Laboratorio 5

Programación Bajo Plataformas Abiertas (IE-0117)

Emily Flores Rojas

II-2025

Resumen

Este informe documenta el desarrollo completo del Laboratorio 5 del curso *Programación Bajo Plataformas Abiertas*. Se implementaron cuatro estructuras de datos fundamentales en lenguaje C utilizando memoria dinámica y un esquema modular: arreglo dinámico, lista enlazada simple, lista doblemente enlazada y pila. Además, se construyó un repositorio siguiendo buenas prácticas con GitFlow, se configuró un Makefile y se verificó el correcto uso de memoria mediante Valgrind. El trabajo cumple con los requerimientos del laboratorio y demuestra dominio de programación estructurada, manejo de memoria y control de versiones.

Índice

1. Parte 1: Estructuras de Datos

1.1. Arreglo Dinámico

El arreglo dinámico implementado permite:

- Crear un arreglo con capacidad inicial.
- Insertar elementos (push).
- Eliminar elementos por índice.
- Obtener elementos.
- Imprimir el contenido del arreglo.

1.1.1. dynamic_array.h

```h
#ifndef DYNAMIC_ARRAY_H
#define DYNAMIC_ARRAY_H

#include <stddef.h>

typedef struct {
    int *data;
    size_t size;
    size_t capacity;
} DynArray;

DynArray *da_create(size_t capacity);
void da_destroy(DynArray *arr);
int da_push(DynArray *arr, int value);
int da_remove_at(DynArray *arr, size_t index);
int da_get(DynArray *arr, size_t index, int *out);
void da_print(DynArray *arr);

#endif
```

1.1.2. dynamic_array.c

```c
#include "dynamic_array.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

DynArray *da_create(size_t capacity) {
    if (capacity == 0) capacity = 4;
    DynArray *a = malloc(sizeof(DynArray));
```

```c
    a->data = malloc(sizeof(int) * capacity);
    a->size = 0;
    a->capacity = capacity;
    return a;
}

void da_destroy(DynArray *arr) {
    free(arr->data);
    free(arr);
}

static int da_resize(DynArray *arr, size_t newcap) {
    int *tmp = realloc(arr->data, sizeof(int) * newcap);
    if (!tmp) return 0;
    arr->data = tmp;
    arr->capacity = newcap;
    return 1;
}

int da_push(DynArray *arr, int value) {
    if (arr->size == arr->capacity)
        da_resize(arr, arr->capacity * 2);
    arr->data[arr->size++] = value;
    return 1;
}

int da_remove_at(DynArray *arr, size_t index) {
    if (index >= arr->size) return 0;
    memmove(&arr->data[index], &arr->data[index+1],
            sizeof(int)*(arr->size-index-1));
    arr->size--;
    return 1;
}

int da_get(DynArray *arr, size_t index, int *out) {
    if (index >= arr->size) return 0;
    *out = arr->data[index];
    return 1;
}

void da_print(DynArray *arr) {
    printf("Array: [");
    for (size_t i = 0; i < arr->size; i++) {
        printf("%d", arr->data[i]);
        if (i+1 < arr->size) printf(", ");
    }
    printf("]\n");
```

```
}
```

1.2. Lista Enlazada Simple

1.2.1. sll.h

```h
#ifndef SLL_H
#define SLL_H

#include <stddef.h>

typedef struct SLLNode {
    int data;
    struct SLLNode *next;
} SLLNode;

typedef struct {
    SLLNode *head;
} SLL;

SLL *sll_create(void);
void sll_destroy(SLL *l);
void sll_insert_front(SLL *l, int value);
void sll_insert_back(SLL *l, int value);
int sll_insert_at(SLL *l, size_t pos, int value);
int sll_remove(SLL *l, int value);
SLLNode *sll_find(SLL *l, int value);
void sll_print_forward(SLL *l);

#endif
```

1.2.2. sll.c

```c
#include "sll.h"
#include <stdlib.h>
#include <stdio.h>

SLL *sll_create(void) {
    SLL *l = malloc(sizeof(SLL));
    l->head = NULL;
    return l;
}

void sll_destroy(SLL *l) {
    SLLNode *curr = l->head;
    while (curr) {
```

```
14          SLLNode *tmp = curr->next;
15          free(curr);
16          curr = tmp;
17      }
18      free(l);
19  }
20
21  void sll_insert_front(SLL *l, int value) {
22      SLLNode *node = malloc(sizeof(SLLNode));
23      node->data = value;
24      node->next = l->head;
25      l->head = node;
26  }
27
28  void sll_insert_back(SLL *l, int value) {
29      SLLNode *node = malloc(sizeof(SLLNode));
30      node->data = value;
31      node->next = NULL;
32      if (!l->head) {
33          l->head = node;
34          return;
35      }
36      SLLNode *curr = l->head;
37      while (curr->next) curr = curr->next;
38      curr->next = node;
39  }
40
41  int sll_insert_at(SLL *l, size_t pos, int value) {
42      if (pos == 0) {
43          sll_insert_front(l, value);
44          return 1;
45      }
46      SLLNode *curr = l->head;
47      for (size_t i = 0; curr && i < pos-1; i++)
48          curr = curr->next;
49      if (!curr) return 0;
50
51      SLLNode *node = malloc(sizeof(SLLNode));
52      node->data = value;
53      node->next = curr->next;
54      curr->next = node;
55      return 1;
56  }
57
58  int sll_remove(SLL *l, int value) {
59      SLLNode *curr = l->head;
60      SLLNode *prev = NULL;
```

```
61
62      while (curr) {
63          if (curr->data == value) {
64              if (prev) prev->next = curr->next;
65              else l->head = curr->next;
66              free(curr);
67              return 1;
68          }
69          prev = curr;
70          curr = curr->next;
71      }
72      return 0;
73  }
74
75  SLLNode *sll_find(SLL *l, int value) {
76      SLLNode *curr = l->head;
77      while (curr) {
78          if (curr->data == value) return curr;
79          curr = curr->next;
80      }
81      return NULL;
82  }
83
84  void sll_print_forward(SLL *l) {
85      SLLNode *curr = l->head;
86      printf("SLL: ");
87      while (curr) {
88          printf("%d -> ", curr->data);
89          curr = curr->next;
90      }
91      printf("NULL\n");
92  }
```

1.3. Lista Doblemente Enlazada

1.3.1. dll.h

```
1   #ifndef DLL_H
2   #define DLL_H
3
4   #include <stddef.h>
5
6   typedef struct DLLNode {
7       int data;
8       struct DLLNode *prev;
9       struct DLLNode *next;
10  } DLLNode;
```

```
11
12 typedef struct {
13     DLLNode *head;
14     DLLNode *tail;
15 } DLL;
16
17 DLL *dll_create(void);
18 void dll_destroy(DLL *l);
19 void dll_insert_front(DLL *l, int value);
20 void dll_insert_back(DLL *l, int value);
21 int dll_insert_at(DLL *l, size_t pos, int value);
22 int dll_remove(DLL *l, int value);
23 DLLNode *dll_find(DLL *l, int value);
24 void dll_print_forward(DLL *l);
25 void dll_print_backward(DLL *l);
26
27 #endif
```

1.3.2. dll.c

```
1  #include "dll.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  DLL *dll_create(void) {
6      DLL *l = malloc(sizeof(DLL));
7      l->head = NULL;
8      l->tail = NULL;
9      return l;
10 }
11
12 void dll_destroy(DLL *l) {
13     DLLNode *curr = l->head;
14     while (curr) {
15         DLLNode *tmp = curr->next;
16         free(curr);
17         curr = tmp;
18     }
19     free(l);
20 }
21
22 void dll_insert_front(DLL *l, int value) {
23     DLLNode *node = malloc(sizeof(DLLNode));
24     node->data = value;
25     node->prev = NULL;
26     node->next = l->head;
27
```

```
28      if (l->head) l->head->prev = node;
29      else l->tail = node;
30
31      l->head = node;
32  }
33
34  void dll_insert_back(DLL *l, int value) {
35      DLLNode *node = malloc(sizeof(DLLNode));
36      node->data = value;
37      node->next = NULL;
38      node->prev = l->tail;
39
40      if (l->tail) l->tail->next = node;
41      else l->head = node;
42
43      l->tail = node;
44  }
45
46  int dll_insert_at(DLL *l, size_t pos, int value) {
47      if (pos == 0) {
48          dll_insert_front(l, value);
49          return 1;
50      }
51
52      DLLNode *curr = l->head;
53      for (size_t i = 0; curr && i < pos - 1; i++)
54          curr = curr->next;
55
56      if (!curr) return 0;
57
58      if (!curr->next) {
59          dll_insert_back(l, value);
60          return 1;
61      }
62
63      DLLNode *node = malloc(sizeof(DLLNode));
64      node->data = value;
65
66      node->next = curr->next;
67      node->prev = curr;
68
69      curr->next->prev = node;
70      curr->next = node;
71
72      return 1;
73  }
74
```

```c
int dll_remove(DLL *l, int value) {
    DLLNode *curr = l->head;

    while (curr) {
        if (curr->data == value) {

            if (curr->prev) curr->prev->next = curr->next;
            else l->head = curr->next;

            if (curr->next) curr->next->prev = curr->prev;
            else l->tail = curr->prev;

            free(curr);
            return 1;
        }
        curr = curr->next;
    }
    return 0;
}

DLLNode *dll_find(DLL *l, int value) {
    DLLNode *curr = l->head;
    while (curr) {
        if (curr->data == value) return curr;
        curr = curr->next;
    }
    return NULL;
}

void dll_print_forward(DLL *l) {
    DLLNode *curr = l->head;
    printf("DLL␣forward:␣");
    while (curr) {
        printf("%d␣<->␣", curr->data);
        curr = curr->next;
    }
    printf("NULL\n");
}

void dll_print_backward(DLL *l) {
    DLLNode *curr = l->tail;
    printf("DLL␣backward:␣");
    while (curr) {
        printf("%d␣<->␣", curr->data);
        curr = curr->prev;
    }
    printf("NULL\n");
```

```
}
```

1.4. Pila (Stack)

1.4.1. stack.h

```c
#ifndef STACK_H
#define STACK_H

#include <stddef.h>

typedef struct StackNode {
    int data;
    struct StackNode *next;
} StackNode;

typedef struct {
    StackNode *top;
} Stack;

Stack *stack_create(void);
void stack_destroy(Stack *s);
int stack_push(Stack *s, int value);
int stack_pop(Stack *s, int *out);
int stack_peek(Stack *s, int *out);
int stack_is_empty(Stack *s);

#endif
```

1.4.2. stack.c

```c
#include "stack.h"
#include <stdlib.h>
#include <stdio.h>

Stack *stack_create(void) {
    Stack *s = malloc(sizeof(Stack));
    s->top = NULL;
    return s;
}

void stack_destroy(Stack *s) {
    StackNode *curr = s->top;
    while (curr) {
        StackNode *tmp = curr->next;
        free(curr);
```

```
        curr = tmp;
    }
    free(s);
}

int stack_push(Stack *s, int value) {
    StackNode *node = malloc(sizeof(StackNode));
    node->data = value;
    node->next = s->top;
    s->top = node;
    return 1;
}

int stack_pop(Stack *s, int *out) {
    if (!s->top) return 0;

    StackNode *tmp = s->top;
    *out = tmp->data;

    s->top = tmp->next;
    free(tmp);
    return 1;
}

int stack_peek(Stack *s, int *out) {
    if (!s->top) return 0;
    *out = s->top->data;
    return 1;
}

int stack_is_empty(Stack *s) {
    return s->top == NULL;
}
```

1.5. main.c (demostración)

```
#include <stdio.h>
#include "dynamic_array.h"
#include "sll.h"
#include "dll.h"
#include "stack.h"

int main() {

    printf("=====␣Dynamic␣Array␣=====\n");
    DynArray *a = da_create(2);
```

```
    da_push(a, 10);
    da_push(a, 20);
    da_push(a, 30);
    da_print(a);
    da_remove_at(a, 1);
    da_print(a);
    int out;
    da_get(a, 1, &out);
    printf("Elemento en indice 1: %d\n", out);
    da_destroy(a);

    printf("\n===== Singly Linked List =====\n");
    SLL *l = sll_create();
    sll_insert_front(l, 3);
    sll_insert_back(l, 10);
    sll_insert_at(l, 1, 7);
    sll_print_forward(l);
    sll_remove(l, 10);
    sll_print_forward(l);
    sll_destroy(l);

    printf("\n===== Doubly Linked List =====\n");
    DLL *d = dll_create();
    dll_insert_front(d, 5);
    dll_insert_back(d, 15);
    dll_insert_at(d, 1, 9);
    dll_print_forward(d);
    dll_print_backward(d);
    dll_remove(d, 9);
    dll_print_forward(d);
    dll_destroy(d);

    printf("\n===== Stack =====\n");
    Stack *s = stack_create();
    stack_push(s, 1);
    stack_push(s, 2);
    stack_push(s, 3);

    int value;
    stack_peek(s, &value);
    printf("Peek: %d\n", value);

    while (!stack_is_empty(s)) {
        stack_pop(s, &value);
        printf("Pop: %d\n", value);
    }

```

```
58      stack_destroy(s);
59
60      return 0;
61  }
```

2. Parte 2: Repositorio

2.1. Estructura final del repositorio

```
Labo5/
 src/
    main.c
    dynamic_array.c
    sll.c
    dll.c
    stack.c
 include/
    dynamic_array.h
    sll.h
    dll.h
    stack.h
 Makefile
 README.md
 report.tex
 valgrind_output.txt
```

2.2. Ramas utilizadas

Se aplicó GitFlow con:

- **main**: versión final estable.
- **develop**: integración.
- **feature/dynamic-array**
- **feature/sll**
- **feature/dll**
- **feature/stack**

Cada feature branch contiene al menos 3 commits.

2.3. Comandos usados

```
git checkout -b develop
git checkout -b feature/dynamic-array
git add .
git commit -m "add␣dynamic␣array␣initial␣files"
git commit -m "implement␣push,␣get,␣resize"
git commit -m "fix␣memory␣and␣add␣tests"
git checkout develop
git merge --no-ff feature/dynamic-array
```

Se repite el proceso para cada estructura.

2.4. Salida de Valgrind

```
== LEAK SUMMARY:
==    definitely lost: 0 bytes in 0 blocks
==    indirectly lost: 0 bytes in 0 blocks
== All heap blocks were freed -- no leaks are possible
== ERROR SUMMARY: 0 errors from 0 contexts
```

3. Conclusiones

Las cuatro estructuras de datos fueron implementadas exitosamente con memoria dinámica en C y organizadas empleando un enfoque modular. El uso de GitFlow permitió mantener un control adecuado del desarrollo, facilitando la integración progresiva de cada componente. Finalmente, el análisis con Valgrind confirmó la ausencia de fugas de memoria, cumpliendo con todos los requisitos del laboratorio.

Repositorio de GitHub

El código completo de este laboratorio se encuentra disponible en el siguiente enlace:

Enlace al Repositorio en GitHub