UNIVERSIDAD DE COSTA RICA

ESCUELA DE INGENIERÍA ELÉCTRICA

IE0117 PROGRAMACIÓN BAJO PLATAFORMAS ABIERTAS

Reporte

Laboratorio 5

Prof. Carolina Trejos Quirós

Estudiante: Diego Pereira[1], B85938

17 de noviembre de 2025

ÍNDICE

1. Introducción

Este laboratorio de programación bajo plataformas abiertas hace uso de memoria dinámica para la creación de nodos y manipulación de listas que pueden simular pilas (*stack*) tipo arreglos que se pueden recorrer mediante el uso apropiado de punteros que enlazan cada nodo como representación de un índice en el arreglo o pila. Además, de mejorar la destreza en el uso del lenguaje C mediante la puesta en práctica de dichos conocimientos adquiridos, también se agrega funcionalidad extra para el comando "make" que ayuda mejorar un poco la automatización en la compilación de estructuras de directorios y archivos un poco más compleja que si fuera compilar un único archivo.

2. Implementación

2.1. **Ejercicio1**: Estructuras de datos

La manipulación de estructuras de datos se puede realizar de diversas maneras, sean arreglos dinámicos, listas enlazadas simples, listas enlazadas dobles, pilas (del inglés *stack*), entre otros. Debido a la amplia variedad de contenidos, este laboratorio se limita a escoger únicamente un tipo de estructura para presentar resultados, el cual muestra las listas enlazadas simples para este caso en particular.

2.1.1. Definición de un nodo para listas enlazadas simples

A diferencia de las listas enlazadas dobles que se componen de dos punteros, uno hacia el nodo anterior y otro hacia el siguiente, este caso muestra un nodo de una lista enlazada simple compuesto únicamente por un puntero que conecta con el siguiente nodo, por lo que dependiendo de la manipulación a realizar en la estructura, la dificultad en el recorrido puede ser un poco más complicada que si fueran listas enlazadas dobles, ya que no se sabe cuál es el nodo anterior, lo que requiere mayor recorrido en la búsqueda de un nodo desde el inicio.

```
52  // data type for a singly node
53  // typedef struct SNode SNode;
54  typedef struct SNode {
55      int index;
56      //char locale[5];
57      struct SNode* next;
58  } SNode;
```

Figura 1. Elaboración propia. *Definición de la estructura para los datos de un nodo simple*, 2025.

2.1.2. Crear un nodo para listas enlazadas simples

Se hace uso de memoria dinámica para reservar el espacio que ocupan los datos de un nodo.

```
146  // return: node
147  // allocate memory to create the new node
148  SNode* s_create(int index) {
149      SNode* node = (NULL);
150      node = (SNode*)malloc(sizeof(SNode));
151      if (!(node)) {
152          //perror("No se pudo reservar memoria para el nodo.");
153          printf("No se pudo reservar memoria para el nodo.");
154          free(node);
155          return (NULL);
156      }
157      (node)->index = index;
158      (node)->next = (NULL);
159      return (node);
160  }
```

Figura 2. Elaboración propia. *Función para crear un nodo de listas enlazadas simples*, 2025.

2.1.3. Insertar un nodo en una lista enlazada simple

Los tipos de inserción en una lista son variados, en este caso se muestran tres tipos, inserción al inicio donde el nodo toma el lugar de la cabeza, inserción al final que requiere recorrer la lista para averiguar cuál es el último nodo y ocupar la última posición, y como tercer tipo una inserción ordenada que puede ser de forma ascendente o descendente según el valor del índice, lo cual requiere averiguar no sólo cuál es el siguiente nodo, sino también el anterior.

```
161  // return: head
162  // insertion at the beginning (head)
163  SNode* s_insert_at_head(SNode *head, SNode *node) {
164      if (!(node)) {
165          printf("%s\n", "Nodo es NULL (no se encuentra inicializado).");
166          return (NULL);
167      }
168      (node)->next = (head);
169      (head) = (node);
170      return (head);
171  }
```

Figura 3. Elaboración propia. *Inserción al inicio de una lista enlazada simple*, 2025.

```
186   // return: head
187   // insertion at the end (tail)
188   SNode* s_insert_at_tail(SNode *head, SNode *node) {
189       if (!(node)) {
190           printf("%s\n", "Nodo es NULL (no se encuentra inicializado).");
191           return (NULL);
192       }
193       SNode* tail = (NULL);
194       tail = s_find_tail((head));
195       if (!(tail)) {
196           (head) = (node);
197       } else {
198           (tail)->next = (node);
199       }
200       return (head);
201   }
```

Figura 4. Elaboración propia. *Inserción al final de una lista enlazada simple*, 2025.

```
304   // return: head
305   // insertion into a sorted linked list, according to order (ASC/DESC)
306   SNode* s_insert_sorted(SNode *head, SNode *node, int order) {
307       if (!(node)) {
308           printf("%s\n", "Nodo es NULL (no se encuentra inicializado).");
309           return (NULL);
310       }
311       SNode* ref = (NULL);
312       if (order == S_ASCENDING) {
313           ref = s_find_prev_smaller((head), (node));
314       } else {
315           ref = s_find_prev_greater((head), (node));
316       }
317       if (!(ref)) {
318           (node)->next = (head);
319           (head) = (node);
320       } else {
321           (node)->next = (ref)->next;
322           (ref)->next = (node);
323       }
324       return (head);
325   }
```

Figura 5. Elaboración propia. *Inserción ordenada (ASC/DESC) en una lista enlazada simple*, 2025.

2.1.4. Eliminar un nodo de una lista enlazada simple

Eliminar un nodo requiere saber no sólo cuál es el siguiente nodo, sino también el anterior.

```
328  // return: head
329  // delete a node from a singly linked list
330  SNode* s_delete(SNode *head, SNode *node) {
331      if (!(head) || !(node)) {
332          return (NULL);
333      }
334      SNode* ref = (NULL);
335      ref = s_find_ref((head), (node));
336      if (!(ref)) {
337          return (head);
338      }
339      SNode* prev = (NULL);
340      prev = s_find_prev_to_ref((head), (ref));
341      if (!(prev)) {
342          // if head node itself holds the value to be deleted
343          SNode* temp = (head);
344          (head) = (ref)->next;
345          free(temp); // free memory
346          (temp) = (NULL);
347      } else {
348          // unlink the node from the linked list
349          (prev)->next = (ref)->next;
350          free(ref); // free allocated memory
351          (ref) = (NULL);
352      }
353      return (head);
354  }
```

Figura 6. Elaboración propia. *Función para eliminar un nodo de una lista enlazada simple*, 2025.

2.1.5. Liberar los nodos de una lista enlazada simple

Se deben liberar los nodos no sólo cuando se eliminan, sino también al finalizar la ejecución.

```
489  // function to print the linked list
490  void s_print_list(SNode *node) {
491      while ((node) != (NULL)) {
492          printf("%3d", (node)->index);
493          (node) = (node)->next;
494      }
495      printf("\n");
496  }
497  // free allocated memory (important to prevent memory leaks)
498  void s_free_list(SNode *head) {
499      SNode* temp = (NULL);
500      while ((head) != (NULL)) {
501          (temp) = (head);
502          (head) = (head)->next;
503          free(temp); // free allocated memory
504          (temp) = (NULL);
505      }
506  }
```

Figura 7. Elaboración propia. *Imprimir y liberar los nodos de una lista enlazada simple*, 2025.

2.1.6. Imprimir los nodos de una lista enlazada simple

Se muestra el uso de funciones para manipular listas, tanto para crear, insertar, e imprimir los nodos.

```
556     printf("%s\n", "-------------- Lista enlazada SIMPLE --------------");
557     // check if the singly linked list is empty
558     printf("La lista se encuentra vacía? %s\n", (s_isempty(head) ? "Sí" :
        "No"));
559     // create a RANDOM singly linked list
560     rand_max = RANDOM_MAX;
561     fill_array((rand_max), (array), BYINDEX);
562     printf("%s\n", "---------------- Orden : ALEATORIO ----------------");
563     for (int i = 0; i < (S_LIST_LENGTH); i++) {
564         int index = random_unique_array_v2((&rand_max), (array));
565         node = s_create(index);
566         //head = s_insert_sorted(head, node, S_DESCENDING);
567         //head = s_insert_sorted(head, node, S_ASCENDING);
568         head = s_insert_at_tail(head, node);
569         //s_print_list(head);
570     }
571     s_print_list(head);
572     // sort S_DESCENDING singly linked list
573     printf("%s\n", "---------------- Orden : DESCENDENTE ----------------");
574     //head = s_bubble_sort(head, S_DESCENDING); // only ASC/DESC order,
        other will stay as is
575     // merge sort for linked lists (more efficient approach than bubble
        sort)
576     head = s_merge_sort(&head, S_DESCENDING); // only ASC/DESC order,
        other will stay as is
577     s_print_list(head);
578     // sort S_ASCENDING singly linked list
579     printf("%s\n", "---------------- Orden :  ASCENDENTE ----------------");
580     head = s_bubble_sort(head, S_ASCENDING); // only ASC/DESC order,
        other will stay as is
581     // merge sort for linked lists (more efficient approach than bubble
        sort)
582     //head = s_merge_sort(&head, S_ASCENDING); // only ASC/DESC order,
        other will stay as is
583     s_print_list(head);
```

Figura 8. Elaboración propia. *Crear, insertar e imprimir los nodos de una lista enlazada simple*, 2025.

2.1.7. Simular una pila (del inglés *stack*) con una lista enlazada simple

La pila (*stack*) también hace uso de las funciones de una lista para manipular los nodos.

```
460   // return: head
461   // add a node to the end of list
462   SNode* s_push(SNode *head, SNode *node) {
463       return (s_insert_at_tail(head, node));
464   }
465   // return: tail
466   // get the last node on the list
467   SNode* s_peek(SNode *head) {
468       return (s_find_tail(head));
469   }
470   // return: tail
471   // get the last node on the list
472   SNode* s_top(SNode *head) {
473       return (s_find_tail(head));
474   }
475   // return: head
476   // delete the last node from the list
477   SNode* s_pop(SNode *head) {
478       SNode* tail = (NULL);
479       tail = s_find_tail((head));
480       return (s_delete(head, tail));
481   }
482   // return true if the list is empty
483   bool s_isempty(SNode *head) {
484       if (!(head)) {
485           return (true);
486       }
487       return (false);
488   }
```

Figura 9. Elaboración propia. *Stack usando las funciones de listas enlazadas simples,* 2025.

2.1.8. Imprimir los nodos de la pila (*stack*)

Se muestra el uso de funciones para manipular pilas, tanto para crear, insertar, e imprimir los nodos.

```
584     printf("%s\n", "-------------- Lista enlazada SIMPLE --------------");
585     // check if the singly linked list is empty
586     printf("La lista se encuentra vacía? %s\n", (s_isempty(head) ? "Sí" :
        "No"));
587     printf("%s\n", "------------ Implementaciones de STACK ------------");
588     printf("%s", "Push:");
589     node = s_create(RANDOM_MAX);
590     printf("%3d\n", (!(node) ? 0 : ((node)->index)));
591     head = s_push(head, node);
592     s_print_list(head);
593     printf("%s", "Peek:");
594     ref = s_peek(head);
595     printf("%3d\n", (!(ref) ? 0 : ((ref)->index)));
596     printf("%s", "Top :");
597     ref = s_top(head);
598     printf("%3d\n", (!(ref) ? 0 : ((ref)->index)));
599     printf("%s", "Pop :");
600     printf("%3d\n", (!(node) ? 0 : ((node)->index)));
601     head = s_pop(head);
602     s_print_list(head);
603     s_free_list(head); // free allocated memory
604     head = (NULL);
605     return (0);
```

Figura 10. Elaboración propia. *Imprimir la simulación de funciones de la pila (stack)*, 2025.

2.2. **Ejercicio2**: Repositorio

Dado que el ejercicio 2 se refiere a la estructura de directorios y archivos en el repositorio, y la nueva forma de compilación y ejecución mediante el uso del comando "*make*", para más información acerca de esta sección se puede consultar el archivo README o la sección de resultados.

3. RESULTADOS

3.1. **Ejercicio1**: Estructuras de datos

Esta sección muestra tanto los resultados obtenidos para listas enlazadas simples y pilas (*stack*), como para los requerimientos del ejercicio 2 acerca de la nueva forma de compilar y ejecutar programas mediante el uso del comando "*make*".

3.1.1. Limpiar compilados con "*make*"

En ocasiones es útil tener una opción que permita eliminar los archivos de forma automática.

```
diego@pereira:~/Desktop/lab05$ make clean
rm -rfv bin obj lab05e1singly
removed 'bin/lab05e1singly'
removed directory 'bin'
removed 'obj/src/umath.o'
removed 'obj/src/slist.o'
removed 'obj/src/sstack.o'
removed directory 'obj/src'
removed 'obj/lab05e1singly.o'
removed directory 'obj'
removed 'lab05e1singly'
diego@pereira:~/Desktop/lab05$
```

Figura 12. Elaboración propia. *Resultado del comando para limpiar los compilados*, 2025.

3.1.2. Pasos para compilar con "*make*"

El siguiente comando permite compilar toda la estructura de directorios sin la ejecución.

```
diego@pereira:~/Desktop/lab05$ make all
Starting...
make lab05e1singly
make[1]: Entering directory '/home/oem/Desktop/lab05'
Compiling without debug flags...
gcc -Iinclude -c lab05e1singly.c -o obj/./lab05e1singly.o -lm -Wall -Wextra
-Wpedantic -std=c11
Compiling without debug flags...
gcc -Iinclude -c src/slist.c -o obj/./src/slist.o -lm -Wall -Wextra -Wpedant
ic -std=c11
Compiling without debug flags...
gcc -Iinclude -c src/sstack.c -o obj/./src/sstack.o -lm -Wall -Wextra -Wpeda
ntic -std=c11
Compiling without debug flags...
gcc -Iinclude -c src/umath.c -o obj/./src/umath.o -lm -Wall -Wextra -Wpedant
ic -std=c11
Building without debug flags...
gcc -Iinclude -o bin/lab05e1singly obj/./lab05e1singly.o obj/./src/slist.o o
bj/./src/sstack.o obj/./src/umath.o -lm -Wall -Wextra -Wpedantic -std=c11
make[1]: Leaving directory '/home/oem/Desktop/lab05'
diego@pereira:~/Desktop/lab05$
```

Figura 13. Elaboración propia. *Resultado del comando para compilar el laboratorio*, 2025.

3.1.3. Pasos para ejecutar con "*make*"

Se muestran dos formas de ejecución, aunque internamente las dos hacen uso del comando "*make*".

```
diego@pereira:~/Desktop/lab05$ ./lab05e1singly_run
diego@pereira:~/Desktop/lab05$ make run
Starting...
make lab05e1singly
make[1]: Entering directory '/home/oem/Desktop/lab05'
Building without debug flags...
gcc -Iinclude -o bin/lab05e1singly obj/./lab05e1singly.o obj/./src/slist.o o
bj/./src/sstack.o obj/./src/umath.o -lm -Wall -Wextra -Wpedantic -std=c11
make[1]: Leaving directory '/home/oem/Desktop/lab05'
Executing...
./bin/lab05e1singly
#valgrind --tool=memcheck --leak-check=full ./bin/lab05e1singly
./bin/lab05e1singly
-------------- Lista enlazada SIMPLE --------------
La lista se encuentra vacía? Sí
---------------- Orden : ALEATORIO ----------------
 40 12 69 67 16 75 47 59 30 44 50 10  5 95 48 28
--------------- Orden : DESCENDENTE ---------------
 95 75 69 67 59 50 48 47 44 40 30 28 16 12 10  5
--------------- Orden :  ASCENDENTE ---------------
  5 10 12 16 28 30 40 44 47 48 50 59 67 69 75 95
-------------- Lista enlazada SIMPLE --------------
La lista se encuentra vacía? No
------------ Implementaciones de STACK ------------
Push: 99
  5 10 12 16 28 30 40 44 47 48 50 59 67 69 75 95 99
Peek: 99
Top : 99
Pop : 99
  5 10 12 16 28 30 40 44 47 48 50 59 67 69 75 95
diego@pereira:~/Desktop/lab05$
```

Figura 14. Elaboración propia. *Resultado del comando para correr el laboratorio*, 2025.

3.1.4. Análisis de memoria con "*valgrind*"

Los resultados del comando *"valgrind"* muestra un correcto uso de la memoria.

```
diego@pereira:~/Desktop/lab05$ ./lab05e1singly_gcc
==3429== Memcheck, a memory error detector
==3429== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==3429== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==3429== Command: ./lab05e1singly
==3429==
-------------- Lista enlazada SIMPLE --------------
La lista se encuentra vacía? Sí
---------------- Orden : ALEATORIO ----------------
 25 43 11 78 93 27 82 52  2 28  9  4 83 51 63 13
--------------- Orden : DESCENDENTE ---------------
 93 83 82 78 63 52 51 43 28 27 25 13 11  9  4  2
--------------- Orden :  ASCENDENTE ---------------
  2  4  9 11 13 25 27 28 43 51 52 63 78 82 83 93
-------------- Lista enlazada SIMPLE --------------
La lista se encuentra vacía? No
------------ Implementaciones de STACK ------------
Push: 99
  2  4  9 11 13 25 27 28 43 51 52 63 78 82 83 93 99
Peek: 99
Top : 99
Pop : 99
  2  4  9 11 13 25 27 28 43 51 52 63 78 82 83 93
==3429==
==3429== HEAP SUMMARY:
==3429==     in use at exit: 0 bytes in 0 blocks
==3429==   total heap usage: 46 allocs, 46 frees, 5,245 bytes allocated
==3429==
==3429== All heap blocks were freed -- no leaks are possible
==3429==
==3429== For lists of detected and suppressed errors, rerun with: -s
==3429== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
diego@pereira:~/Desktop/lab05$
```

76 × 40

Figura 15. Elaboración propia. *Resultado de la revisión de memoria con "valgrind"*, 2025.

3.2. **Ejercicio2**: Repositorio

3.2.1. Contenido del repositorio

Se muestra la misma estructura de directorios y archivos que contiene el repositorio en la nube.

```
diego@pereira:~/Desktop/lab05$ ls -las
total 88
 4 drwxrwxr-x  4 diego diego  4096 Nov 17 12:21 .
 4 drwxr-x--- 16 diego diego  4096 Nov 17 12:21 ..
 4 drwxr-xr-x  2 diego diego  4096 Nov 17 12:21 include
32 -rwxr-xr-x  1 diego diego 30512 Nov 17 12:21 lab05e1singly
20 -rw-r--r--  1 diego diego 19765 Nov 17 12:21 lab05e1singly_aio.c
 8 -rw-r--r--  1 diego diego  6018 Nov 17 12:21 lab05e1singly.c
 4 -rwxr-xr-x  1 diego diego   399 Nov 17 12:21 lab05e1singly_gcc
 4 -rwxr-xr-x  1 diego diego   284 Nov 17 12:21 lab05e1singly_run
 4 -rwxr-xr-x  1 diego diego  3969 Nov 17 12:21 Makefile
 4 drwxr-xr-x  2 diego diego  4096 Nov 17 12:21 src
diego@pereira:~/Desktop/lab05$ ls -las include/
total 24
4 drwxr-xr-x 2 diego diego 4096 Nov 17 12:21 .
4 drwxrwxr-x 4 diego diego 4096 Nov 17 12:21 ..
4 -rw-r--r-- 1 diego diego 2745 Nov 17 12:21 slist.h
4 -rw-r--r-- 1 diego diego  201 Nov 17 12:21 SNode.h
4 -rw-r--r-- 1 diego diego  642 Nov 17 12:21 sstack.h
4 -rw-r--r-- 1 diego diego 1268 Nov 17 12:21 umath.h
diego@pereira:~/Desktop/lab05$ ls -las src/
total 32
 4 drwxr-xr-x 2 diego diego 4096 Nov 17 12:21 .
 4 drwxrwxr-x 4 diego diego 4096 Nov 17 12:21 ..
12 -rw-r--r-- 1 diego diego 9629 Nov 17 12:21 slist.c
 4 -rw-r--r-- 1 diego diego   47 Nov 17 12:21 SNode.c
 4 -rw-r--r-- 1 diego diego  840 Nov 17 12:21 sstack.c
 4 -rw-r--r-- 1 diego diego 4003 Nov 17 12:21 umath.c
diego@pereira:~/Desktop/lab05$
```

Figura 16. Elaboración propia. *Estructura de archivos que contiene el repositorio*, 2025.

4. Análisis de Resultados

Para la sección 3.1 correspondiente al ejercicio 1 acerca de una estructura de datos para simular una pila (*stack*), las Figuras 12 a la 15 muestran los resultados obtenidos tanto con el comando "*make*" para compilar y ejecutar, como con el comando "*valgrind*" para analizar el correcto uso de la memoria y verificar que no haya espacios en memoria sin ser liberados.

La Figura 14 es el resultado de ejecución con "*make*", mientras que la Figura 15 es el resultado de la ejecución "*valgrind*". En ambos casos se llena una pila de 16 nodos con números aleatorios entre 1 y 99, siendo una inserción de forma ordenada, mientras que la segunda de forma aleatoria para antes de imprimirla ser ordenada. Con el número 99 se crea el nodo 17 usado para realizar manipulaciones de "*stack*" realizando la inserción al final para luego ser extraído de la pila.

Las Figuras 12 y 13 no son de ejecución, pero sí realizan operaciones útiles como la limpieza de archivos compilados, así como el proceso de compilación que corresponde al paso previo que se realiza antes de la ejecución en caso de querer realizarlo de forma separada a la ejecución.

Para la sección 3.2 que corresponde al ejercicio 2, la Figura 16 muestra la misma estructura de directorios y archivos que contiene el repositorio en la nube, la cual tiene dos directorios, uno con los "*header files*" o archivos con la declaración de funciones, y otro con los "*source files*" o archivos fuente con la definición de las funciones escritas en el lenguaje C.

5. Conclusiones y Recomendaciones

Se verifica que el comando "*valgrind*" es una buena herramienta para ayudar a mejorar la administración de memoria en nuestro código, especialmente cuando se usan punteros y memoria dinámica, el cual contribuye a chequear si toda la memoria reservada durante la ejecución del programa es liberada correctamente.

Y como punto adicional, se comprueba que el comando "*make*" también contribuye a agilizar el proceso de compilación y ejecución de programas, especialmente cuando la estructura de directorios y archivos se vuelve más compleja, sin embargo, es importante mencionar que no es el único comando para dicho fin, ya que también existen otras opciones como "*cmake*", "*meson*", "*ninja*", entre otros que pueden realizar la misma tarea con menor cantidad de líneas de código.