| | Carátula para entrega de prácticas |
|---|---|
| Facultad de Ingeniería | Laboratorio de docencia |

# Laboratorios de computación salas A y B

*Profesor:* MI. Edgar Tista García

*Asignatura:* Estructura de Datos y Algoritmos I

*Grupo:* 4

*No de Práctica(s):* 7-8

*Integrante(s):* León Sánchez José Alejandro

*No. de Equipo de cómputo empleado:* -

*No. de Lista o Brigada:* -

*Semestre:* 2020-2

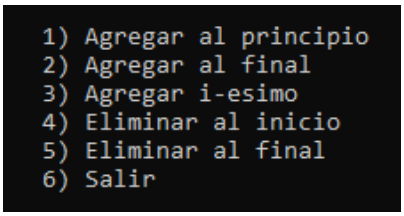
*Fecha de entrega:* 13 de junio del 2020

*Observaciones:* -

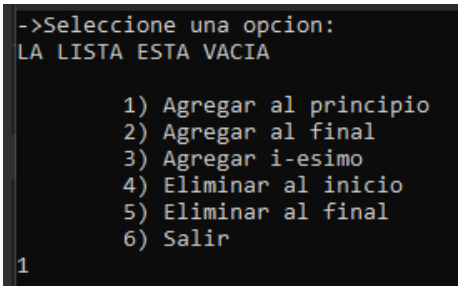
CALIFICACIÓN: _____

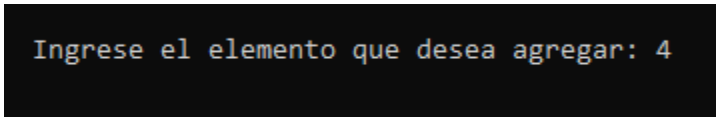
## Actividad 1

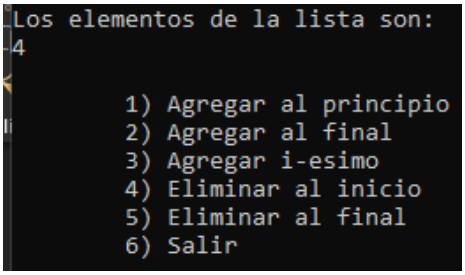
a) Esta sección requirió un programa para crear una lista de elementos y verificar las funciones de la lista, el menú contiene las siguientes opciones:



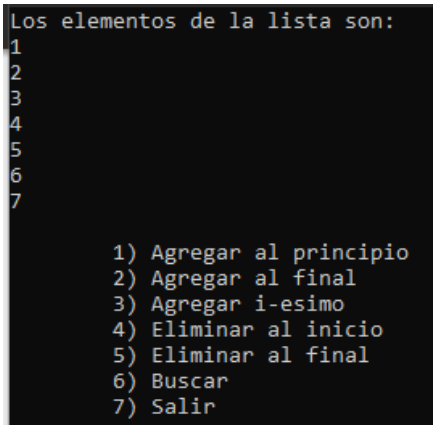

Para crear el menú usé un ciclo *do while* que permite regresar al menú luego de ejecutar una de las opciones infinitamente hasta que se seleccione la opción salir, la pantalla se limpia en cada selección para no saturarse e imprime el estado de la lista antes de seleccionar una opción, como sigue:

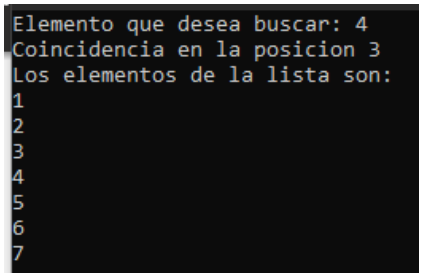


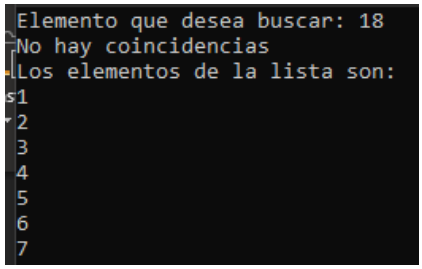




b) En este inciso se pidió escribir una función para buscar un elemento dentro de la lista, dentro de la biblioteca escribí una función que con un nodo temporal avanza a través de los elementos de la lista hasta encontrar coincidencias. Devuelve la posición en la que encontró dichas coincidencias y si no las encontró imprime un mensaje. Considere la siguiente ejecución como ejemplo:



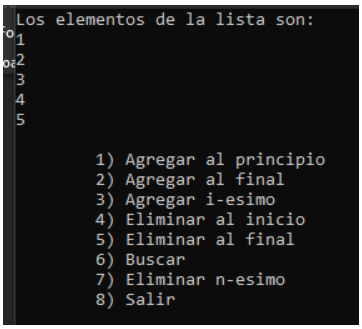



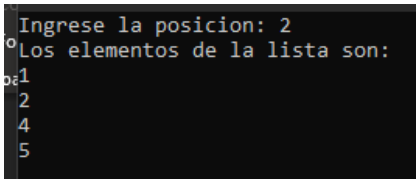

c) En este inciso se pidió agregar una nueva función al menú que eliminara un elemento n-ésimo indicado por el usuario. En esta función es necesario recorrer el listado con un nodo temporal hasta llegar a la posición indicada y borrar el elemento cuidando de las referencias de los elementos adyacentes. La estructura de la función queda como sigue:

```
if (lista->head == NULL) {
    printf("La lista esta vacia");
    return;
}
else{
    for(int c=0;c<pos-1;c++){
        tmp=tmp->next;
    }
    tmp2=tmp->next;
    tmp->next=tmp2->next;
    free(tmp2);
    return;
}
```

Obsérvese que se usa un segundo nodo temporal para poder “brincar” al elemento que se desea eliminar, una vez que se actualiza la referencia next del elemento anterior al que

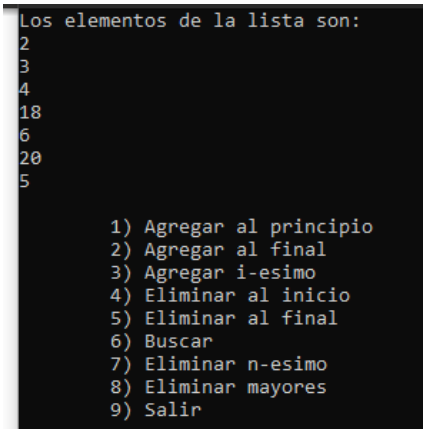
se desea eliminar, se libera el espacio del nodo. Considere la siguiente ejecución:




Nótese que se considera al primer elemento como el elemento 0, por lo que al seleccionar el índice 2 se eliminó el elemento cuyo contenido era un número 3.

d) Para esta sección se requirió un programa que eliminara los elementos de la lista que fueran mayores a un número dado por el usuario. Para solucionar este problema se me ocurrió crear dos listas extra e iterando la lista principal se examinan los valores de los elementos de la lista (con ayuda de un nodo temporal). En una de las listas se van poniendo los valores que se van a eliminar y en otra los que se van a quedar, si la lista de "trash" está vacía significa que no se eliminaron elementos, por lo tanto se despliega un mensaje, finalmente; si hubo elementos eliminados, se despliega la lista que los contiene. Considere la siguiente ejecución:



```
Se eliminaran los numeros mayores a: 5
Los elementos eliminados fueron:
18
6
20
Los elementos de la lista son:
2
3
4
5
```

Y si ahora volvemos a eliminar los mayores a 5:

```
Se eliminaran los numeros mayores a: 5
NO HUBO ELEMENTOS MAYORES
Los elementos de la lista son:
2
3
4
5
```

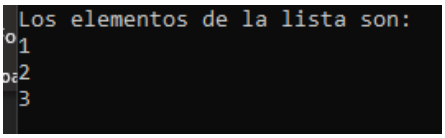
e) Considero que lo más difícil de esta actividad es la lógica detrás de cada función, es útil anotar los requerimientos del ejercicio y plasmar algunas ideas en papel, así es más claro lo que se debe hacer en el código. Especialmente el último ejercicio me resultó complicado ya que es una función que requiere de varios condicionales, aunque a decir verdad no fue tan complejo como parece, sino más bien largo.

## Actividad 2

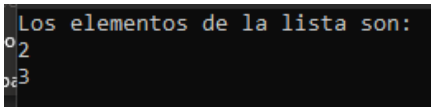
a) En esta sección se pidió un programa que comprobara el funcionamiento de la biblioteca “listacirc.h”, la cual contenía un par de errores, la función “borrarUltimo” no funcionaba y en ambas funciones de eliminación no se decrementaba el tamaño de la lista (me costó horas encontrar este problema) y por lo tanto al imprimir la lista salían otros valores aleatorios almacenados en memoria (basura). Las funciones para agregar elementos funcionaban bien. El programa que yo elaboré fue un menú al igual que la actividad anterior. Considere la siguiente ejecución:

```
Los elementos de la lista son:
1
2
3
4


        1) Agregar al principio
        2) Agregar al final
        3) Eliminar al inicio
        4) Eliminar al final
        5) Recorrer lista
        6) Salir
```

Al eliminar al final:




Al eliminar al inicio:



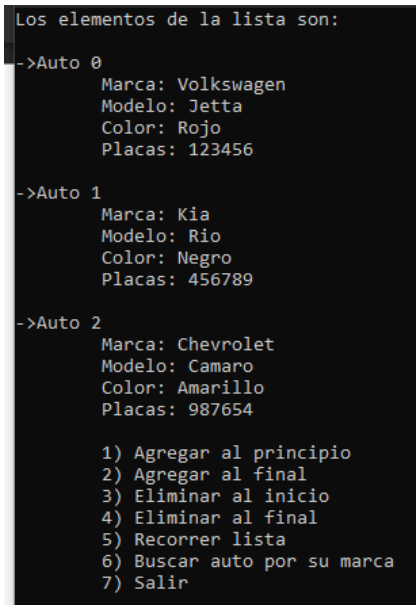

b) En este inciso fue necesario diseñar un TDA para modelar un auto, para esto es necesario cambiar gran parte del código, desde cómo se ingresan los datos hasta como se agregan y quitan elementos, dado que el TDA tendrá mas elementos que un simple número entero. A mi TDA agregué marca, modelo, placas y color, todos de tipo char. El código de la biblioteca que maneja los autos queda de la siguiente forma: listacirc_auto.h y el programa principal: 2_auto.c .

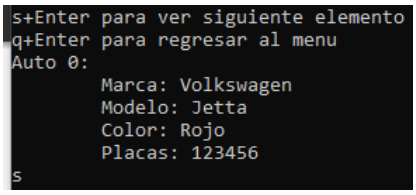
c) En esta sección se requirió modificar la biblioteca para que se pudiera realizar una búsqueda de autos a través de su marca, para lo cual fue necesario crear una función en la cual se crea otra lista que almacenará las coincidencias, como sigue:

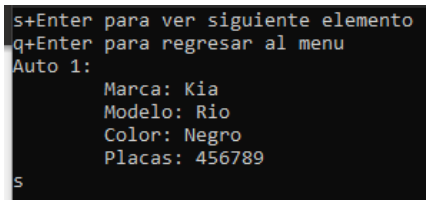
```
void buscarAuto(Lista *lista){
    system("cls");
    char marca[20];
    Nodo *tmp = lista->head;
    Lista encontrados=crearLista();
    printf("Ingrese la marca a buscar: ");
    scanf("%s",&marca);
    for (int i=0; i<lista->tamano; i++){
        tmp=tmp->next;
        if (strcmp(marca,tmp->carro.marca)==0){
            addFinalLista(&encontrados,tmp->carro);
        }
    }
    printf("COINCIDENCIAS: ");
    print_list(encontrados);
    system("timeout 10");
    system ("cls");
    return;
}
```

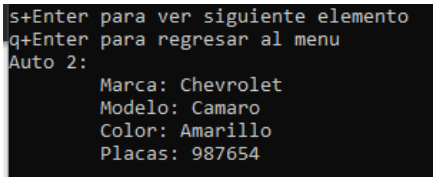
La función strcmp es una comparación de strings, si la comparación resulta verdadera devuelve un 0, entonces con un ciclo for itera sobre el contenido de la lista y si va encontrando coincidencias las almacena en la nueva lista, para luego mostrar esta última durante 10 segundos y luego volver al menú principal.

d) Para este inciso fue necesario crear una función que permitiera moverse a través de los elementos de la lista, yo diseñé una función con un while infinito que sale solo cuando se presiona la tecla “q”, una vez que el usuario presiona “s” se muestra el siguiente auto, con su respectivo índice dentro de la lista, cabe resaltar que la lista es circular, por lo que al llegar al último elemento y dar siguiente, se vuelve a desplegar el primer elemento. Considere la siguiente ejecución:




Y si se vuelve a ingresar s+Enter se mostraría el primer elemento (Auto 0) de nueva cuenta.

## Actividad 3

a) En este inciso se requiere encontrar por qué no se muestra el primer elemento de la lista en el programa P8programa.c. El erro se encontraba en la función print_list, donde se imprimía el siguiente del nodo current, y dado que dicho nodo empieza siendo igual a head, nunca se mostraba el primer elemento. El código era:

```
void print_list(Lista lista) {
    if(lista.head==NULL){
        printf("LA LISTA ESTA VACIA \n");
    }
    else{
        printf("Los elementos de la lista son: \n")
        Nodo *current = lista.head;
        while (current != NULL){
            printf("%d\n",current->next->val);
            current = current->next;
        }
    }
}
```

La parte resaltada es la línea donde se encontraba el error, la corrección queda:

```
void print_list(Lista lista) {
    if(lista.head==NULL){
        printf("LA LISTA ESTA VACIA \n");
    }
    else{
        printf("Los elementos de la lista son: \n")
        Nodo *current = lista.head;
        while (current != NULL){
            printf("%d\n",current->val);
            current = current->next;
        }
    }
}
```

Y de esta forma ya se muestra el primer elemento de la lista.

b) Las diferencias entre las funciones print_list y print_list2 es la forma en que se imprime la lista, print_list lo hace comenzando por el primer elemento y hasta el último, y prnt_list2 lo hace al revés, empezando por el último elemento hasta llegar al primero, considere las siguientes ejecuciones:

```
Bienvenido al programa
Los elementos de la lista son:
10
20
30
40
50
60
```

```
Bienvenido al programa
Los elementos de la lista son:
60
50
40
30
20
10
```

La imagen de la izquierda muestra la impresión con print_list y la de la derecha la impresión con print_list2.

Ambas funciones están diseñadas con un ciclo while, la diferencia radica en el uso de “prev” en la función print_list2, lo que hace la función es primero ir hasta el final de la lista con el nodo current y luego regresar, esta vez imprimiendo los valores almacenados en los nodos de la lista.

c) La función primer nodo sirve para agregar un nodo cuando la lista está vacía. Si una lista está vacía se usa el mismo procedimiento, aunque se indique agregar al inicio o agregar al final, entonces las funciones addPrincipio y addFinal primero revisan que la lista no esté vacía, y si lo está llaman a la función primerNodo para que agregue el primer elemento.

d) Yo agregaría un ciclo while infinito que no permita avanzar a menos que se ingrese un valor válido para esa lista. El ciclo irá aumentando un contador y evaluando; si el nodo tmp llega al final y el contador sigue siendo menor que la posición ingresada, marcará un error y pedirá ingresar de nuevo el valor para posición, el ciclo solo se romperá en el caso de que el contador sea mayor que el índice ingresado, volverá a llevar al nodo tmp al inicio de la lista para seguir con el flujo del programa.

e)

```
while(1){
    if (temp==NULL && i<=posicion){
        printf("error. Ingrese otra posicion");
        scanf("%d",&posicion);
        temp=lista->head;
    }
    else{
        temp=lista->head;
        break;
    }
    temp=temp->next;
    i++;
}
```

Esto considerando que las posiciones comiencen desde 1, ya que i comienza siendo igual a 0, por lo que si se intenta colocar en la posición 0 marcaría error.

## Actividad 4

a) En este inciso se pide completar las funciones de borrarPrimero y borrarUltimo y realizar un programa que pruebe su funcionamiento.

Para completar la función de borrarPrimero hice lo siguiente:

```
void borrarPrimero(Lista *lista) {
    if (lista->head == NULL) {
        printf("La lista esta vacia");
    }
    else{
        Nodo *nuevo_head = NULL;
        Nodo *tmp=lista->head;
        nuevo_head=tmp->next;
        free(lista->head);
        lista->head = nuevo_head;
        lista->tamano--;
    }
}
```

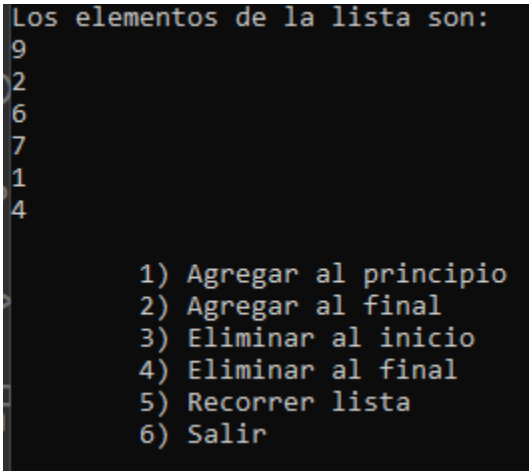
La función estaba incompleta en la parte del else, donde se elimina el primer elemento en caso de que la lista no esté vacía. Lo que hace esta sección de código es crear dos nodos que servirán como auxiliares, el “nuevo_head” se posiciona contiguo al head y el nodo tmp se posiciona en head, luego se libera el contenido de head y finalmente se actualizan la referencia head, con lo que se asegura que el último elemento vuelva a apuntar a este nuevo head, finalmente se decrementa el tamaño de la lista dado que se eliminó un elemento.

La función borrarUltimo también estaba incompleta, aunque el ejercicio no lo decía, le faltaba lo que va dentro del else. Para esta función hice lo siguiente:

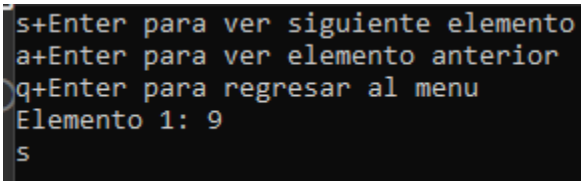
```
void borrarUltimo(Lista *lista) {
    if (lista->head == NULL) {
        printf("La lista esta vacia");
    }
    else{
        Nodo *tmp=lista->head;
        while(tmp->next->next != lista->head){
            tmp=tmp->next;
        }
        free(tmp->next);
        tmp->next=lista->head;
        lista->head->prev = tmp;
        lista->tamano--;
    }

}
```

Esta función elimina un elemento al final una vez que comprueba que la lisa no está vacía. Esto yendo hacia el penúltimo elemento con un nodo tmp, una vez ahí libera al siguiente de tmp, es decir, al último elemento. Luego de esto actualiza la referencia “prev” de head y la referencia siguiente del nodo tmp, para que ahora tmp sea el último elemento.
El programa donde se prueban estas funciones: 4.c

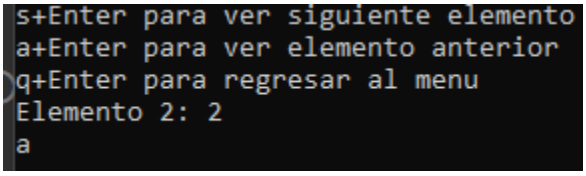
b) Para recorrer la lista creé una función desde cero usando como referencia la creada en la actividad anterior para recorrer los autos. Esta función consiste en un ciclo while infinito que no termina hasta que el usuario presione la tecla correspondiente para salir. Inicia mostrando el primer elemento de la lista y proporciona dos opciones, una para ver el elemento siguiente y otra para ver el elemento anterior. Los elementos de la lista se van mostrando con su respectivo índice(empezando desde 1); este índice se reinicia cuando se llega a head, y si se retrocede estando en head, el índice será igual al tamaño de la lista. Dependiendo de si se avanza o retrocede el índice va aumentando o disminuyendo. Considere la siguiente ejecución:



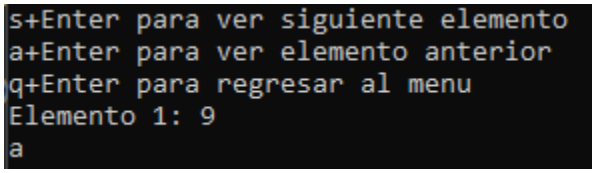

Entonces la lista se recorre como:



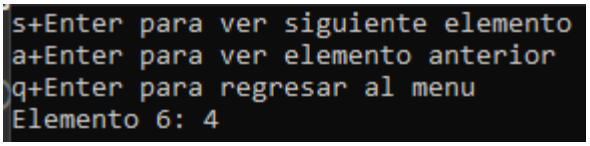

Como se muestra en la imagen, se eligió “s”, por lo que:




Si en este estado regresamos pulsando “a”:




Otra vez “a”:




Y se observa que se posiciona en el último elemento de la lista, por lo que se comprueba su comportamiento de lista ligada doble circular.

## Conclusiones

Durante esta práctica revisamos varios conceptos pertinentes a las listas, tales como sus principales funciones, procedimientos y características, remontándonos a sus definiciones y estructuras principales. Todo esto lo pusimos en práctica con diferentes ejemplos que nos permitieron manipular las listas de una forma muy dinámica, implementando las diferentes funciones para crear programas que simularan el manejo completo de los diferentes tipos de listas; desde agregar y eliminar elementos hasta recorrer la lista.

Además, construimos una lista de datos abstractos lo cual, para mí, fue lo más complicado dado que hay que atender los diferentes atributos del dato abstracto, en estos casos es donde se demuestran los beneficios de la programación funcional, generando códigos prácticos y fáciles de leer.

En esta última práctica sentí más que en ninguna otra cómo usaba todos los conocimientos aprendidos en el curso, conjuntando todos los conceptos previos, desde apuntadores, TDA y finalmente estructuras de ordenamiento.