Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КОМП'ЮТЕРНИХ НАУК ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Катедра «Комп'ютерна інженерія та програмування»

ЗВІТ

про виконання лабораторної роботи №7

з навчальної дисципліни «Алгоритми та структури даних»

Варіант 9

Виконав студент:

Ульянов Кирило Юрійович
Група: КН-1023b

Перевірив:

старший викладач

Бульба С.С.

Харків-2024

1 Мета роботи

Набути практичного досвіду та закріпити знання про подання стека, дека, пріоритетної черги та дисципліни їх обслуговування.

2 Хід роботи

Розробити функції, що забезпечують запис та читання запитів із пріоритетної черги, стека або дека.

В кожному завданні для організації вказаної черги використати дві структури. Перевірити працездатність розроблених функцій. Послідовність виконання операцій запису та читання обирати випадково.

Моє завдання: Пріоритетна черга. Постановка запитів у чергу виконується за пріоритетом, зняття – підряд зі старших адрес (кінця черги). Черга організована на масиві та на списку. Пріоритет: max значення числового параметра; при збігу параметрів – FIFO.

2.1 Реалізація черги на базі масиву

Було реалізовано такі методи: *ініціалізація черги, додавання у кінець черги, вивід черги, видалення з початку черги, знищення черги.*

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

typedef struct
{
  int data;
  int priority;
} PriorityQueueElement;

// Array based priority queue
typedef struct
{
  PriorityQueueElement *elements;
  int size;
  int capacity;
  int front;
  int rear;
  size_t allocated_memory;
} PriorityQueue;

PriorityQueue init_priority_queue(int capacity)
{
  PriorityQueue pq;
  pq.elements =
      (PriorityQueueElement *)malloc(sizeof(PriorityQueueElement) *
          capacity);
  pq.size = 0;
  pq.capacity = capacity;
  pq.front = 0;
  pq.rear = 0;
  pq.allocated_memory = sizeof(PriorityQueueElement) * capacity;
  return pq;
}

void resize(PriorityQueue *pq)
{
  pq->capacity *= 2;
  pq->elements = (PriorityQueueElement *)realloc(
      pq->elements, sizeof(PriorityQueueElement) * pq->capacity);
  pq->allocated_memory = sizeof(PriorityQueueElement) * pq->capacity;
}

void push_priority_queue(PriorityQueue *pq, int data, int priority)
{
  if (pq->size == pq->capacity)
  {
    resize(pq);
  }

  PriorityQueueElement newElem = {data, priority};

  int i = pq->size - 1;

  while (i >= 0 && pq->elements[i].priority < priority)
```

```
54    {
55      pq->elements[i + 1] = pq->elements[i];
56      i--;
57    }
58
59    pq->elements[i + 1] = newElem;
60    pq->size++;
61  }
62
63  PriorityQueueElement pop_priority_queue(PriorityQueue *pq)
64  {
65    if (pq->size == 0)
66    {
67      printf("Priority queue is empty.\n");
68      PriorityQueueElement emptyElement = {0, 0};
69      return emptyElement;
70    }
71
72    PriorityQueueElement topElement = pq->elements[pq->front];
73
74    for (int i = 0; i < pq->size - 1; i++)
75    {
76      pq->elements[i] = pq->elements[i + 1];
77    }
78
79    pq->size--;
80    return topElement;
81  }
82
83  void destroy_priority_queue(PriorityQueue *pq)
84  {
85    free(pq->elements);
86    pq->allocated_memory = 0;
87  }
88
89  void print_priority_queue(PriorityQueue *pq)
90  {
91    if (pq->size == 0)
92    {
93      printf("Priority queue is empty.\n");
94      return;
95    }
96
97    for (int i = 0; i < pq->size; i++)
98    {
99      printf("Value: %d, Priority: %d", pq->elements[i].data, pq->elements[i
          ].priority);
100     if (i < pq->size - 1)
101     {
102       printf("\n");
103     }
104   }
105   printf("\n");
106 }
```

2.2 Реалізація черги на базі списку

Дана структура була побудована на базі перевикористовуваного зв‘язного списку який буде наведено нижче.

Було реалізовано такі методи: *ініціалізація черги, додавання у кінець черги, вивід черги, видалення з початку черги, знищення черги.*

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

static int compare_priority(const void *a, const void *b)
{
  const LPQElement *elemA = (const LPQ *)a;
  const LPQElement *elemB = (const LPQ *)b;

  if (elemA->priority != elemB->priority)
  {
    return elemB->priority - elemA->priority; // Max priority first
  }
  return 0; // FIFO when priority same
}

static void free_lpq_element(void *data)
{
  free(data);
}

static void print_lpq_element(const void *data)
{
  const LPQElement *elem = (const LPQ *)data;
  printf("Value: %d, Priority: %d\n", elem->value, elem->priority);
}

LPQ *init_lpq()
{
  LPQ *queue = (LPQ *)malloc(sizeof(LPQ));
  if (!queue)
  {
    perror("Cannot able to allocate memory!");
    exit(EXIT_FAILURE);
  }
  queue->head = NULL;
  queue->allocated_memory = sizeof(LPQ);
  return queue;
}

void enqueue_lpq(LPQ *queue, int value, int priority)
{
  LPQElement *newElement = (LPQElement *)malloc(sizeof(LPQElement));
  if (!newElement)
  {
    perror("Cannot able to allocate memory!");
    exit(EXIT_FAILURE);
  }
  newElement->value = value;
  newElement->priority = priority;
```

```
51
52    queue->allocated_memory += sizeof(LPQElement);
53
54    Node **head = &(queue->head);
55
56    if (!*head || compare_priority((*head)->data, newElement) > 0)
57    {
58      insert_head_linked_list(head, newElement);
59    }
60    else
61    {
62      Node *current = *head;
63      while (current->next && compare_priority(current->next->data,
          newElement) <= 0)
64      {
65        current = current->next;
66      }
67      insert_after_element_linked_list(current, newElement);
68    }
69  }
70
71  int dequeue_lpq(LPQ *queue)
72  {
73    if (!queue->head)
74    {
75      printf("Queue is empty!\n");
76      return -1;
77    }
78    LPQElement *top = (LPQElement *)queue->head->data;
79    int value = top->value;
80    queue->allocated_memory -= sizeof(LPQElement);
81
82    delete_node_linked_list(&(queue->head), top, compare_priority,
        free_lpq_element);
83    return value;
84  }
85
86  void destroy_lpq(LPQ *queue)
87  {
88    destroy_linked_list(&(queue->head), free_lpq_element);
89    free(queue);
90    queue->allocated_memory = 0;
91  }
92
93  void print_lpq(LPQ *queue)
94  {
95    print_linked_list(queue->head, print_lpq_element);
96  }
```

2.3 Реалізація універсального зв‘язного списку

Він є універсальним бо не має визначений тип даних для зберігання, а зберігає пустий вказівник, що дає змогу хранити там будь-що.

Також можна перевизначити певні методи що дає також універсальність використання.

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

Node *create_node_linked_list(void *data)
{
  Node *newNode = (Node *)malloc(sizeof(Node));
  if (!newNode)
  {
    perror("Cannot able to allocate memory!\n");
    exit(EXIT_FAILURE);
  }
  newNode->data = data;
  newNode->next = NULL;
  return newNode;
}

void insert_head_linked_list(Node **head, void *data)
{
  Node *newNode = create_node_linked_list(data);
  newNode->next = *head;
  *head = newNode;
}

void insert_after_element_linked_list(Node *node, void *data)
{
  if (!node)
    return;
  Node *newNode = create_node_linked_list(data);
  newNode->next = node->next;
  node->next = newNode;
}

void delete_node_linked_list(Node **head, void *target, CompareFunc
    comparer, void (*free_data)(void *))
{
  if (!head || !*head)
    return;

  Node *temp = *head;
  Node *prev = NULL;

  while (temp && comparer(temp->data, target) != 0)
  {
    prev = temp;
    temp = temp->next;
  }

  if (!temp)
    return;
```

```
50
51    if (prev)
52    {
53      prev->next = temp->next;
54    }
55    else
56    {
57      *head = temp->next;
58    }
59
60    if (free_data)
61    {
62      free_data(temp->data);
63    }
64    free(temp);
65  }
66
67  Node *find_element_linked_list(Node *head, void *target, CompareFunc
        comparer)
68  {
69    while (head)
70    {
71      if (comparer(head->data, target) == 0)
72      {
73        return head;
74      }
75      head = head->next;
76    }
77    return NULL;
78  }
79
80  void destroy_linked_list(Node **head, void (*free_data)(void *))
81  {
82    while (*head)
83    {
84      Node *temp = *head;
85      *head = (*head)->next;
86
87      if (free_data)
88      {
89        free_data(temp->data);
90      }
91      free(temp);
92    }
93  }
94
95  void print_linked_list(Node *head, void (*print_data)(const void *))
96  {
97    while (head)
98    {
99      print_data(head->data);
100     head = head->next;
101   }
102 }
```

2.4 Реалізація лабораторної програми

Було протестовано дві структури даних та порівняно за кількість використаної пам‘яті.

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

#include "general_utils.h"
#include "priority_queue_arr.h"
#include "priority_queue_list.h"

void task1()
{
  PriorityQueue pq = init_priority_queue(5);
  LPQ *lpq = init_lpq();

  push_priority_queue(&pq, 10, 1);
  push_priority_queue(&pq, 20, 4);
  push_priority_queue(&pq, 15, 2);
  push_priority_queue(&pq, 69, 5);

  enqueue_lpq(lpq, 10, 1);
  enqueue_lpq(lpq, 20, 4);
  enqueue_lpq(lpq, 15, 2);
  enqueue_lpq(lpq, 69, 5);

  highlightText("Elements of LIST base priority queue:\n", "blue");
  print_lpq(lpq);
  printf("Memory allocated: \033[34m%zu\033[0m bytes\n", lpq->
      allocated_memory);
  puts("");

  highlightText("Elements of ARRAY priority queue:\n", "blue");
  print_priority_queue(&pq);
  printf("Memory allocated: \033[32m%zu\033[0m bytes\n", pq.
      allocated_memory);
  puts("");

  highlightText("Adding new element with highest priority to LIST priority
       queue:\n", "yellow");
  enqueue_lpq(lpq, 100, 10);
  print_lpq(lpq);
  printf("Memory allocated: \033[32m%zu\033[0m bytes\n", lpq->
      allocated_memory);
  puts("");

  highlightText("Adding new element with highest priority to ARRAY
      priority queue:\n", "yellow");
  push_priority_queue(&pq, 100, 10);
  print_priority_queue(&pq);
  printf("Memory allocated: \033[32m%zu\033[0m bytes\n", pq.
      allocated_memory);
  puts("");

  highlightText("Adding new element with similar priority to LIST priority
       queue:\n", "blue");
  enqueue_lpq(lpq, 777, 2);
  print_lpq(lpq);
```

```
48      printf("Memory allocated: \033[32m%zu\033[0m bytes\n", lpq->
            allocated_memory);
49      puts("");
50
51      highlightText("Adding new element with similar priority to ARRAY
            priority queue:\n", "blue");
52      push_priority_queue(&pq, 777, 2);
53      print_priority_queue(&pq);
54      printf("Memory allocated: \033[32m%zu\033[0m bytes\n", pq.
            allocated_memory);
55      puts("");
56
57      highlightText("Pop 4 elements from LIST priority queue:\n", "yellow");
58      dequeue_lpq(lpq);
59      dequeue_lpq(lpq);
60      dequeue_lpq(lpq);
61      dequeue_lpq(lpq);
62      print_lpq(lpq);
63      printf("Memory allocated: \033[32m%zu\033[0m bytes\n", lpq->
            allocated_memory);
64      puts("");
65
66      highlightText("Pop 4 elements from ARRAY priority queue:\n", "yellow");
67      pop_priority_queue(&pq);
68      pop_priority_queue(&pq);
69      pop_priority_queue(&pq);
70      pop_priority_queue(&pq);
71      print_priority_queue(&pq);
72      printf("Memory allocated: \033[32m%zu\033[0m bytes\n", pq.
            allocated_memory);
73      puts("");
74
75      destroy_lpq(lpq);
76      destroy_priority_queue(&pq);
77  }
```

2.5 Результати роботи програми:

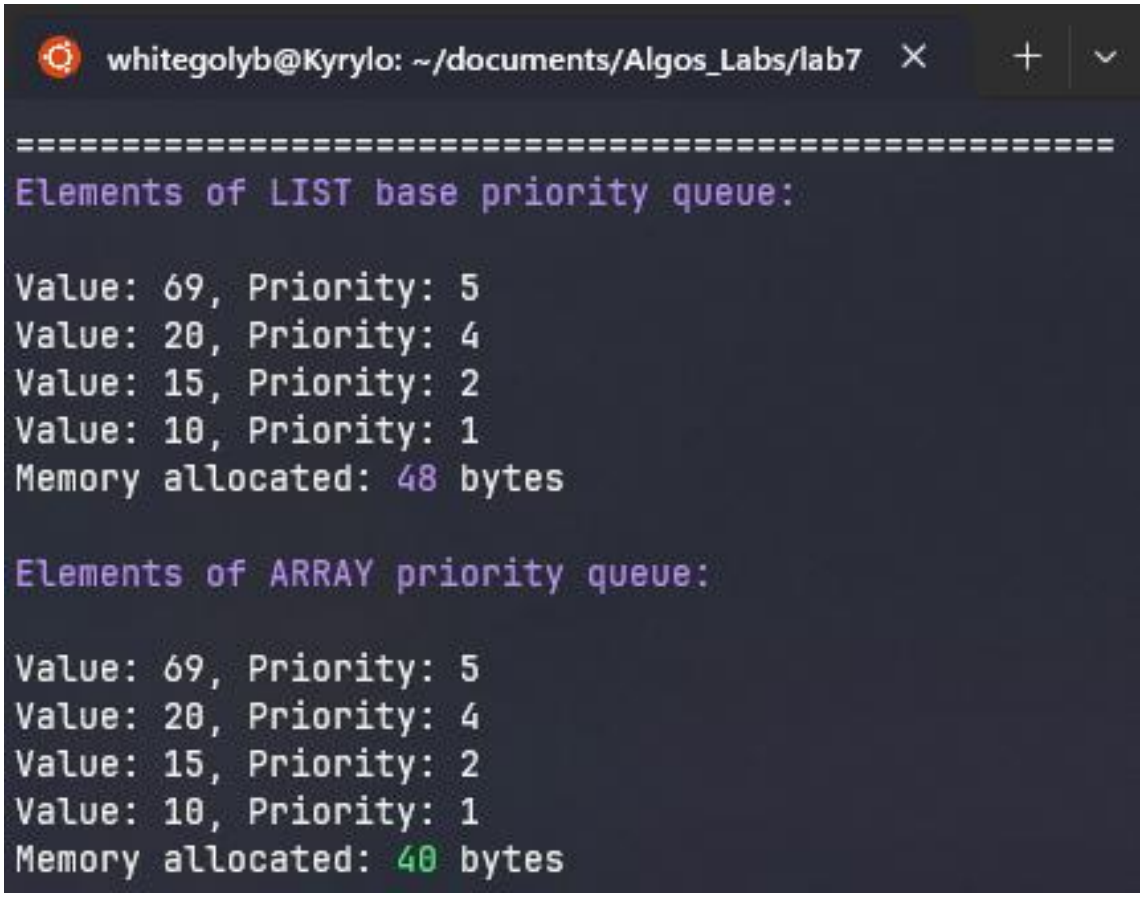

Рис. 1. Занесення елементів у черги

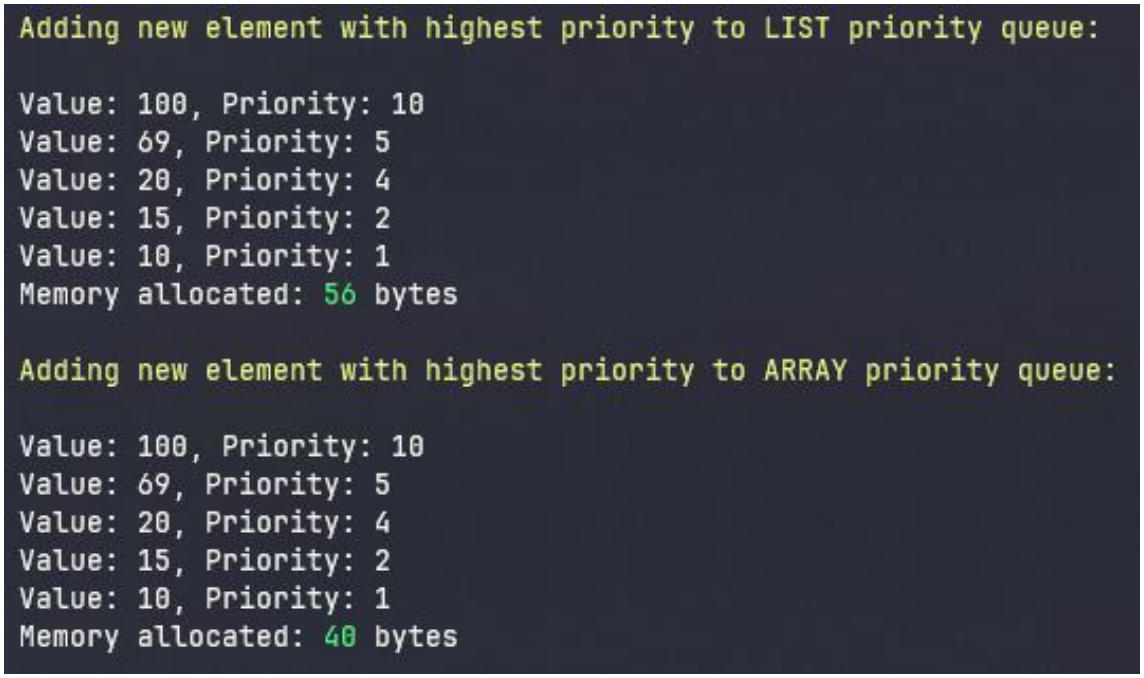

Рис. 2. Додавання нового елемента з найвищим пріорітетом

```
Adding new element with similar priority to LIST priority queue:

Value: 100, Priority: 10
Value: 69, Priority: 5
Value: 20, Priority: 4
Value: 15, Priority: 2
Value: 777, Priority: 2
Value: 10, Priority: 1
Memory allocated: 64 bytes

Adding new element with similar priority to ARRAY priority queue:

Value: 100, Priority: 10
Value: 69, Priority: 5
Value: 20, Priority: 4
Value: 15, Priority: 2
Value: 777, Priority: 2
Value: 10, Priority: 1
Memory allocated: 80 bytes
```

Рис. 3. Додавання елемента вже з існуючим пріорітетом

```
Pop 4 elements from LIST priority queue:

Value: 777, Priority: 2
Value: 10, Priority: 1
Memory allocated: 32 bytes

Pop 4 elements from ARRAY priority queue:

Value: 777, Priority: 2
Value: 10, Priority: 1
Memory allocated: 80 bytes
```

Рис. 4. Видалення 4 елементів з черг

3 Висновки

В ході виконання лабораторної роботи було розроблено дві структури даних відповідно свого варіанту.

Можна зазначити що всі структури даних працюють коректно та мають однаковий інтерфейс, але різну реалізацію.

Черга на базі масиву виділяє неперервну ділянку пам‘яті заздалегідь під декілька елементів, що може бути корисно коли черга заздалегідь визначена, швидкість доступу за індексом до елементів буде швидше.

Черга на базі списку виділя пам‘ять для кожного елементу окремо але не неперервно, це добре коли черга невизначена на може динамічно змінюватися, також швидше буде працювати операція видалення та додавання елементів.

Порівняння використаної пам‘яті можна побачити на скріншотах результатів.