia el primer elemento de la lista.

Tail: Apuntador hacia el último elemento de la lista.

Guardar la dirección del nodo anterior y siguiente en un solo campo del nodo de la lista es posible gracias al uso del operador binario XOR (^) y sus múltiples propiedades. Gracias a esto, es posible el implementar operaciones que aprovechen la cualidad del desplazamiento de derecha a izquierda o de izquierda a derecha.

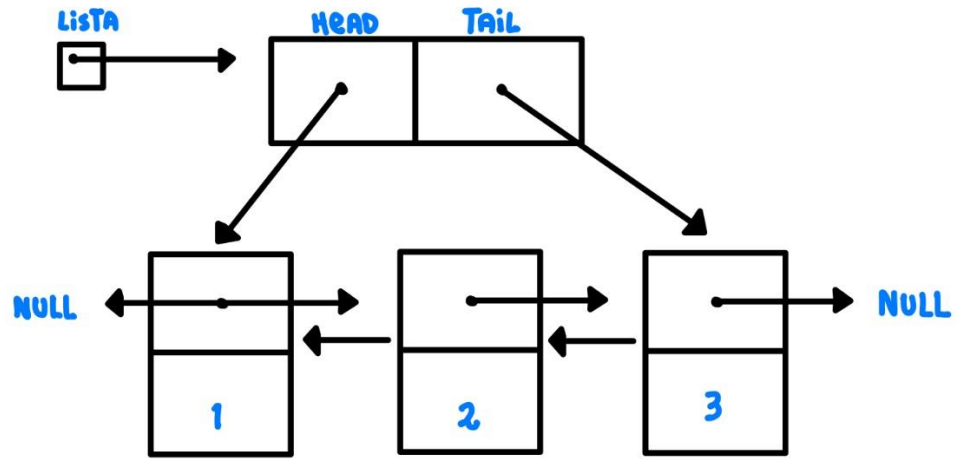
Debido a que la operación XOR se realizara de forma constante en la lista, se recomienda la creación de una función que permita el realizar la acción entre 2 punteros, por ende, se considero adecuado el uso de la librería <stdint.h> para poder acceder a (uintptr_t) lo cual permite cambiar la dirección guardada por el apuntador a entero, luego se procede a hacer el XOR y finalmente se retorna de entero a apuntador:

- Función XOR:

```
node * XOR(node *x, node *y)
{
    node *Tmp_x, *Tmp_y;
    Tmp_x = x;
    Tmp_y = y;
    RETURN (node *)((uintptr_t)(x) ^ (uintptr_t)(y));
}
```

Como ejemplo, consideremos una lista de 3 nodos realizada bajo la estructura anteriormente definida, el resultado será el siguiente:

- LISTA XOR DE 3 ELEMENTOS:



Funcionalidad del XOR y Propiedad clave:

La propiedad más importante del XOR para este caso será:

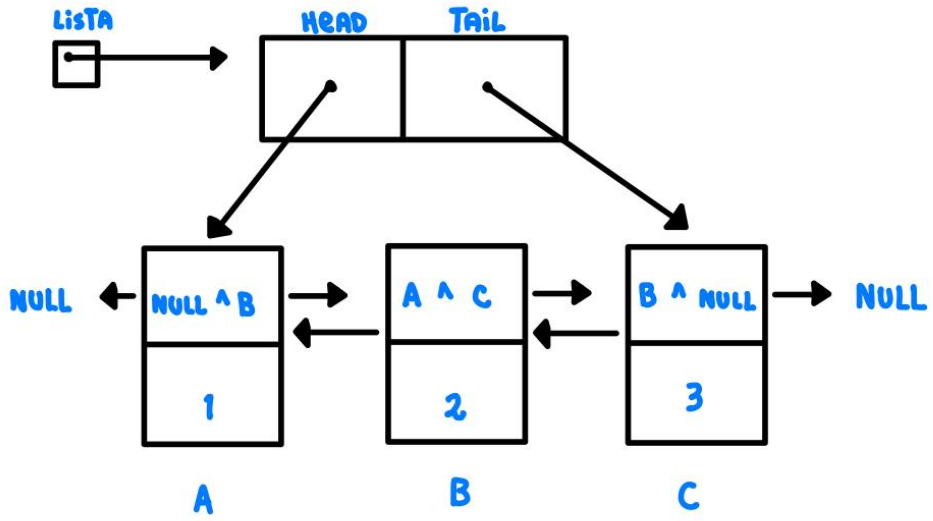
(X ^ Y) ^ X = Y

(X ^ Y) ^ Y = X

Pues es gracias a esta propiedad que será posible el iterar mediante la lista tanto de derecha a izquierda, siempre trabajando con un apuntador en el nodo actual, otro apuntador en el nodo al que se desea iterar y un apuntador en el anterior para simplificar el trabajo del XOR.

En el siguiente diagrama de la misma lista de 3 elementos queda mas claro el almacenamiento de direcciones:

• ALMACENAMIENTO DE DIRECCIONES:



A → PREV_NEXT = XOR (NULL, B)

B → PREV_NEXT = XOR (A, C)

C → PREV_NEXT = XOR (B, NULL)

Justificación de las Estructuras de Datos y Algoritmos utilizados:

El principal motivo por el cual se considero este plan de implementación es por el hecho de que muchas de las operaciones base del TAD Lista se benefician de la existencia de un apuntador al primer elemento de la lista y un apuntador al último elemento, permitiendo que la iteración de Inicio a Final o Final a Inicio sea mucho mas sencillo, por nombrar un ejemplo.

En el caso del XOR, siendo un operador binario base en el lenguaje de programación C, se considero optimo el implementar una función que sea capaz de realizar un XOR entre apuntadores con simplemente el introducir 2 punteros como argumentos para aplicar la propiedad de XOR explicada anteriormente o simplemente para guardar la referencia del nodo anterior y próximo.

Detalles de la Implementación:

Descripción de las funciones utilizadas en el programa

Lista crearLista():

Esta función será utilizada para la creación de una nueva lista. Asignara memoria mediante **malloc()** y asignara los punteros Head y Tail con un valor NULL.

Inicializar(Lista):

Elimina todos los elementos de la lista y asigna los apuntadores de Head y Tail con un valor de NULL.

bool esVacia(Lista):

Retorna 1 si la lista enviada como argumento esta vacía (es decir, no contiene elementos) o retorna 0 si dicha lista contiene mínimo un elemento.

bool insertarPrincipio(Lista, elementoTipo):

Crea y asigna un nuevo nodo a la lista en el inicio, además lo asigna como nueva cabeza de la lista. Siempre se actualizará el **prev_next** del antiguo primer elemento de la lista para que pase a ser el segundo elemento de la lista.

bool insertarFinal(Lista, elementoTipo):

Crear y asigna un nuevo nodo al final de la lista, además este nuevo nodo será el nuevo final de la lista. Siempre se actualiza el **prev_next** del antiguo ultimo elemento de la lista para que este pase a ser el penúltimo elemento de la lista.

bool insertarOrdenado(Lista, elementoTipo):

Insertara un nodo de forma ordenada en la lista, en el primer espacio en el que encuentre un lugar será insertado, es por eso que se le recomienda al usuario usar esta función con listas ordenada (La aplicación ordenara las listas antes de insertar el elemento). Siempre se actualizará el **prev_next** de los nodos anterior y siguiente afectados por insertar el nuevo nodo.

bool buscar(Lista, elementoTipo):

Retorna 1 si se encuentra en la lista el elemento enviado como argumento por el usuario o en su defecto, retornara 0 si dicho elemento no se encuentra en la lista o si la lista esta vacía.

bool sacarPrincipio(Lista, elementoTipo):

Elimina el primer nodo de la lista y asigna su valor (nodo->data) a un apuntador de **elementoTipo**. El segundo nodo de la lista pasara a ser el primero, por lo tanto, el apuntador de Head guardara la dirección del segundo nodo de la lista.

bool sacarFinal(Lista, elementoTipo):

Elimina el ultimo nodo de la lista y asigna su valor (nodo->data) a un apuntador de **elementoTipo**. El penúltimo nodo de la lista pasara a ser el último, por lo tanto L->tail guardara la dirección del antiguo penúltimo nodo.

bool sacarPrimeraOcurrencia(Lista, elementoTipo):

Elimina el nodo que coincida con el elemento introducido como argumento por el usuario, si dicho nodo esta al inicio o al final de la lista, se podrá utilizar **sacarPrincipio** o **sacarFinal** para mejorar la eficacia de la solución.

listarInicioAFinal(Lista):

Imprime en consola todos los elementos de la lista de Derecha a Izquierda o desde Head hasta Tail. Se inicia desde L->head y se itera mientras que el apuntador auxiliar difiera de NULL.

listarFinalAInicio(Lista):

Imprime en consola todos los elementos de la lista de Izquierda a Derecha o desde Tail hasta Head. Se inicia desde L->tail y se itera mientras que el apuntador auxiliar difiera de NULL.

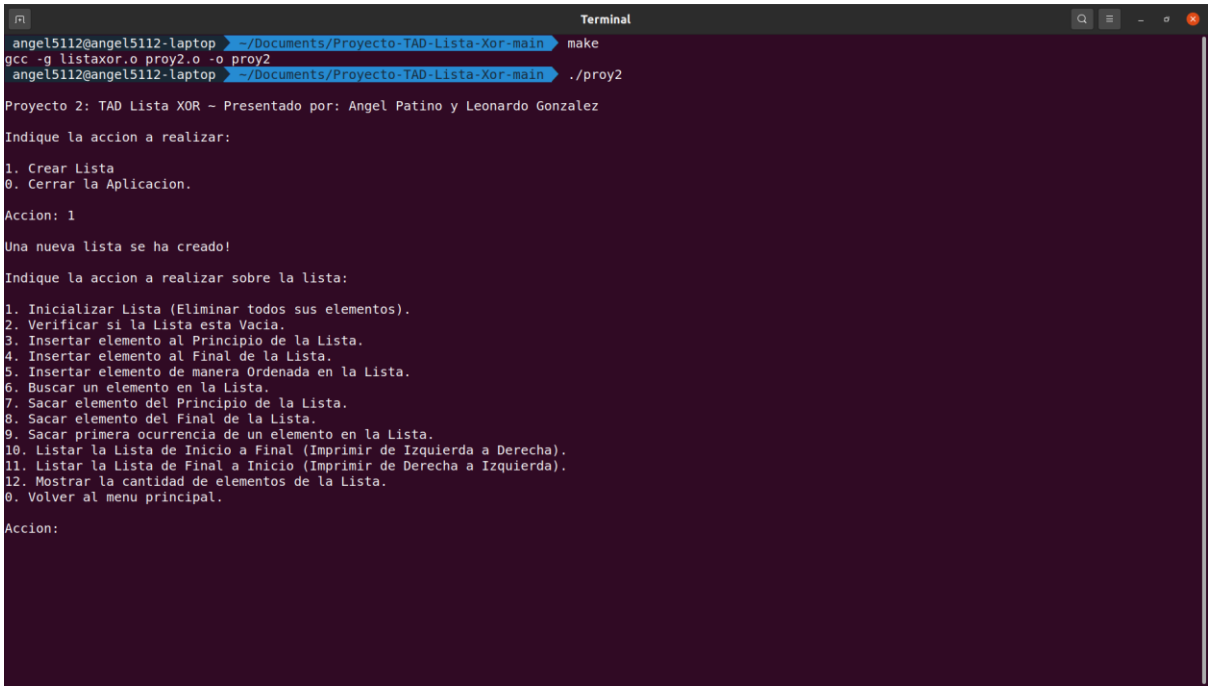
entero cantidadElementos(Lista):

Inicializa un contador y procede a iterar desde L->head hasta L->tail contando nodo por nodo hasta llegar al final de la lista. Retornara la cantidad de elementos existentes en la lista (El valor del contador creado dentro de la función).

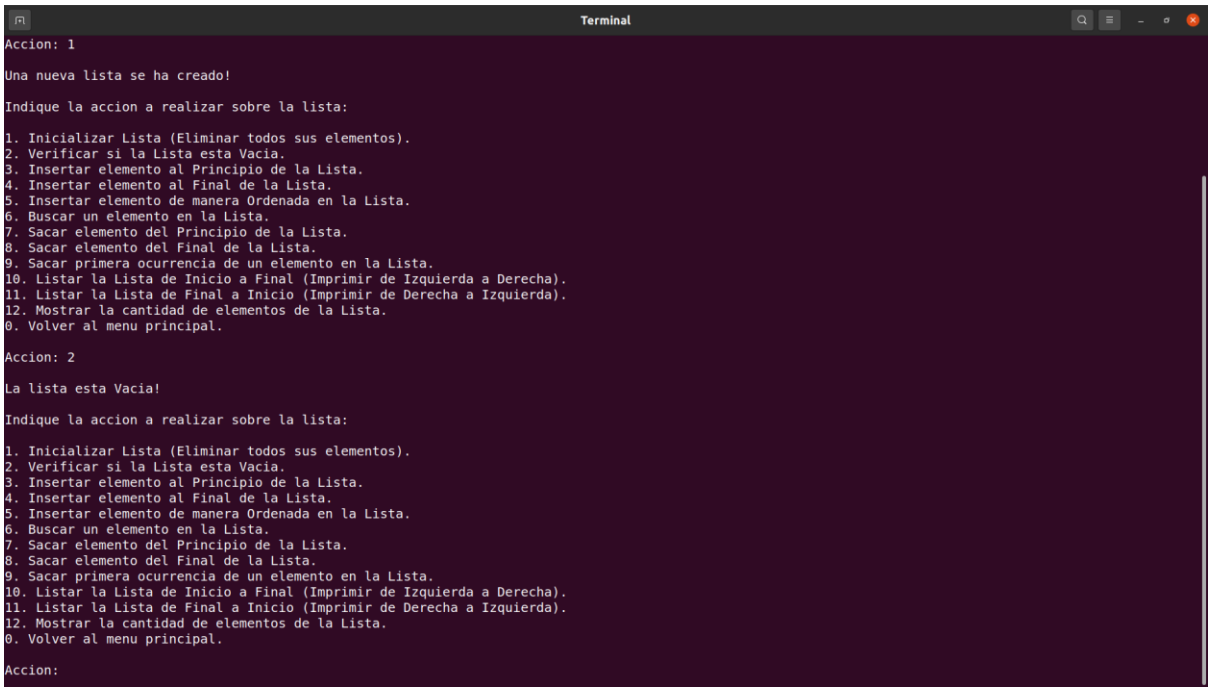
Casos de Prueba:

Creación de lista XOR y uso de todas las operaciones (funciones) definidas anteriormente

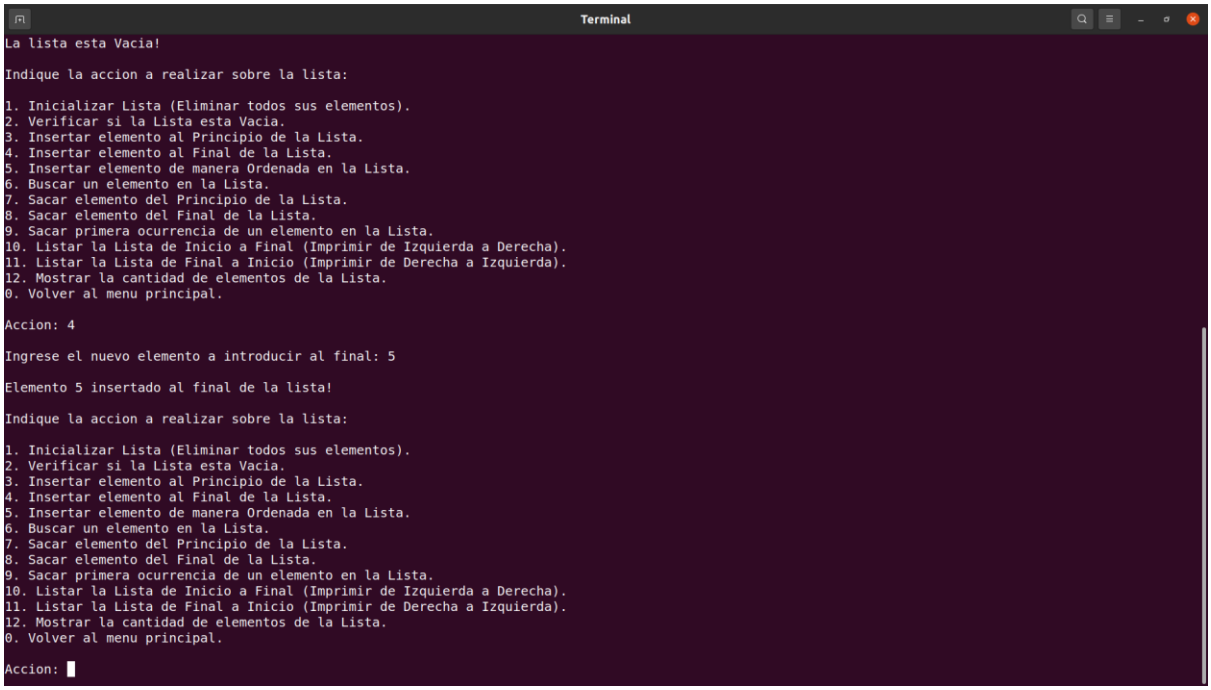
Menu Inicial y Creacion de la Lista:



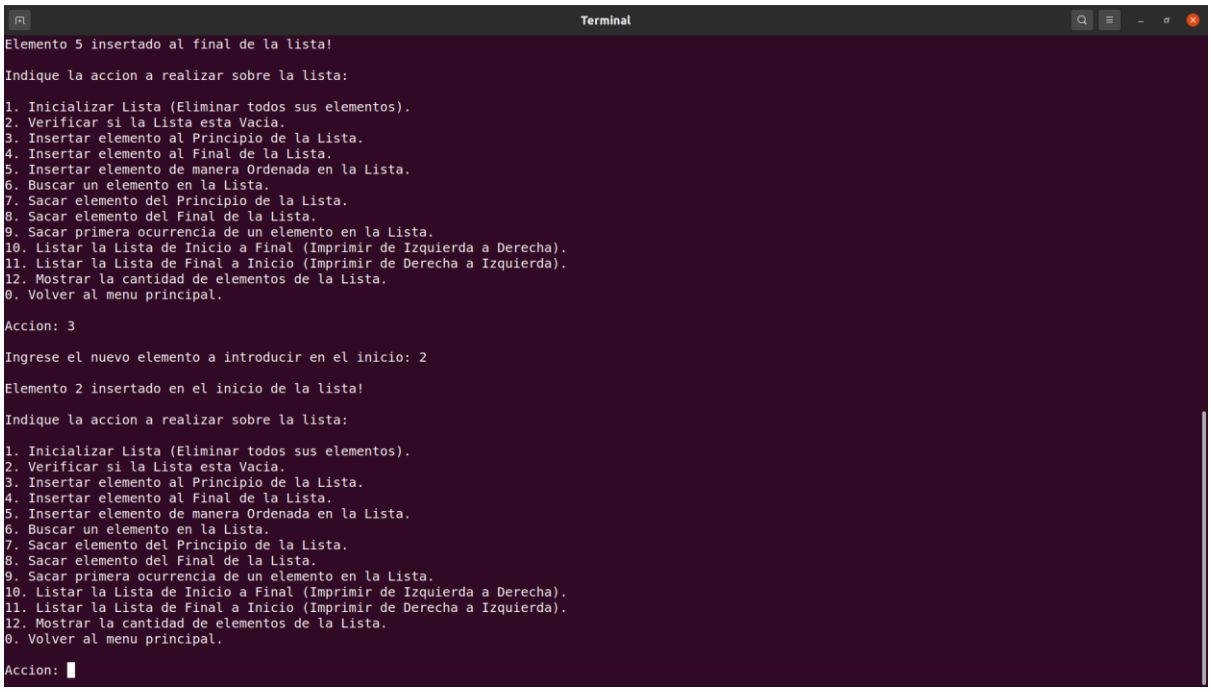
Verificar si la Lista esta Vacia:



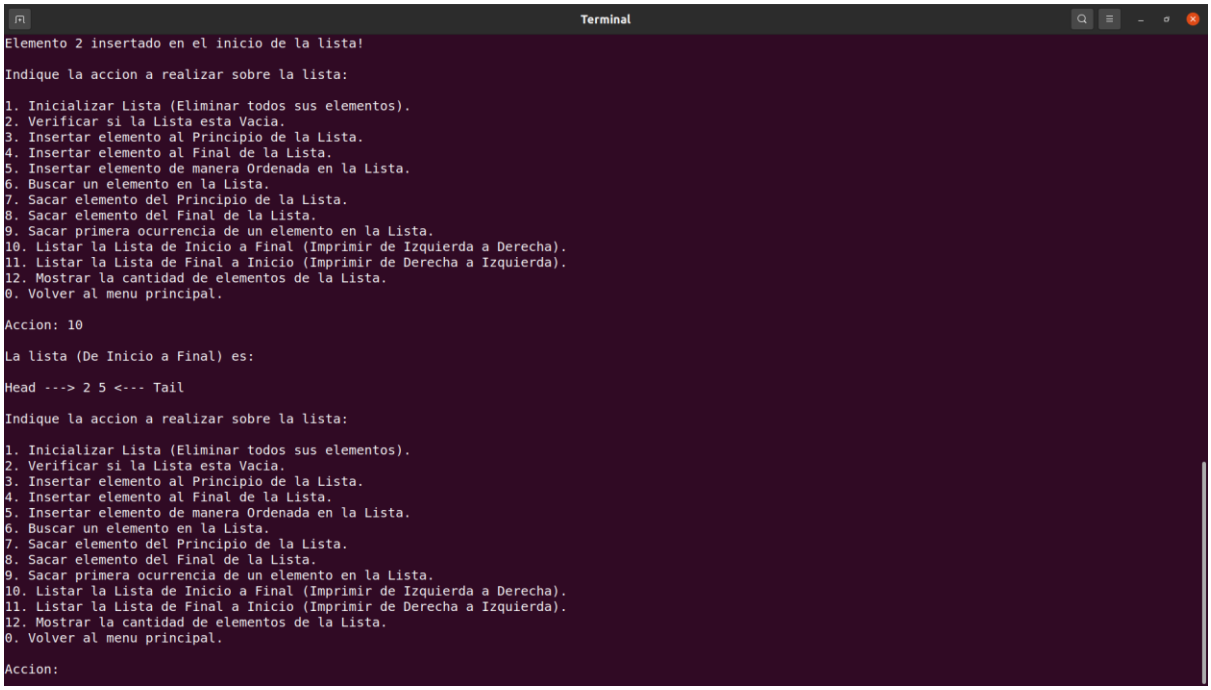
Insertar un elemento al final:



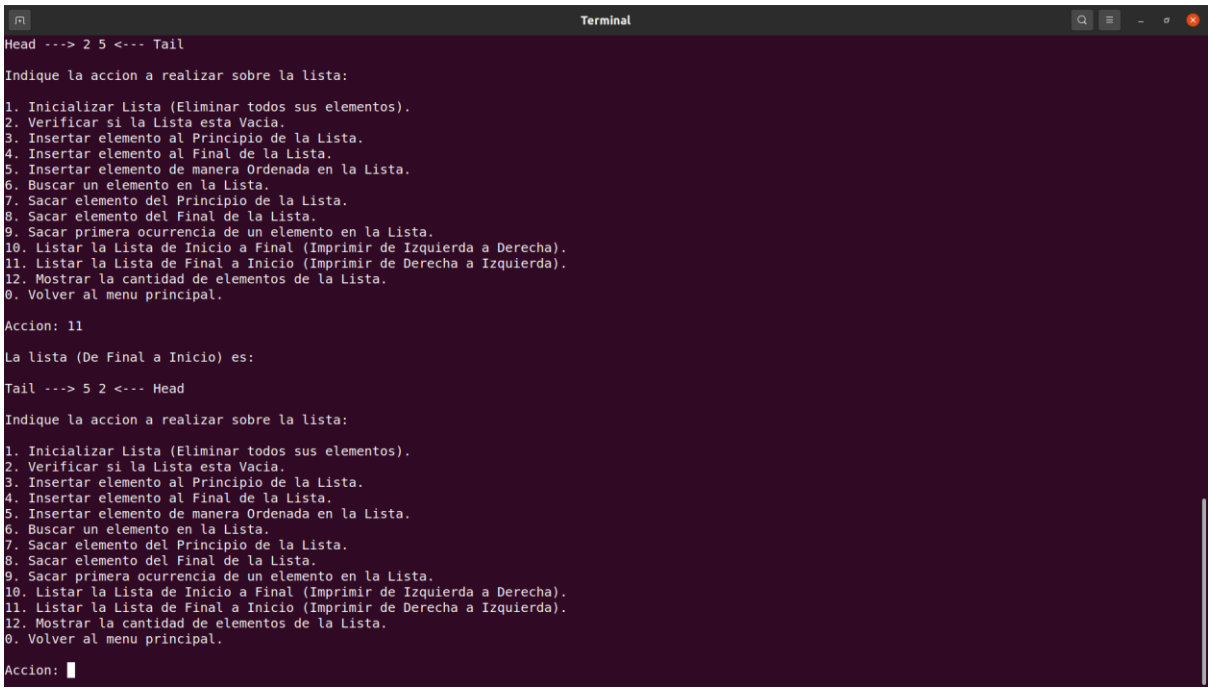
Insertar un elemento al inicio:



Imprimir la lista de inicio a fin:



Imprimir la lista de fin a inicio:



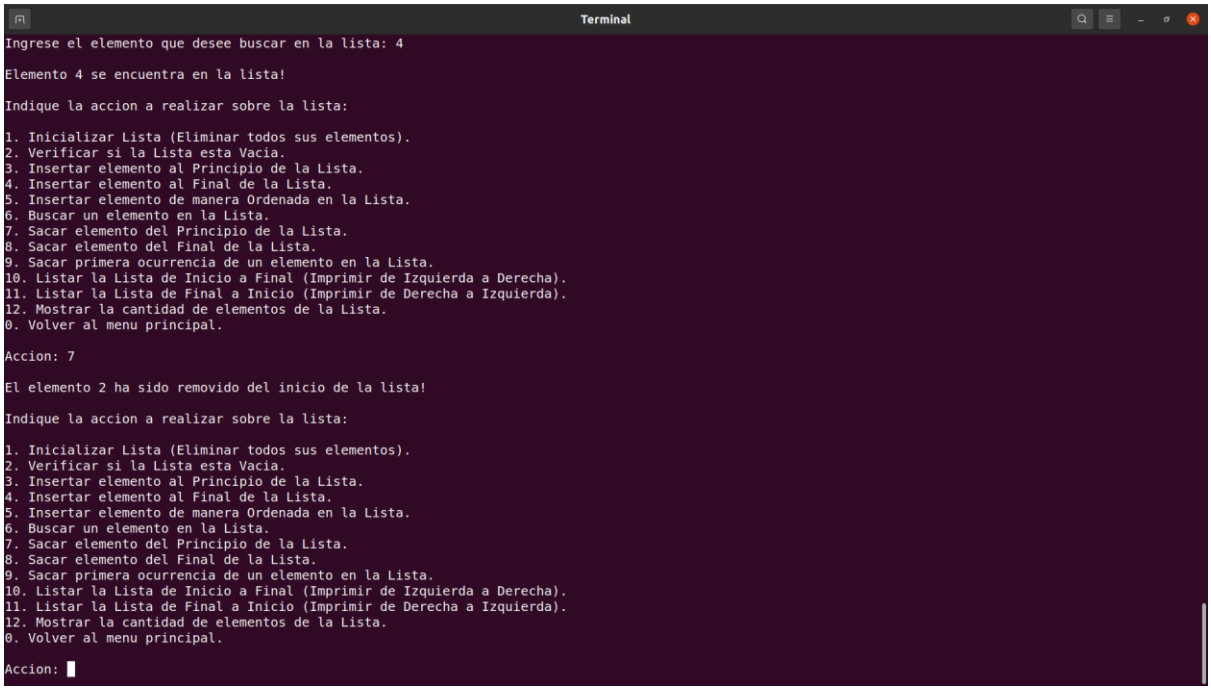
Insertar un elemento de forma ordenada (Para efectos de prueba, insertado en el medio):

```
La lista (De Inicio a Final) es:

Head ---> 2 4 5 <--- Tail

Indique la accion a realizar sobre la lista:

1. Inicializar Lista (Eliminar todos sus elementos).
2. Verificar si la Lista esta Vacia.
3. Insertar elemento al Principio de la Lista.
4. Insertar elemento al Final de la Lista.
5. Insertar elemento de manera Ordenada en la Lista.
6. Buscar un elemento en la Lista.
7. Sacar elemento del Principio de la Lista.
8. Sacar elemento del Final de la Lista.
9. Sacar primera ocurrencia de un elemento en la Lista.
10. Listar la Lista de Inicio a Final (Imprimir de Izquierda a Derecha).
11. Listar la Lista de Final a Inicio (Imprimir de Derecha a Izquierda).
12. Mostrar la cantidad de elementos de la Lista.
0. Volver al menu principal.

Accion:
```

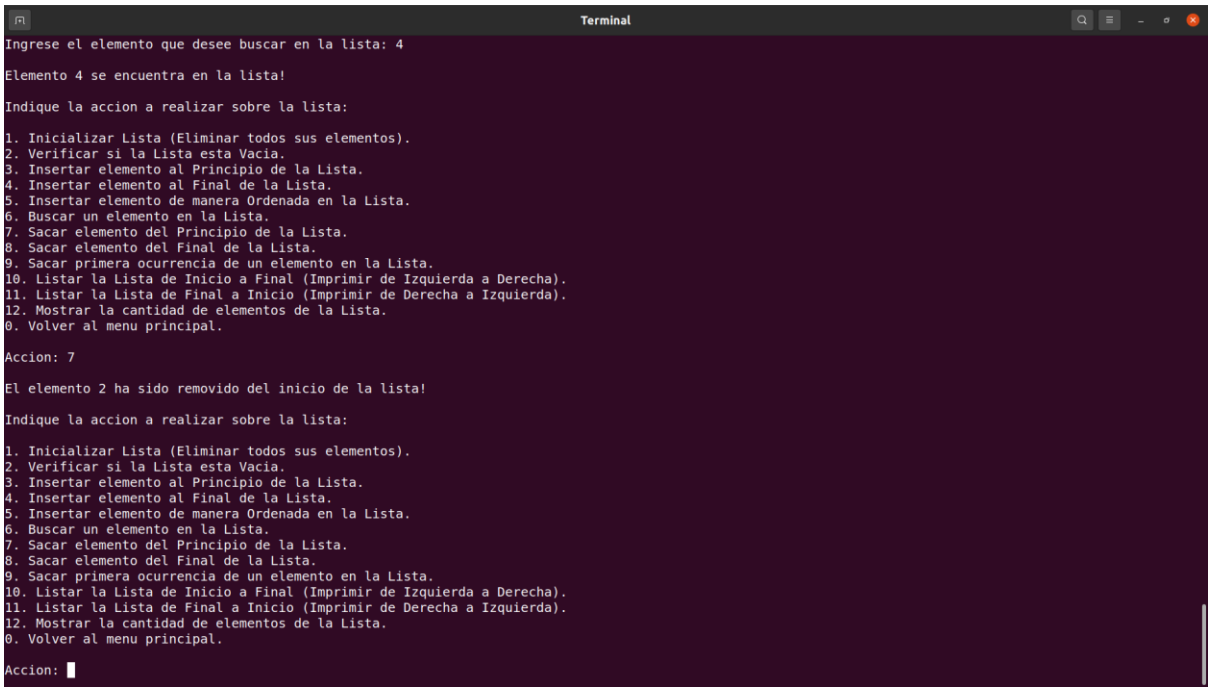
Buscar un elemento en la lista:

```
Terminal
Head ---> 2 4 5 <--- Tail

Indique la accion a realizar sobre la lista:

1. Inicializar Lista (Eliminar todos sus elementos).
2. Verificar si la Lista esta Vacia.
3. Insertar elemento al Principio de la Lista.
4. Insertar elemento al Final de la Lista.
5. Insertar elemento de manera Ordenada en la Lista.
6. Buscar un elemento en la Lista.
7. Sacar elemento del Principio de la Lista.
8. Sacar elemento del Final de la Lista.
9. Sacar primera ocurrencia de un elemento en la Lista.
10. Listar la Lista de Inicio a Final (Imprimir de Izquierda a Derecha).
11. Listar la Lista de Final a Inicio (Imprimir de Derecha a Izquierda).
12. Mostrar la cantidad de elementos de la Lista.
0. Volver al menu principal.

Accion: 6

Ingrese el elemento que desee buscar en la lista: 4

Elemento 4 se encuentra en la lista!

Indique la accion a realizar sobre la lista:

1. Inicializar Lista (Eliminar todos sus elementos).
2. Verificar si la Lista esta Vacia.
3. Insertar elemento al Principio de la Lista.
4. Insertar elemento al Final de la Lista.
5. Insertar elemento de manera Ordenada en la Lista.
6. Buscar un elemento en la Lista.
7. Sacar elemento del Principio de la Lista.
8. Sacar elemento del Final de la Lista.
9. Sacar primera ocurrencia de un elemento en la Lista.
10. Listar la Lista de Inicio a Final (Imprimir de Izquierda a Derecha).
11. Listar la Lista de Final a Inicio (Imprimir de Derecha a Izquierda).
12. Mostrar la cantidad de elementos de la Lista.
0. Volver al menu principal.

Accion:
```

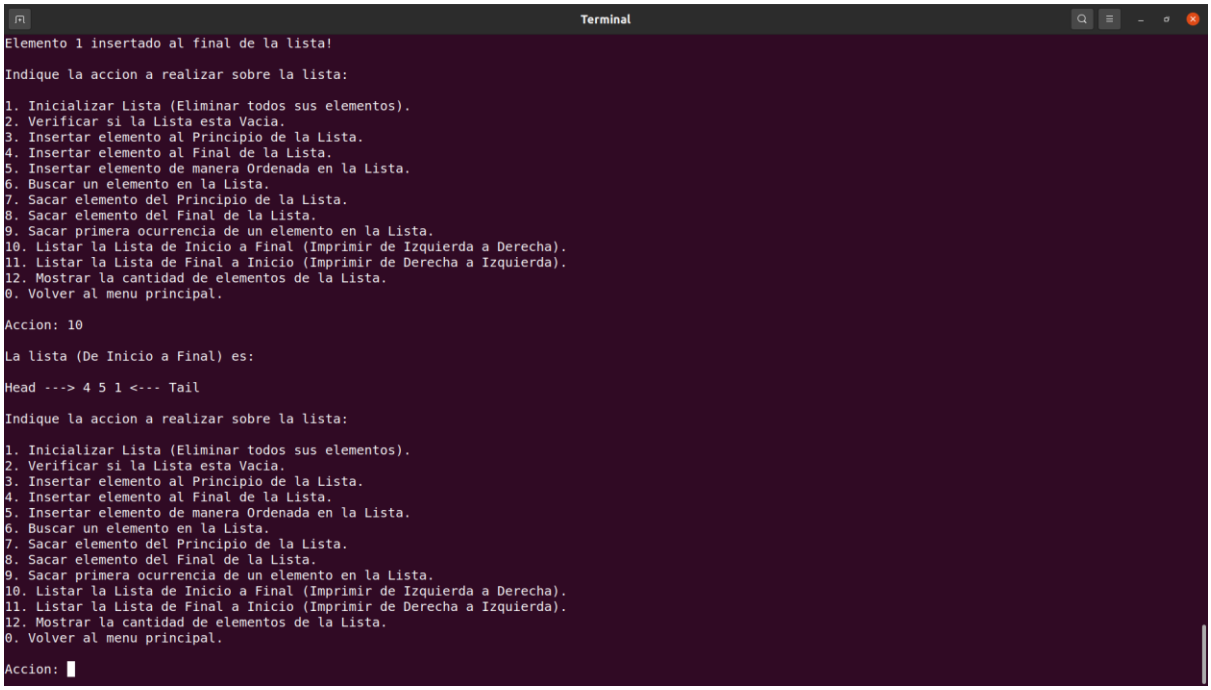
Sacar el primer elemento de la lista:

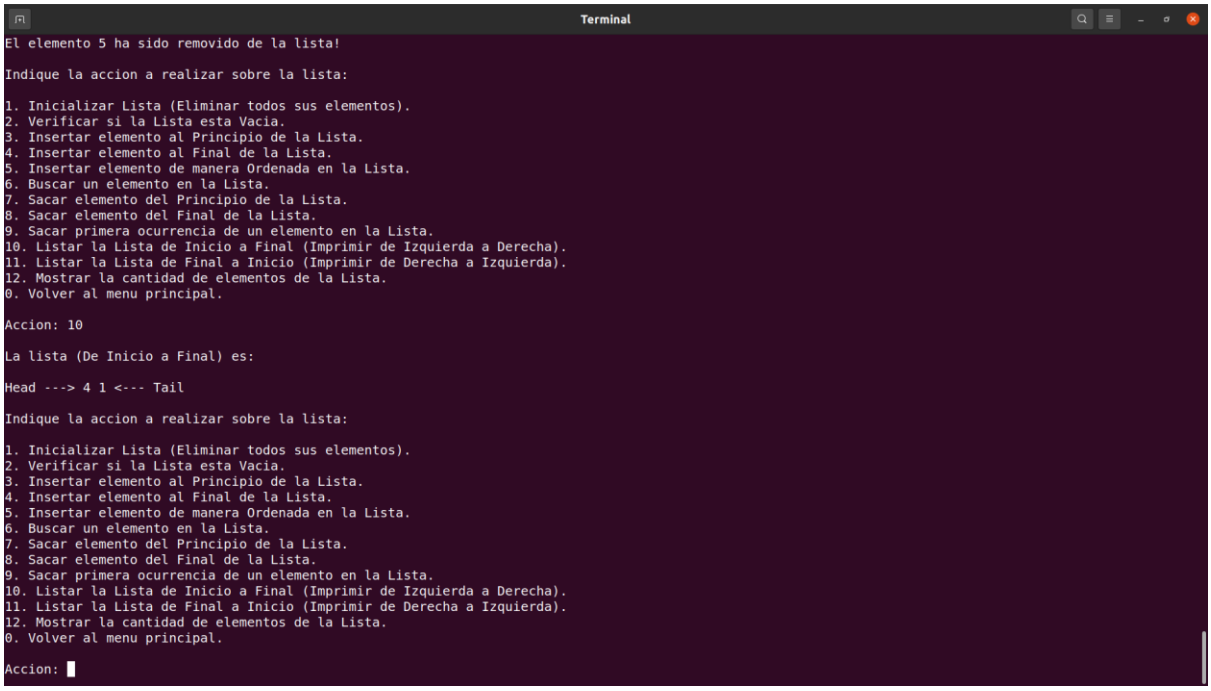


Sacar el ultimo elemento de la lista:

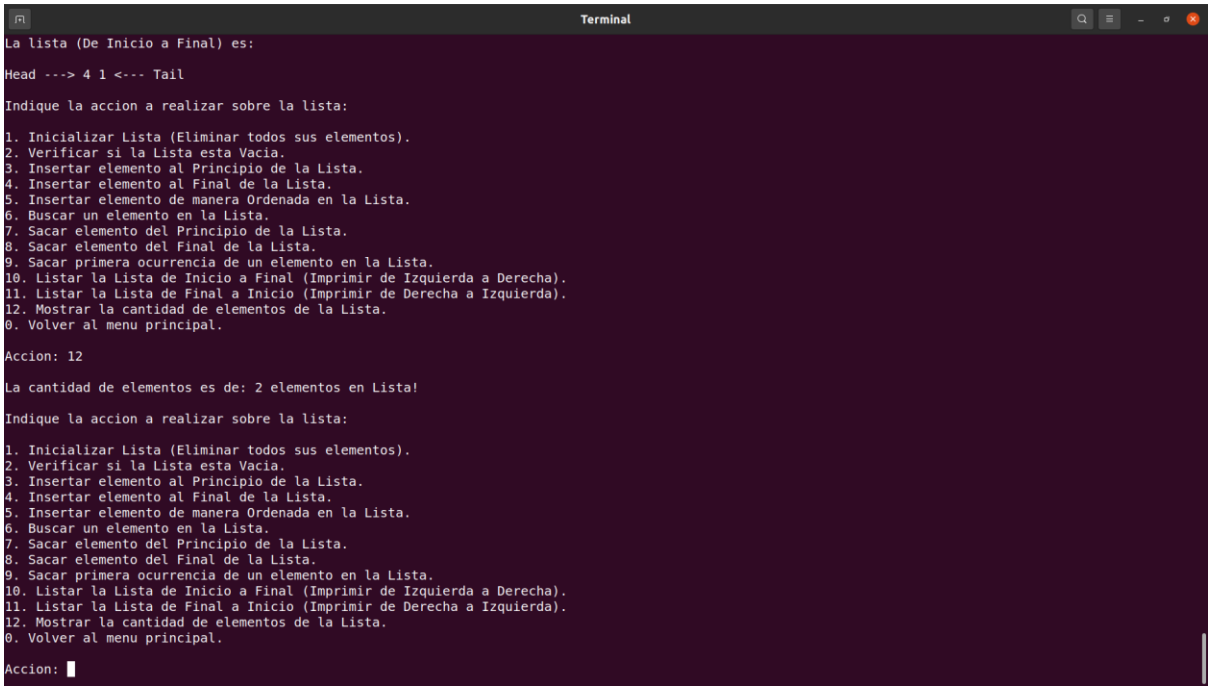


Agregar dos elementos mas (Insertar al Final) y Sacar primera ocurrencia de uno de esos elementos:

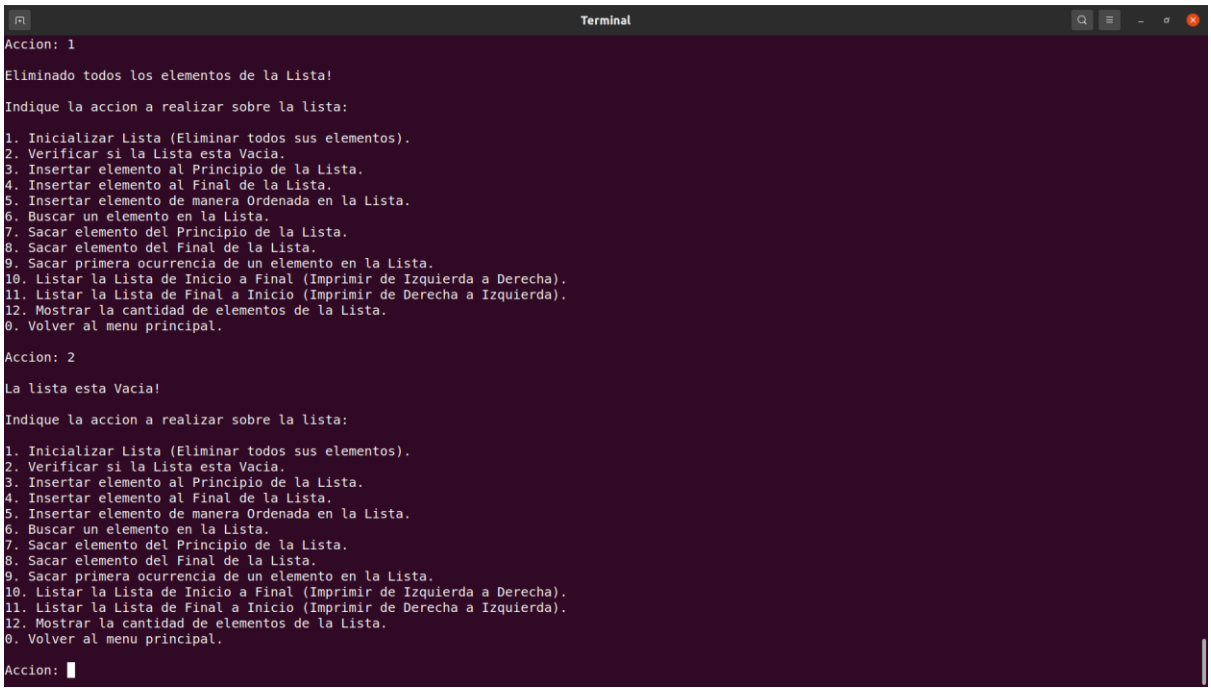




Cantidad de elementos de la lista:



Inicializar (Eliminar elementos) de la lista y mostrar que quedo vacia (esVacia):



Código Fuente:

/ * listaxor.h * /

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

```
// Definicion de las estructuras para la lista y los nodos

typedef struct Node
{
   int data;
   struct Node *prev_next;
}node;

typedef struct List
{
   node *head, *tail;
}Lista;

// Funcion para hacer XOR entre direcciones de punteros (Usando la libreria <stdint.h>)

node *XOR(node *x, node *y);

// Funcion (Nro 1) para Crear una nueva Lista

Lista *crearLista(Lista *L);

// Funcion (Nro 2) para Inicializar una Lista (Eliminar todos sus elementos)

void Inicializar(Lista *L);

// Funcion (Nro 3) para verificar si una Lista esta Vacia

int esVacia(Lista *L);

// Funcion (Nro 4) para insertar un elemento al Inicio de la Lista

int insertarPrincipio(Lista *L, int element);

// Funcion (Nro 5) para insertar un elemento al Final de la Lista

int insertarFinal(Lista *L, int element);

// Funcion (Nro 7) para buscar un elemento en la Lista

int buscar(Lista *L, int element);

// Funcion (Nro 8) para sacar el elemento del inicio en la Lista

int sacarPrincipio(Lista *L, int *element);

// Funcion (Nro 9) para sacar el elemento del final en la Lista

int sacarFinal(Lista *L, int *element);
```

```
// Funcion (Nro 11) para listar de Inicio a Final (Imprimir de Izquierda a Derecha)

void listarInicioAFinal(Lista *L);

// Funcion (Nro 12) para listar de Final a Inicio (Imprimir de Derecha a Izquierda)

void listarFinalAInicio(Lista *L);

// Funcion (Nro 13) para determinar la cantidad de elementos en la Lista.

int cantidadElementos(Lista *L);

// Funcion (Nro 14) para agregar un elemento en la lista en posicion especifica.

int insertarOrden(Lista *L, int element);

// Funcion (Nro 15) Remover de la lista la primera ocurrencia del elemento suministrado.

int sacarPrimeraOcurrencia(Lista *L, int element);

Lista *ordenarLista(Lista *L);
```

/ * listaxor.c * /

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "listaxor.h"

// Funcion para hacer XOR entre direcciones de punteros (Usando la libreria <stdint.h>)

node *XOR(node *x, node *y)
{
   node *tmp_x, *tmp_y;
   tmp_x = x;
   tmp_y = y;
   return (node*)((uintptr_t)(x) ^ (uintptr_t)(y));
}

// Funcion (Nro 1) para Crear una nueva Lista

Lista *crearLista(Lista *L)
{
   L = (Lista*)malloc(sizeof(Lista));

   // Inicializamos la Cabeza y Cola de la Lista
```

```
    L->head = NULL;
    L->tail = NULL;

    return L;   // Regresamos los cambios hechos a main
}

// Funcion (Nro 2) para Inicializar una Lista (Eliminar todos sus elementos)

void Inicializar(Lista *L)
{
    if (esVacia(L) == 1)
    {
        // Verificar si la lista esta vacia para evitar hacer operaciones innecesarias

        printf("La lista ya esta Vacia!\n");
        return;
    }
    else
    {
        int element;
        while (cantidadElementos(L) != 0)   // Mientras la lista siga teniendo elementos, eliminar el final
            sacarFinal(L, &element);
        L->head = L->tail = NULL;       // Asignar los punteros de Head y Tail a NULL
        return;
    }
}

// Funcion (Nro 3) para verificar si una Lista esta Vacia

int esVacia(Lista *L)
{
    // Si la Lista esta vacia (No tiene ningun elemento) entonces retornara un 1 a boolean en main()

    if (L->head == NULL && L->tail == NULL)
        return 1;
    else
        return 0;   // Si no esta vacia (Tiene minimo 1 un elemento), entonces retornara un 0 a boolean en main()
}

// Funcion (Nro 4) para insertar un elemento al Inicio de la Lista

int insertarPrincipio(Lista *L, int element)
{
    node *new_node, *ptr_next;
    new_node = (node*)malloc(sizeof(node));

    if (esVacia(L) == 1)
    {
```

```
    // Si la lista esta vacia, se crea un nuevo nodo y es asignado a Head y Tail de la lista

    new_node->data = element;
    new_node->prev_next = XOR(NULL, NULL);
    L->head = L->tail = new_node;

    return 1;
  }
  else if (esVacia(L) == 0)
  {
    new_node->data = element;   // Asignar el elemento al nodo
    new_node->prev_next = XOR(NULL, L->head);   // Asignarle la direccion en prev_next
    ptr_next = XOR(NULL, L->head->prev_next);   // Tener referencia del siguiente a Head
    L->head->prev_next = XOR(new_node, ptr_next);   // Asignar la direccion del nodo siguiente al nuevo como la siguiente de Head
    L->head = new_node;

    return 1;
  }
  else
    return 0;
}

// Funcion (Nro 5) para insertar un elemento al Final de la Lista

int insertarFinal(Lista *L, int element)
{
  node *new_node, *ptr_prev;
  new_node = (node*)malloc(sizeof(node));

  if (esVacia(L) == 1)
  {
    // Si la lista esta vacia, el nuevo nodo sera el unico elemento de la lista

    new_node->data = element;
    new_node->prev_next = XOR(NULL, NULL);
    L->head = L->tail = new_node;       // Tanto Head como Tail apuntaran al nuevo nodo

    return 1;
  }
  else if (esVacia(L) == 0)
  {
    new_node->data = element;
    new_node->prev_next = XOR(L->tail, NULL);   // Prev_next del nuevo nodo guardara la posicion de Tail
    ptr_prev = XOR(L->tail->prev_next, NULL);   // Guardar referencia del nodo anterior a Tail
    L->tail->prev_next = XOR(ptr_prev, new_node);  // Asignar la direccion del anterior al nuevo nodo
    L->tail = new_node;                 // La nueva Tail sera el nuevo nodo
```

```
      return 1;
   }
   else
      return 0;
}

// Funcion (Nro 7) para buscar un elemento en la Lista

int buscar(Lista *L, int element)
{
   if (esVacia(L) == 1)
      return 0;
   else
   {
      node * ptr_aux, *ptr_prev, *ptr_next;
      ptr_aux = L->head;  // Se comenzara la busqueda desde la Head
      ptr_prev = NULL;

      while (ptr_aux != NULL) // Mientras el elemento actual, no sea NULL, recorrer la lista
      {
         if (ptr_aux->data == element)   // Si un elemento en la lista coincide con el elemento a buscar, se retorna 1
            return 1;
         else
         {
            ptr_next = XOR(ptr_prev, ptr_aux->prev_next);   // Si no se consigue en ese nodo, iterar al siguiente
            ptr_prev = ptr_aux;
            ptr_aux = ptr_next;
         }
      }

      return 0;    // No se consigue el elemento tras iterar dentro de la lista, retornar 0
   }
}

// Funcion (Nro 8) para sacar el elemento del inicio en la Lista

int sacarPrincipio(Lista *L, int *element)
{
   if (esVacia(L) == 1)    // Si la lista es vacia, no se puede eliminar nada
      return 0;
   else
   {
      node *ptr_aux, *ptr_free, *ptr_prev, *ptr_next;
      ptr_prev = NULL;
      ptr_aux = L->head;

      if (L->head == L->tail) // Si la Head apunta al mismo nodo que la Tail, se elimina el unico nodo de la lista
      {
```

```
            *element = L->head->data;
            free(ptr_aux);
            L->head = NULL;
            L->tail = NULL;

            return 1;
        }
        else    // Se guarda la referencia al siguiente de la Head y se le asigna un puntero a Head
        {
            *element = L->head->data;
            ptr_next = XOR(ptr_prev, ptr_aux->prev_next);
            ptr_free = ptr_aux;
            ptr_aux = ptr_next; // Se mueve el puntero auxiliar al siguiente de Head
            ptr_prev = ptr_free;
            L->head = ptr_aux;  // Se asigna la nueva Head
            ptr_aux->prev_next = XOR(ptr_prev, L->head->prev_next); // Se actualiza el prev_next de la nueva Head
            ptr_prev = NULL;
            free(ptr_free);     // Liberar el nodo guardado en la anterior Head

            return 1;
        }
    }
}

// Funcion (Nro 9) para sacar el elemento del final en la Lista

int sacarFinal(Lista *L, int *element)
{
    if (esVacia(L) == 1)    // Si la lista esta vacia, no se puede hacer nada
        return 0;
    else
    {
        node *ptr_aux, *ptr_free, *ptr_prev, *ptr_next;
        ptr_next = NULL;
        ptr_aux = L->tail;

        if (L->head == L->tail) // Si la Head apunta al mismo nodo que Tail, se elimina el unico nodo de la lista
        {
            *element = L->head->data;
            free(ptr_aux);
            L->head = NULL;
            L->tail = NULL;

            return 1;
        }
        else
        {
            *element = L->tail->data;
```

```
            ptr_prev = XOR(L->tail->prev_next, ptr_next);   // Guardar la referencia del anterior a Tail
            ptr_free = ptr_aux;     // Asignar un puntero a Tail
            ptr_aux = ptr_prev; // Mover el puntero original al anterior a Tail
            L->tail = ptr_aux;  // Reasignar Tail
            ptr_next = ptr_free;
            ptr_aux->prev_next = XOR(L->tail->prev_next, ptr_next); // Ajustar el prev_next del anterior al antiguo Tail
            ptr_next = NULL;
            free(ptr_free);     // Eliminar el nodo que era la anterior Tail

            return 1;
        }
    }
}

// Funcion (Nro 11) para listar de Inicio a Final (Imprimir de Izquierda a Derecha)

void listarInicioAFinal(Lista *L)
{
    if (esVacia(L) == 1)    // Si la lista esta vacia, no se puede imprimir nada
    {
        printf("La lista es vacia! No tiene elementos para listar...\n");
        return;
    }
    else
    {
        node *ptr_aux, *prev, *next;
        prev = NULL;
        printf("Head ---> ");
        ptr_aux = L->head;
    int count;
    count = 0;
        while (ptr_aux != NULL) // Imprimir elemento a elemento de la lista mientras que el puntero auxiliar != NULL
        {
            /* printf("%d-", count); */
            printf("%d ", ptr_aux->data);
            next = XOR(prev, ptr_aux->prev_next);   // Iterar de derecha a izquierda
            prev = ptr_aux;
            ptr_aux = next;
        count++;
        }
        printf("<--- Tail\n");
    }
}

// Funcion (Nro 12) para listar de Final a Inicio (Imprimir de Derecha a Izquierda)

void listarFinalAInicio(Lista *L)
{
```

```
    if (esVacia(L) == 1)    // Si la lista esta vacia, no se puede imprimir nada
    {
        printf("La lista es vacia! No tiene elementos para listar...\n");
        return;
    }
    else
    {
        node *ptr_aux, *prev, *next;
        ptr_aux = L->tail;
        next = NULL;
        printf("Tail ---> ");

        while (ptr_aux != NULL) // Imprimir elemento a elemento de la lista mientras que el puntero auxiliar != NULL
        {
            printf("%d ", ptr_aux->data);
            prev = XOR(next, ptr_aux->prev_next);
            next = ptr_aux;
            ptr_aux = prev;
        }
        printf("<--- Head\n");
    }
}

// Funcion (Nro 13) para determinar la cantidad de elementos en la Lista.

int cantidadElementos(Lista *L)
{
    int element_count = 0;
    node *ptr_aux, *ptr_prev, *ptr_next;
    ptr_aux = L->head;
    ptr_prev = NULL;

    if (esVacia(L) == 1)    // Si es vacia, entonces tiene 0 elementos
        return element_count;
    else
    {
        while (ptr_aux != NULL) // Recorrer la lista y usar un contador elemento por elemento
        {
            element_count += 1;
            ptr_next = XOR(ptr_prev, ptr_aux->prev_next);
            ptr_prev = ptr_aux;
            ptr_aux = ptr_next;
        }

        return element_count;
    }
}
```

```
// Verificar si los elementos de la lista estan ordenados

int estaOrdenada(Lista *L) {
    node *ptr_aux, *ptr_prev, *ptr_next;
    ptr_aux = L->head;
    ptr_prev = NULL;

    if (esVacia(L) == 1)    // Si es vacia, entonces tiene 0 elementos
        return 1;
    else
    {
        while (ptr_aux != NULL)
        {
        if (ptr_prev != NULL && ptr_aux != NULL && ptr_prev->data > ptr_aux->data)
        return 0;
            ptr_next = XOR(ptr_prev, ptr_aux->prev_next);
            ptr_prev = ptr_aux;
            ptr_aux = ptr_next;
        }

        return 1;
    }
}

int obtenerSiguiente(Lista *L, int element) {
    node *ptr_aux, *ptr_prev, *ptr_next;
    ptr_aux = L->head;
    ptr_prev = NULL;
    int menor;
    menor = 0;

    while (ptr_aux != NULL) {
    if (menor == 0 && element < ptr_aux->data)
        menor = ptr_aux->data;
    else if (ptr_aux->data < menor && element < ptr_aux->data)
        menor = ptr_aux->data;
    ptr_next = XOR(ptr_prev, ptr_aux->prev_next);
    ptr_prev = ptr_aux;
    ptr_aux = ptr_next;
    }
    return menor;
}

Lista *ordenarLista(Lista *L) {
    int last;
    last = 0;

    Lista *new_list;
```

```
    new_list = crearLista(new_list);

    for (int i=0; i<cantidadElementos(L); i++) {
    last = obtenerSiguiente(L, last);
    insertarFinal(new_list, last);
    }

    return new_list;
}

// Funcion (Nro 14) para agregar un elemento en la lista en posicion ordenada.

int insertarOrden(Lista *L, int element) {
    node *ptr_aux, *ptr_prev, *ptr_next;
    ptr_aux = L->head;
    ptr_prev = NULL;

    if (esVacia(L) == 1)
    return insertarPrincipio(L, element);

    while (ptr_aux != NULL)
    {
    ptr_next = XOR(ptr_prev, ptr_aux->prev_next);
    if (ptr_next == NULL)  // Insertar al final de la lista
        return insertarFinal(L, element);
    if (ptr_next->data > element) {  // Insertar antes de primera ocurrencia mayor
        if (ptr_prev == NULL)  // Insertar al principio de la lista
        return insertarPrincipio(L, element);

        node *new_node;  // Insertar entre dos elementos
        new_node = (node*)malloc(sizeof(node));
        new_node->data = element;   // Asignar el elemento al nodo
        new_node->prev_next = XOR(ptr_aux, ptr_next);   // Asignarle la direccion en prev_next

        // Ajustar enlace del elemento anterior
        ptr_aux->prev_next = XOR(ptr_prev, new_node);

        // Ajustar enlace del elemento siguiente
        node *ptr_next_t_next;
        ptr_next_t_next = XOR(ptr_aux, ptr_next->prev_next);
        ptr_next->prev_next = XOR(new_node, ptr_next_t_next);

        return 1;
    }
    ptr_prev = ptr_aux;
    ptr_aux = ptr_next;
    }
```

```
    return 0;
}

int sacarPrimeraOcurrencia(Lista *L, int element) {
    node *ptr_aux, *ptr_prev, *ptr_next, *ptr_free;
    ptr_aux = L->head;
    ptr_prev = NULL;
    int aux_element;

    if (esVacia(L) == 1)
    return 1;

    while (ptr_aux != NULL)
    {
    ptr_next = XOR(ptr_prev, ptr_aux->prev_next);
    if (ptr_aux->data == element && ptr_next == NULL)
        return sacarFinal(L, &aux_element);
    if (ptr_aux->data == element) {
        if (ptr_prev == NULL)
        return sacarPrincipio(L, &aux_element);

        // Ajustar enlace del elemento anterior
        node *ptr_prev_t_prev;
        ptr_prev_t_prev = XOR(ptr_aux, ptr_prev->prev_next);
        ptr_prev->prev_next = XOR(ptr_prev_t_prev, ptr_next);

        // Ajustar enlace del elemento siguiente
        node *ptr_next_t_next;
        ptr_next_t_next = XOR(ptr_aux, ptr_next->prev_next);
        ptr_next->prev_next = XOR(ptr_prev, ptr_next_t_next);

            ptr_free = ptr_aux;
            free(ptr_free);     // Liberar el nodo guardado en la anterior Head

        return 1;
    }
    ptr_prev = ptr_aux;
    ptr_aux = ptr_next;
    }

    return 0;
}
```

/ * proy2.c * /

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

```
#include "listaxor.h"

int main(void)
{
   int action, boolean, new_element, element_num;

   action = -1;

   printf("\nProyecto 2: TAD Lista XOR ~ Presentado por: Angel Patino y Leonardo Gonzalez\n");

   while (action != 0)
   {
      printf("\nIndique la accion a realizar:\n\n");
      printf("1. Crear Lista\n");
      printf("0. Cerrar la Aplicacion.\n\n");
      printf("Accion: ");
      scanf("%d", &action);

      if (action == 1)
      {
         Lista *list;
         list = crearLista(list);
         printf("\nUna nueva lista se ha creado!\n");

         while (action != 0)
         {
            printf("\nIndique la accion a realizar sobre la lista: \n\n");
            printf("1. Inicializar Lista (Eliminar todos sus elementos).\n");
            printf("2. Verificar si la Lista esta Vacia.\n");
            printf("3. Insertar elemento al Principio de la Lista.\n");
            printf("4. Insertar elemento al Final de la Lista.\n");
            printf("5. Insertar elemento de manera Ordenada en la Lista.\n");
            printf("6. Buscar un elemento en la Lista.\n");
            printf("7. Sacar elemento del Principio de la Lista.\n");
            printf("8. Sacar elemento del Final de la Lista.\n");
            printf("9. Sacar primera ocurrencia de un elemento en la Lista.\n");
            printf("10. Listar la Lista de Inicio a Final (Imprimir de Izquierda a Derecha).\n");
            printf("11. Listar la Lista de Final a Inicio (Imprimir de Derecha a Izquierda).\n");
            printf("12. Mostrar la cantidad de elementos de la Lista.\n");
            printf("0. Volver al menu principal.\n\n");
            printf("Accion: ");
            scanf("%d", &action);

            if (action == 1)
            {
               Inicializar(list);
               printf("\nEliminado todos los elementos de la Lista!\n");
            }
```

```
    else if (action == 2)
    {
        boolean = esVacia(list);
        if (boolean == 0)
            printf("\nLa lista NO esta Vacia!\n");
        else
            printf("\nLa lista esta Vacia!\n");
    }
    else if (action == 3)
    {
        printf("\nIngrese el nuevo elemento a introducir en el inicio: ");
        scanf("%d", &new_element);
        boolean = insertarPrincipio(list, new_element);
        if (boolean == 0)
            printf("\nError: No se pudo insertar el elemento %d!\n", new_element);
        else
            printf("\nElemento %d insertado en el inicio de la lista!\n", new_element);
    }
    else if (action == 4)
    {
        printf("\nIngrese el nuevo elemento a introducir al final: ");
        scanf("%d", &new_element);
        boolean = insertarFinal(list, new_element);
        if (boolean == 0)
            printf("\nError: No se pudo insertar el elemento %d!\n", new_element);
        else
            printf("\nElemento %d insertado al final de la lista!\n", new_element);
    }
    else if (action == 5)
    {
list = ordenarLista(list);
printf("\nIngrese el nuevo elemento a introducir de manera ordenada: ");
        scanf("%d", &new_element);
boolean = insertarOrden(list, new_element);
        if (boolean == 0)
            printf("\nError: No se pudo insertar el elemento %d!\n", new_element);
        else
            printf("\nElemento %d insertado en la lista!\n", new_element);
    }
    else if (action == 6)
    {
        printf("\nIngrese el elemento que desee buscar en la lista: ");
        scanf("%d", &new_element);
        boolean = buscar(list, new_element);
        if (boolean == 0)
            printf("\nEl elemento %d NO se encuentra en la lista!\n", new_element);
        else
            printf("\nElemento %d se encuentra en la lista!\n", new_element);
```

```
    }
    else if (action == 7)
    {
        int pop_element;
        boolean = sacarPrincipio(list, &pop_element);
        if (boolean == 0)
            printf("\nEl elemento del inicio no se ha podida eliminar, la lista esta vacia!\n");
        else
            printf("\nEl elemento %d ha sido removido del inicio de la lista!\n", pop_element);
    }
    else if (action == 8)
    {
        int pop_element;
        boolean = sacarFinal(list, &pop_element);
        if (boolean == 0)
            printf("\nEl elemento del final no se ha podida eliminar, la lista esta vacia!\n");
        else
            printf("\nEl elemento %d ha sido removido del final de la lista!\n", pop_element);
    }
    else if (action == 9)
    {
        printf("\nIngrese el elemento que desee remover de la lista: ");
        scanf("%d", &new_element);
        boolean = sacarPrimeraOcurrencia(list, new_element);
        if (boolean == 0)
            printf("\nEl elemento no se ha podida eliminar, la lista esta vacia!\n");
        else
            printf("\nEl elemento %d ha sido removido de la lista!\n", new_element);
    }
    else if (action == 10)
    {
        printf("\nLa lista (De Inicio a Final) es:\n\n");
        listarInicioAFinal(list);
    }
    else if (action == 11)
    {
        printf("\nLa lista (De Final a Inicio) es:\n\n");
        listarFinalAInicio(list);
    }
    else if (action == 12)
    {
        element_num = cantidadElementos(list);
        printf("\nLa cantidad de elementos es de: %d elementos en Lista!\n", element_num);
    }

boolean = -1;

}
```

```
            action = -1;
        }
    }

    printf("\nCerrando aplicacion...\n\n");
}
```

/ * makefile * /

```
all: listaxor.o proy2.o
	gcc -g listaxor.o proy2.o -o proy2

listaxor.o: listaxor.c
	gcc -c listaxor.c -o listaxor.o

proy2.o: proy2.c
	gcc -c proy2.c -o proy2.o
```

Enlace al repositorio de GitHub del Proyecto:

https://github.com/Angel5112/Proyecto-TAD-Lista-Xor

ó

https://github.com/Angel5112/Proyecto-TAD-Lista-Xor.git