МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра защиты информации

**

**ОТЧЁТ**

**по лабораторной работе №1**

**«**Абстрактные структуры данных**»**

**по дисциплине: «***Программирование***»**

Выполнил:Проверил:

Студент гр. «АБс-222», «АВТФ» *доцент кафедры ЗИ*

*Азява Д.А. Архипова А. Б.*

«\_\_\_» \_\_\_\_\_\_ 2023 г.«\_\_\_» \_\_\_\_\_\_ 2023 г.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (подпись)

Новосибирск 2023

Цели и задачи работы: изучение алгоритмов формирования и

работы с абстрактными структурами данных.

1. Реализовать структуру «Динамический массив»

Операция добавления элемента: O(n)

Операция удаления элемента: O(n)

Операция обращения к элементу: O(1)

Динамические массивы в отличии от статических могут менять свою длину.

Принцип работы: Когда мы хотим добавить новый элемент в динамический массив, в памяти ищется место, в которое мы сможем записать блок памяти, состоящий из текущего массива + 1 элемент. Затем в новый массив перезаписываются значения старого, а место, занимаемое старым, высвобождается, при необходимости, в новый элемент записываются данные.

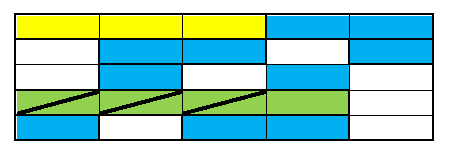


Рисунок 1 – Представление динамического массива в памяти

На рисунке 1 изображено представление перезаписи динамического массива в памяти. Где, жёлтым цветом изображён исходный массив из 3-х элементов, голубым занятые ячейки памяти, а зелёным получившийся массив, в котором косой чертой обозначены данные из исходного массива.

Таблица 1 - Листинг программы

|  |  |
| --- | --- |
| Файл | Код |
| array.c | #include "array.h"  #include <stdio.h>  #include <stdlib.h>  ARR\*\* create\_array(int size)  {  ARR\* array = (ARR\*)malloc(sizeof(ARR));  array->item = (ARR\*\*)calloc(size, sizeof(ARR\*));  array->size = size;  for (int i = 0; i < size; i++)  {  array->item[i] = NULL;  }  return array;  }  void array\_push(ARR\*\* arr, int position, void\* data)  {  if (arr == NULL)  {  printf("ERROR: the array is empty\n");  return;  }  if (position < 0)  {  printf("ERROR: the position can`t be less than zero\n");  return;  }  ARR\* array = \*arr;  if (position > array->size)  {  ARR\* new\_array = create\_array(position + 1);  node\* item = (node\*)malloc(sizeof(node));  for (int i = 0; i < array->size; i++)  {  new\_array->item[i] = array->item[i];  }  item->data = data;  new\_array->item[position] = item;  new\_array->size = position + 1;  \*arr = new\_array;  return;  }  node\* item = (node\*)malloc(sizeof(node));  item->data = data;  array->item[position] = item;  \*arr = array;  return;  }  void array\_pop(ARR\*\* arr, int index)  {  if (arr == NULL)  {  printf("ERROR: the array is empty\n");  return;  }  ARR\* array = \*arr;  if (index < 0 || index > array->size)  {  printf("ERROR: index isn`t right\n");  return;  }  ARR\* new\_array = create\_array(index);  node\* item = (node\*)malloc(sizeof(node));  for (int i = 0; i < index; i++)  {  new\_array->item[i] = array->item[i];  }  for (int i = index; i < array->size; i++)  {  new\_array->item[i] = array->item[i];  }  \*arr = new\_array;  }  void print\_element(ARR\* array, int index)  {  if (array == NULL)  {  printf("ERROR: the array is empty\n");  return;  }  if (index < 0 || index > array->size)  {  printf("ERROR: index isn`t right\n");  return;  }  node\* item = (node\*)malloc(sizeof(node));  item = array->item[index];  if (item == NULL)  {  printf("%d - NULL\n", index);  }  else if (item->data == NULL)  {  printf("%d - NULL\n", index);  }  else  {  printf("%d - %s\n", index, item->data);  }  }  void print\_array\_string(ARR\* array)  {  if (array == NULL)  {  printf("ERROR: the array is empty\n");  return;  }  node\* item = (node\*)malloc(sizeof(node));  for (int i = 0; i < array->size; i++)  {  item = array->item[i];  if (item == NULL)  {  printf("%d - NULL\n", i);  }  else if (item->data == NULL)  {  printf("%d - NULL\n", i);  }  else  {  printf("%d - %s\n", i, item->data);  }  }  } |
| array.h | #pragma once  typedef struct node {  void\* data;  }node;  typedef struct ARR  {  struct ARR\*\* item;  node\* node;  int size;  }ARR;  ARR\*\* create\_array(int);  void array\_push(ARR\*\*, int, void\*);  void print\_array\_string(ARR\*);  void print\_element(ARR\*, int);  void array\_pop(ARR\*\*, int); |

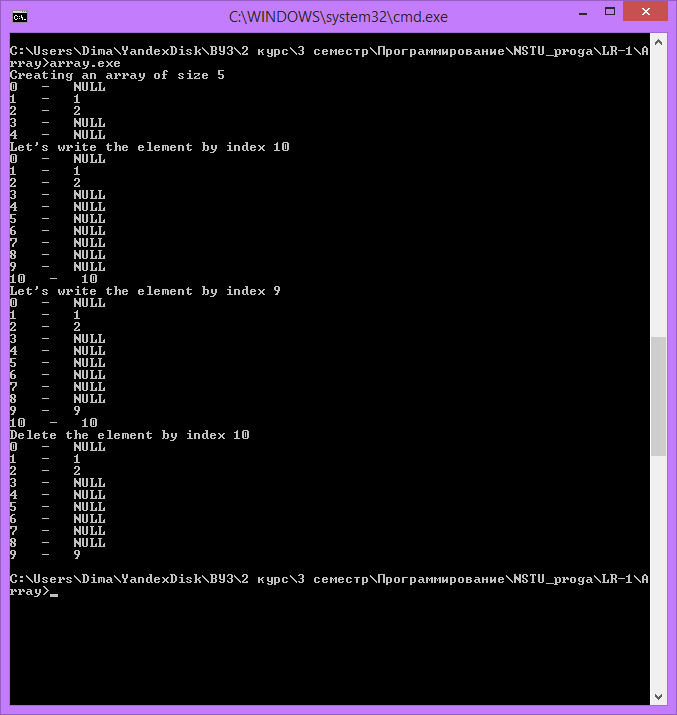


Рисунок 2 – Результат работы программы

1. Реализовать структуру «Список»

Операция добавления элемента: O(1)

Операция удаления элемента: O(1)

Операция обращения к элементу: O(n)

Основное отличие от массива заключается в том, что узлы(ячейки) расположены в памяти не последовательно, одним массивным блоком, а разбросаны по ней в по принципу куда смогли, туда и записали.

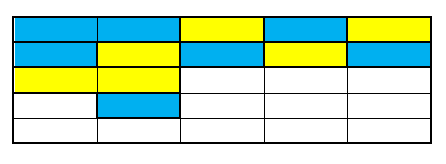


Рисунок 3 – Представление связных списков в памяти

* 1. Односвязный список

В односвязном списке вместе с элементом хранится указатель на следующий элемент, а в последнем элементу на месте указателя записывается NULL (Рисунок 4). Перемещаться по данной структуре мы можем, только перебирая элементы, поэтому сложность обращения к элементу O(n).

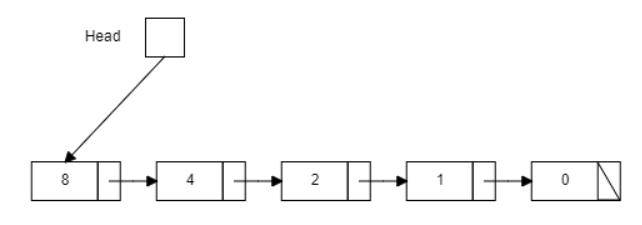


Рисунок 4 – «Структура односвязный список»

Таблица 2 - Листинг программы

|  |  |
| --- | --- |
| Файл | Код |
| singly\_list.c | #include "singly\_list.h"  #include <stdio.h>  #include <stdlib.h>  node\* HEAD;  node\* TAIL;  //-------------------- CHECK FUNCTION --------------------  int is\_empty(node\* list\_copy)  {  return list\_copy == NULL;  }  int count(node\* list\_copy)  {  int x = 0;  //обход всего списока  for (; list\_copy != NULL; list\_copy = list\_copy->next)  {  x++;  }  return x;  }  //-------------------- PUSH FUNCTION --------------------  void push\_to\_head(node\*\* list, void\* data)  {  node\* tmp = (node\*)malloc(sizeof(node));  tmp->data = data;  tmp->next = \*list;  \*list = tmp;  HEAD = tmp;  }  void push\_to\_tail(node\* list\_copy, void\* data)  {  if (list\_copy == NULL)  {  push\_to\_head(&list\_copy, data);  TAIL = HEAD;  return;  }  node\* tmp = (node\*)malloc(sizeof(node));  tmp->data = data;  tmp->next = NULL;  while ((list\_copy->next) != NULL)  {  list\_copy = list\_copy->next;  }  list\_copy->next = tmp;  TAIL = list\_copy;  }  //-------------------- POP FUNCTION --------------------  void\* pop\_from\_head(node\*\* list)  {  if (\*list == NULL)  {  printf("Element doesn't exist. List is empty.\n");  return NULL;  }  node\* tmp = \*list;  void\* res = tmp->data;  \*list = tmp->next;  free(tmp);  return res;  }  void\* pop\_from\_tail(node\* list\_copy)  {  if (list\_copy == NULL)  {  printf("Element doesn't exist. List is empty.\n");  return NULL;  }  node\* tmp = list\_copy;  while (list\_copy->next != NULL)  {  tmp = list\_copy;  list\_copy = list\_copy->next;  }  void\* res = list\_copy->data;  tmp->next = NULL;  free(list\_copy);  return res;  }  //-------------------- PRINT FUNCTION --------------------  void print\_from\_head(node\* list\_copy)  {  if (list\_copy == NULL)  {  printf("List is empty.\n");  return;  }  printf("Print from HEAD: ");  while (list\_copy != NULL)  {  printf("%d ", list\_copy->data);  list\_copy = list\_copy->next;  }  printf("\n");  }  void real\_print\_from\_tail(node\* list\_copy)  {  if (list\_copy != NULL)  {  real\_print\_from\_tail(list\_copy->next);  printf("%d ", list\_copy->data);  }  }  void print\_from\_tail(node\* list\_copy)  {  if (list\_copy == NULL)  {  printf("List is empty.\n");  return;  }  printf("Print from TAIL: ");  real\_print\_from\_tail(list\_copy);  printf("\n");  }  //-------------------- EDIT FUNCTION --------------------  void edit\_from\_head(node\* list\_copy, void\* data)  {  if (list\_copy == NULL)  {  printf("Element to edit doesn't exist. List is empty.\n");  return;  }  list\_copy->data = data;  }  void edit\_from\_tail(node\* list\_copy, void\* data)  {  if (list\_copy == NULL)  {  printf("Element to edit doesn't exist. List is empty.\n");  return;  }  while (list\_copy->next != NULL)  {  list\_copy = list\_copy->next;  }  list\_copy->data = data;  }  void edit\_from\_position(node\* list\_copy, int position, void\* data)  {  if (list\_copy == NULL)  {  printf("Element to edit doesn't exist. List is empty.\n");  return;  }  if (position < 1 || position > count(list\_copy))  {  printf("Impossible position for the element.\n");  return;  }  for (int i = 1; i < position; i++, list\_copy = (list\_copy)->next); //переход к запрашиваемому элементу  list\_copy->data = data;  } |
| singly\_list.h | #pragma one  //-------------------- STRUCTURE --------------------  typedef struct node  {  void\* data;  struct node\* next;  } node;  extern node\* HEAD;  extern node\* TAIL;  void push\_to\_head(node\*\*, void\*);  void push\_to\_tail(node\*, void\*);  void\* pop\_from\_head(node\*\*);  void\* pop\_from\_tail(node\*);  void print\_from\_head(node\*);  void print\_from\_tail(node\*);  void edit\_from\_head(node\*, void\*);  void edit\_from\_tail(node\*, void\*);  void edit\_from\_position(node\*, int, void\*); |

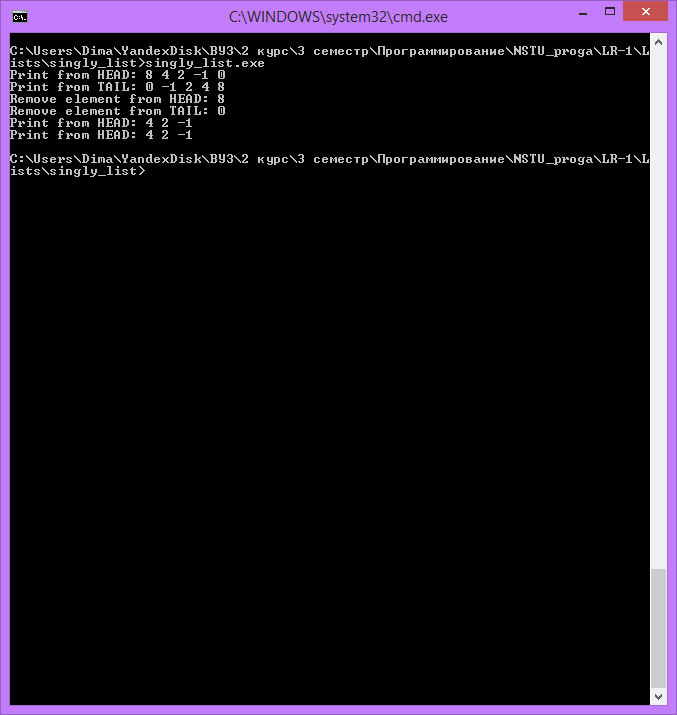


Рисунок 5 - Результат работы программы

* 1. Двусвязный список

Принципиальным отличием двусвязного списка, от односвязного заключается в том, что в узле, вместе с данными хранятся указатели на следующий и предыдущий элементы (Рисунок 5). Что несколько упрощает работу со списком, так как теперь, если мы хотим обратиться к предыдущему элементы, нам не нужно пробегать весь список с головы.

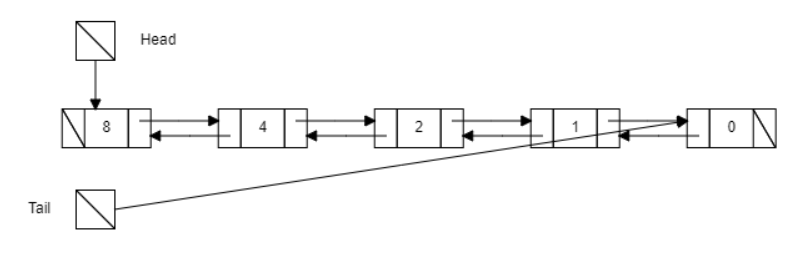


Рисунок 6 – Структура «двусвязный список»

Таблица 3 - Листинг программы

|  |  |
| --- | --- |
| Файл | Код |
| doubly\_list.c | #include "doubly\_list.h"  #include <stdlib.h>  #include <stdio.h>  node\* HEAD = NULL;  node\* TAIL = NULL;  //-------------------- CHECK FUNCTION --------------------  int is\_empty(node\* list\_copy)  {  return list\_copy == NULL;  }  //-------------------- NUMBER OF NODES --------------------  int count(node\* list\_copy) {  int i = 0;  for (; list\_copy != NULL; list\_copy = list\_copy->next)  {  i++;  }  return i;  }  //-------------------- PUSH FUNCTION --------------------  void push\_to\_head(node\*\* list, void\* data)  {  node\* tmp = (node\*)malloc(sizeof(node));  tmp->data = data;  tmp->next = \*list;  tmp->previous = NULL;  if (\*list != NULL)  {  tmp->next->previous = tmp;  }  else  {  TAIL = tmp;  }  \*list = tmp;  HEAD = tmp;  }  void push\_to\_tail(node\* list\_copy, int data)  {  node\* tmp = (node\*)malloc(sizeof(node));  tmp->data = data;  tmp->next = NULL;  if (TAIL == NULL)  {  TAIL = tmp;  }  else  {  tmp->previous = TAIL;  tmp->previous->next = tmp;  TAIL = tmp;  }  }  void push\_to\_position(node\*\* list, int position, int data)  {  if (position < 1 || position > count(\*list) + 1)  {  printf("Impossible position for the element\n");  return;  }  if (position == 1)  {  push\_to\_head(list, data);  return;  }  if (position == count(\*list) + 1)  {  push\_to\_tail(\*list, data);  return;  }  node\* tmp = (node\*)malloc(sizeof(node));  node\* list\_copy = \*list;  tmp->data = data;  for (int i = 1; i < position - 1; i++, list\_copy = list\_copy->next);  tmp->next = list\_copy->next;  list\_copy->next = tmp;  }  //-------------------- POP FUNCTION --------------------  int pop\_from\_head(node\*\* list)  {  if (\*list == NULL)  {  printf("Element to remove doesn't exist. List is empty.\n");  return NULL;  }  node\* tmp = \*list;  int res = tmp->data;  \*list = tmp->next;  free(tmp);  return res;  }  int pop\_from\_tail(node\* list\_copy)  {  if (list\_copy == NULL)  {  printf("Element to remove doesn't exist. List is empty.\n");  return NULL;  }  list\_copy = TAIL;  node\* tmp = list\_copy->previous;  tmp->next = NULL;  TAIL = tmp;  int res = list\_copy->data;  free(list\_copy);  return res;  }  int pop\_from\_position(node\*\* list, int position)  {  if (\*list == NULL)  {  printf("Element to remove doesn't exist. List is empty.\n");  return NULL;  }  if (position < 1 || position > count(\*list))  {  printf("Impossible position for the element\n");  return NULL;  }  if (position == 1)  {  return pop\_from\_head(list);  }  if (position == count(\*list))  {  return pop\_from\_tail(\*list);  }  node\* list\_copy = \*list;  node\* tmp = list\_copy;  for (int i = 1; i < position; i++, list\_copy = (list\_copy)->next)  {  tmp = list\_copy;  }  int res = list\_copy->data;  tmp->next = list\_copy->next;  free(list\_copy);  return res;  }  //-------------------- EDIT FUNCTION --------------------  int edit\_from\_head(node\* list\_copy, int data)  {  if (list\_copy == NULL)  {  printf("Element to edit doesn't exist. List is empty.\n");  return NULL;  }  int res = list\_copy->data;  list\_copy->data = data;  return res;  }  int edit\_from\_tail(node\* list\_copy, int data)  {  if (list\_copy == NULL)  {  printf("Element to edit doesn't exist. List is empty.\n");  return NULL;  }  list\_copy = TAIL;  int res = list\_copy->data;  list\_copy->data = data;  return res;  }  int edit\_from\_position(node\* list\_copy, int position, int data)  {  if (list\_copy == NULL)  {  printf("Element to edit doesn't exist. List is empty.\n");  return NULL;  }  if (position < 1 || position > count(list\_copy))  {  printf("Impossible position for the element.\n");  return NULL;  }  for (int i = 1; i < position; i++, list\_copy = (list\_copy)->next);  int res = list\_copy->data;  list\_copy->data = data;  return res;  }  //-------------------- PRINT FUNCTION --------------------  void print\_from\_head(node\* list\_copy)  {  if (list\_copy == NULL)  {  printf("List is empty.\n");  return;  }  printf("Print from HEAD: ");  while (list\_copy != NULL)  {  printf("%d ", list\_copy->data);  list\_copy = list\_copy->next;  }  printf("\n");  }  void print\_from\_tail(node\* list\_copy)  {  if (list\_copy == NULL)  {  printf("List is empty.\n");  return;  }  list\_copy = TAIL;  printf("Print from TAIL: ");  while (list\_copy != NULL)  {  printf("%d ", list\_copy->data);  list\_copy = list\_copy->previous;  }    printf("\n");  }  void print\_element(node\* list\_copy, int position)  {  if (list\_copy == NULL)  {  printf("List is empty.\n");  return;  }  if (position < 1 || position > count(list\_copy))  {  printf("Impossible position for the element.\n");  return;  }  for (int i = 1; i < position; i++, list\_copy = (list\_copy)->next);  printf("Print %d element: %d", position, list\_copy->data);  printf("\n");  } |
| doubly\_list.h | #pragma once  typedef struct node  {  void\* data;  struct node\* previous;  struct node\* next;  } node;  void push\_to\_head(node\*\*, void\*);  void push\_to\_tail(node\*, int);  void push\_to\_position(node\*\*, int, int);  int pop\_from\_head(node\*\*);  int pop\_from\_tail(node\*);  int pop\_from\_position(node\*\* , int);  int edit\_from\_head(node\*, int);  int edit\_from\_tail(node\*, int);  int edit\_from\_position(node\*, int, int);  void print\_from\_head(node\*);  void print\_from\_tail(node\*);  void print\_element(node\*, int); |

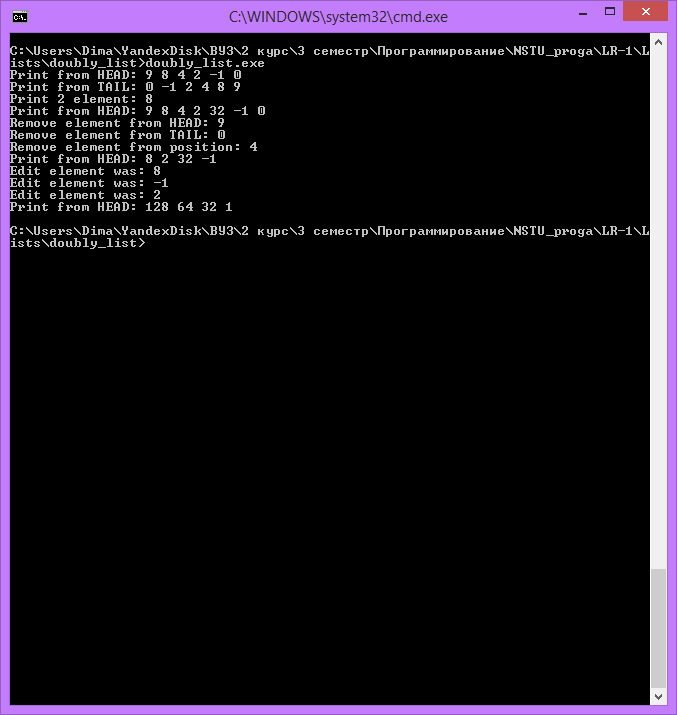


Рисунок 7 - Результат работы программы

1. Реализовать структуру «Очередь»

Операция добавления элемента: O(1)

Операция удаления элемента: O(1)

Операция обращения к элементу: -

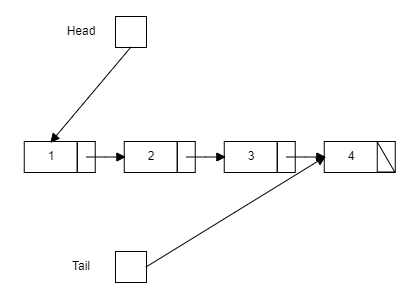


Рисунок 8 – Структура «очередь»

Представляет собой список элементов, организованных по принципу FIFO (англ. first in — first out, «первым пришёл — первым вышел»).

В данной работе структура очередь организована на основе односвязных списков. Для записи в очередь используется функция записи в хвост, а для чтения используется функция удаления из головы с возвращением данных удаляемого узла.

Таблица 4 - Листинг программы

|  |  |
| --- | --- |
| Файл | Код |
| queue.c | #include "queue.h"  #include <stdio.h>  #include <stdlib.h>  node\* HEAD = NULL;  node\* TAIL = NULL;  //-------------------- CHECK FUNCTION --------------------  int is\_empty(node\* list\_copy)  {  return list\_copy == NULL;  }  int count(node\* list\_copy)  {  int x = 0;  //обход всего списока  for (; list\_copy != NULL; list\_copy = list\_copy->next)  {  x++;  }  return x;  }  //-------------------- PUSH FUNCTION --------------------  void push\_to\_head(node\*\* list, void\* data)  {  node\* tmp = (node\*)malloc(sizeof(node));  tmp->data = data;  tmp->next = \*list;  \*list = tmp;  HEAD = tmp;  }  void push\_to\_tail(node\* list\_copy, void\* data)  {  if (list\_copy == NULL)  {  push\_to\_head(&list\_copy, data);  TAIL = HEAD;  return;  }  node\* tmp = (node\*)malloc(sizeof(node));  tmp->data = data;  tmp->next = NULL;  while ((list\_copy->next) != NULL)  {  list\_copy = list\_copy->next;  }  list\_copy->next = tmp;  TAIL = list\_copy;  }  //-------------------- POP FUNCTION --------------------  void\* pop\_from\_head(node\*\* list)  {  if (\*list == NULL)  {  printf("Element doesn't exist. List is empty.\n");  return NULL;  }  node\* tmp = \*list;  void\* res = tmp->data;  \*list = tmp->next;  free(tmp);  return res;  }  void push\_queue(void\* data)  {  push\_to\_tail(TAIL, data);  }  void pop\_queue()  {  printf("%s\n", pop\_from\_head(&HEAD));  } |
| queue.h | #pragma once  typedef struct node  {  void\* data;  struct node\* next;  } node;  void push\_queue(void\*);  void pop\_queue(); |

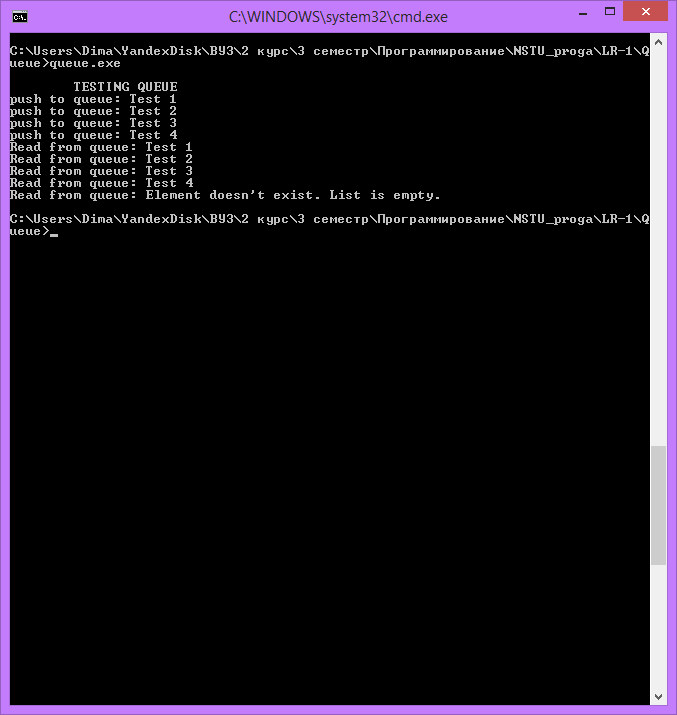


Рисунок 9 - Результат работы программы

1. Реализовать структуру «Стек»

Операция добавления элемента: O(1)

Операция удаления элемента: O(1)

Операция обращения к элементу: -

Представляет собой список элементов, организованных по принципу LIFO (англ. last in — first out, «последним пришёл — первым вышел»).

В данной работе структура стек организована на основе односвязных списков. Для записи в стек используется функция записи в голову, а для чтения используется функция удаления из головы с возвращением данных удаляемого узла.

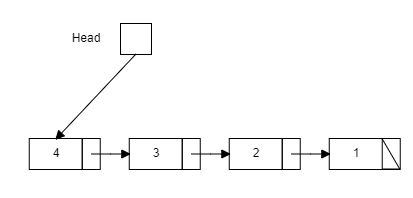


Рисунок 10 – Структура «стек»

Таблица 5 - Листинг программы

|  |  |
| --- | --- |
| Файл | Код |
| stack.c | #include "stack.h"  #include <stdio.h>  #include <stdlib.h>  node\* HEAD;  node\* TAIL;  //-------------------- PUSH FUNCTION --------------------  void push\_to\_head(node\*\* list, void\* data)  {  node\* tmp = (node\*)malloc(sizeof(node));  tmp->data = data;  tmp->next = \*list;  \*list = tmp;  HEAD = tmp;  }  //-------------------- POP FUNCTION --------------------  void\* pop\_from\_head(node\*\* list)  {  if (\*list == NULL)  {  printf("Element doesn't exist. List is empty.\n");  return NULL;  }  node\* tmp = \*list;  void\* res = tmp->data;  \*list = tmp->next;  free(tmp);  return res;  }  void push\_stack(void\* data)  {  push\_to\_head(&HEAD, data);  }  void pop\_stack()  {  printf("%s\n", pop\_from\_head(&HEAD));  } |
| stack.h | #pragma once  //-------------------- STRUCTURE --------------------  typedef struct node  {  void\* data;  struct node\* next;  } node;  void push\_stack(void\*);  void pop\_stack(); |

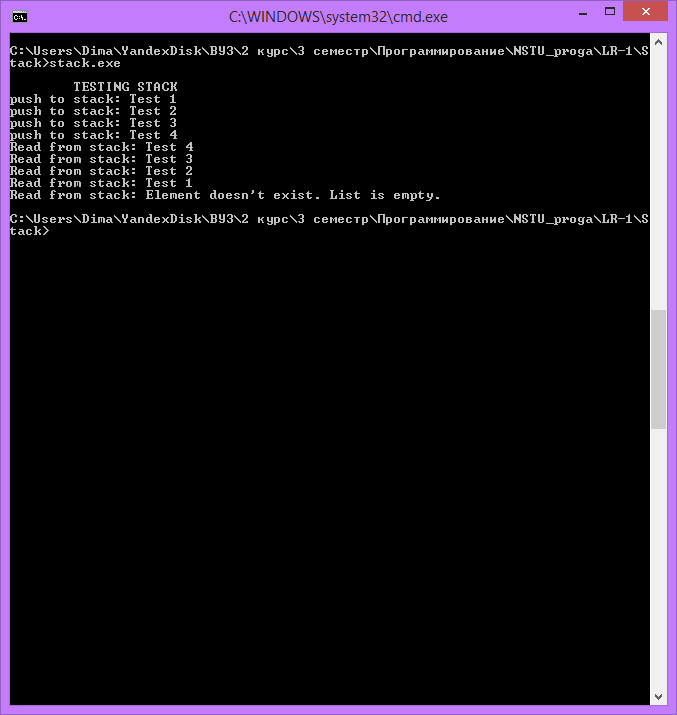


Рисунок 11 - Результат работы программы

1. Реализовать структуру «Хеш таблица»

Операция добавления элемента: O(1)

Операция удаления элемента: O(1)

Операция обращения к элементу: O(1)

Хеш-таблица — это список элементов, которые сами по себе являются парой {ключ: значение}. Хеш-таблица хранит в себе массив указателей, которые ведут к элементам таблицы.

Принцип работы хеш-таблицы основан на использовании функции хеширования для преобразования ключа в индекс ячейки, где хранится соответствующая пара (ключ, значение). Если хеш-функция генерирует коллизию (два разных ключа имеют одинаковый хеш), то используются техники разрешения коллизий, такие как цепочечная схема, квадратичное probing, двойное хеширование и другие.

В рамках данной лабораторной работы был реализован способ разрешения коллизии «цепочечная схема», то есть если две или более записей оказываются в одной и той же ячейке (коллизия), то они хранятся в виде связного списка. При поиске элемента хеш-функция сначала определяет индекс ячейки, а затем просматривает всю цепочку в этой ячейке, пока не найдет нужный элемент. Однако, если коллизии становятся слишком частыми, это может привести к снижению производительности из-за увеличения времени поиска элементов.

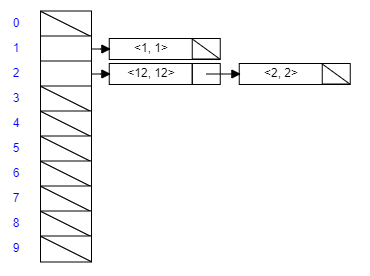
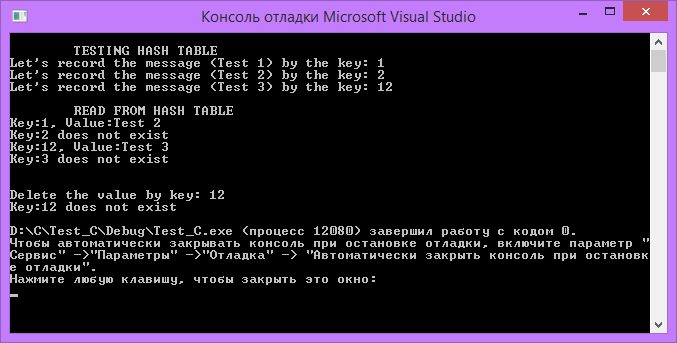


Рисунок 12 – Структура «хеш-таблица»

Таблица 6 - Листинг программы

|  |  |
| --- | --- |
| Файл | Код |
| hash\_table.c | #include "hash\_table.h"  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #define CAPACITY 10 // Size of the Hash Table  List\* HEAD;  List\* TAIL;  //-----------------HASH FUNCK-------------------------------  unsigned long hash\_function(char\* str)  {  unsigned long i = 0;  for (int j = 0; str[j]; j++)  {  i += str[j];  }  return i % CAPACITY;  }  //-----------------CREATE HASH TABLE--------------------  node\* create\_item(char\* key, char\* value)  {  node\* item = (node\*)malloc(sizeof(node));  item->key = (char\*)malloc(strlen(key) + 1);  item->value = (char\*)malloc(strlen(value) + 1);  strcpy(item->key, key);  strcpy(item->value, value);  return item;  }  List\*\* create\_overflow(HT\* table)  {  List\*\* my\_list = (List\*\*)calloc(table->size, sizeof(List\*));  for (int i = 0; i < table->size; i++)  {  my\_list[i] = NULL;  }  return my\_list;  }  HT\* create\_table(int size)  {  HT\* table = (HT\*)malloc(sizeof(HT));  table->size = size;  table->count = 0;  table->items = (node\*\*)calloc(table->size, sizeof(node\*));  for (int i = 0; i < table->size; i++)  {  table->items[i] = NULL;  }  table->overflow = create\_overflow(table);  return table;  }  void free\_list(List\* list)  {  List\* temp = list;  while (list != NULL)  {  temp = list;  list = list->next;  free(temp->node->key);  free(temp->node->value);  free(temp->node);  free(temp);  }  }  void free\_overflow(HT\* table)  {  List\*\* my\_list = table->overflow;  for (int i = 0; i < table->size; i++)  {  free\_list(my\_list[i]);  }  free(my\_list);  }  void free\_item(node\* item)  {  // Frees an item  free(item->key);  free(item->value);  free(item);  }  void free\_table(HT\* table)  {  // Frees the table  for (int i = 0; i < table->size; i++)  {  node\* item = table->items[i];  if (item != NULL)  {  free\_item(item);  }  }  free\_overflow(table);  free(table->items);  free(table);  }  List\* list\_insert(List\* list, node\* item)  {  if (list == NULL)  {  list = (List\*)malloc(sizeof(List));  list->node = item;  list->next = NULL;  return list;  }  else if(list->next == NULL)  {  List\* tmp = (List\*)malloc(sizeof(List));  tmp->node = item;  tmp->next = NULL;  list->next = tmp;  return list;  }  else  {  while (list->next != NULL)  {  list = list->next;  }  List\* tmp = (List\*)malloc(sizeof(List));  tmp->node = item;  tmp->next = NULL;  list->next = tmp;  return list;  }  }  void handle\_collision(HT\* table, unsigned long index, node\* item)  {  List\* list = table->overflow[index];  if (list == NULL)  {  // We need to create the list  list = (List\*)malloc(sizeof(List));  list->node = item;  list->next = NULL;  table->overflow[index] = list;  return;  }  else {  // Insert to the list  table->overflow[index] = list\_insert(list, item);  return;  }  }  //----------------------PUSH TO HASH TABLE-------------------  void ht\_insert(HT\* table, char\* key, char\* value)  {  if (table == NULL)  {  return;  }  // Create the item  node\* item = create\_item(key, value);  int index = hash\_function(key);  node\* current\_item = table->items[index];  if (current\_item == NULL)  {  // Key does not exist.  if (table->count == table->size)  {  printf("Insert Error: Hash Table is full\n");  return;  }  table->items[index] = item;  table->count++;  }  else  {  if (strcmp(current\_item->key, key) == 0)  {  strcpy(current\_item->value, value);  //strcpy(table->items[index], current\_item->value);  return;  }  else  {  handle\_collision(table, index, item);  return;  }  }  }  char\* ht\_search(HT\* table, char\* key)  {  int index = hash\_function(key);  node\* item = table->items[index];  List\* list = table->overflow[index];  while (item != NULL)  {  if (strcmp(item->key, key) == 0)  {  return item->value;  }  if (list == NULL)  {  return NULL;  }  item = list->node;  list = list->next;  }  return NULL;  }  void print\_search(HT\* table, char\* key)  {  char\* val;  if ((val = ht\_search(table, key)) == NULL)  {  printf("Key:%s does not exist\n", key);  return;  }  else  {  printf("Key:%s, Value:%s\n", key, val);  }  }  void print\_table(HT\* table)  {  printf("\nHash Table\n-------------------\n");  for (int i = 0; i < table->size; i++)  {  if (table->items[i])  {  node\* tmp = table->items[i];  printf("Index:%d, Key:%s, Value:%s\n", i, tmp->key, tmp->value);  }  }  printf("-------------------\n\n");  }  void ht\_delete(HT\* table, char\* key)  {  // Deletes an item from the table  int index = hash\_function(key);  node\* item = table->items[index];  List\* head = table->overflow[index];  if (item == NULL)  {  return;  }  else  {  if (head == NULL && strcmp(item->key, key) == 0)  {  table->items[index] = NULL;  free\_item(item);  table->count--;  return;  }  else if (head != NULL)  {  // Collision Chain exists  if (strcmp(item->key, key) == 0)  {  free\_item(item);  List\* list = head;  head = head->next;  list->next = NULL;  table->items[index] = create\_item(list->node->key, list->node->value);  free\_list(list);  table->overflow[index] = head;  return;  }  List\* curr = head;  List\* prev = NULL;  while (curr)  {  if (strcmp(curr->node->key, key) == 0)  {  if (prev == NULL)  {  // First element of the chain. Remove the chain  free\_list(head);  table->overflow[index] = NULL;  return;  }  else  {  // This is somewhere in the chain  prev->next = curr->next;  curr->next = NULL;  free\_list(curr);  table->overflow[index] = head;  return;  }  }  curr = curr->next;  prev = curr;  }  }  }  } |
| hash\_table.h | #pragma once  typedef struct node {  char\* key;  void\* value;  } node;  typedef struct HT {  struct HT\*\* items;  struct List\*\* overflow;  int size;  int count;  }HT;  typedef struct List {  node\* node;  struct List\* next;    } List;  void ht\_insert(HT\*, char\* , char\*);  void print\_search(HT\*, char\*);  void print\_table(HT\*);  void ht\_delete(HT\*, char\*);  void free\_table(HT\*); |

  
Рисунок 13 - Результат работы программы

1. Реализовать структуру «Full Binary Tree»

Операция добавления элемента: O(log n)

Операция удаления элемента: O(log n)

Операция обращения к элементу: O(log n)

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение и ссылки на левого и правого потомка. Принцип выбора расположения потомка заключается в следующем: левым потомком является элемент меньше родительского, справа элемент больше родительского.

Существует проблема, если записывать в дерево упорядоченные данные, то оно превращается в связный список. Для решения этой проблемы используется балансировка бинарного дерева. Балансировка применяется для того, чтобы уравновесить плечи бинарного дерева, что обеспечит поиск элемента O(log n).

Full Binary Tree – это особый вид бинарного дерева, в котором каждый узел имеет либо левого и правого ребенка, либо ни одного из них.

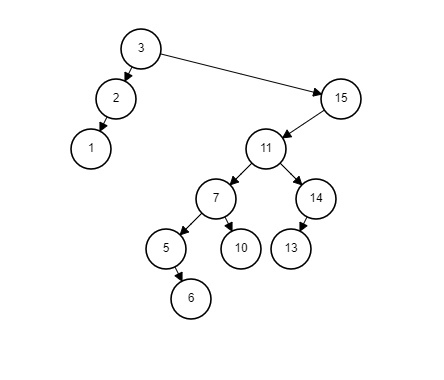


Рисунок 14 - Структура «бинарное дерево»

Таблица 7 - Листинг программы

|  |  |
| --- | --- |
| Файл | Код |
| trees.c | #include <stdio.h>  #include <stdlib.h>  #include <locale.h>  #include "trees.h"  #define RUS setlocale(LC\_ALL, "RU");  node\* ROOT;  node\* search\_the\_node(node\* tree, int key)  {  if (tree == NULL)  {  return NULL;  }  if (tree->key == key)  {  return tree;  }  else if(key < tree->key)  {  return search\_the\_node(tree->left, key);  }  else  {  return search\_the\_node(tree->right, key);  }  }  node\* finde\_parend(node\* parent, int key)  {  if (key < parent->key)  {  if (parent->left != NULL)  {  return finde\_parend(parent->left, key);  }  else  {  return parent;  }  }  else  {  if (parent->right != NULL)  {  return finde\_parend(parent->right, key);  }  else  {  return parent;  }  }  }  void create\_node(int key, void\* data)  {  node\* tree = NULL;  if ((tree = (node\*)malloc(sizeof(node))))  {  tree->data = data;  tree->key = key;  tree->left = tree->right = NULL;  if (ROOT != NULL)  {  node\* parent = finde\_parend(ROOT, key);  tree->parent = parent;  if (key < parent->key)  {  parent->left = tree;  }  else  {  parent->right = tree;  }  }  else  {  tree->left = tree->right = tree->parent = NULL;  ROOT = tree;  }  }  else  {  printf("Íåäîñòàòî÷íî ïàìÿòè. Íåâîçìîæíî ñîçäàòü óçåë.\n");  }  }  node\* get\_max(node\* tree)  {  RUS  if (tree == NULL)  {  printf("Äåðåâà íå ñóùåñòâóåò!\n");  return NULL;  }  if (tree->right == NULL)  {  return tree;  }  return(get\_max(tree->right));  }  node\* get\_min(node\* tree)  {  RUS  if (tree == NULL)  {  printf("Äåðåâà íå ñóùåñòâóåò!\n");  return NULL;  }  if (tree->left == NULL)  {  return tree;  }  return(get\_max(tree->left));  }  void showLine(char\* c, int p, int s) {  int t = s;  for (int i = 0; i < p; i++)  {  printf(t & 1 ? "| " : " ");  t /= 2;  }  printf(c);  }  void showTree(node\* tree, int p, int s)  {  if (tree == NULL)  {  return;  }  printf("%d", tree->data);  printf("\n");  if (tree->left != NULL) {  showLine("|\n", p, s);  showLine("L: ", p, s);  showTree(tree->left, p + 1, s + ((tree->right == NULL ? 0 : 1) << p));  }  if (tree->right != NULL) {  showLine("|\n", p, s);  showLine("R: ", p, s);  showTree(tree->right, p + 1, s);  }  }  void printTree(node\* tree)  {  showTree(tree, 0, 0);  }  void delete\_node(int key)  {  node\* root = ROOT;  root = search\_the\_node(root, key);  if (root == NULL)  {  return;  }  if (root->left == NULL || root->right == NULL)  {  if (root->left == NULL)  {  node\* tmp = root->parent;  if (key < tmp->key)  {  tmp->left = root->right;  }  else  {  tmp->right = root->right;  }  tmp = root;  root = root->right;  if (tmp->right != NULL)  {  node\* that\_node = tmp->right;  that\_node->parent = tmp->parent;  }  if (tmp->left != NULL)  {  node\* that\_node = tmp->left;  that\_node->parent = tmp->parent;  }  free(tmp);  }  else  {  node\* tmp = root->parent;  if (key < tmp->key)  {  tmp->left = root->left;  }  else  {  tmp->right = root->left;  }  tmp = root;  root = root->left;  if (tmp->right != NULL)  {  node\* that\_node = tmp->right;  that\_node->parent = tmp->parent;  }  if (tmp->left != NULL)  {  node\* that\_node = tmp->left;  that\_node->parent = tmp->parent;  }  free(tmp);  }  }  else if (root->left == NULL && root->right == NULL)  {  node\* tmp = root->parent;  if (key < tmp->key)  {  tmp->left = NULL;  free(root);  }  }  else  {  node\* receiver = root->right;  if (receiver->left == NULL)  {  root->data = receiver->data;  root->key = receiver->key;  root->right = receiver->right;  root->level = receiver->level;  root->parent = receiver->parent;  return;  }  else  {  receiver = receiver->left;  root->data = receiver->data;  root->key = receiver->key;  root->right = receiver->parent;  root->level = receiver->level - 1;  receiver = receiver->parent;  receiver->left = NULL;  return;  }  }  }  int max\_level\_node(node\* tree, int level)  {  if (tree)  {  int new\_level\_1 = max\_level\_node(tree->left, level + 1);  int new\_level\_2 = max\_level\_node(tree->right, level + 1);  if (new\_level\_1 > level) level = new\_level\_1;  if (new\_level\_2 > level) level = new\_level\_2;  }  return level;  }  int max\_level(node\* tree)  {  return max\_level\_node(tree, -1);  }  void balance\_tree(node\*\* tree)  {  if (\*tree)  {  while (max\_level((\*tree)->left) - max\_level((\*tree)->right) >= 2)  {  node\* tmp = (\*tree)->left;  (\*tree)->left = tmp->right;  tmp->right = \*tree;  \*tree = tmp;  }  while (max\_level((\*tree)->right) - max\_level((\*tree)->left) >= 2)  {  node\* tmp = (\*tree)->right;  (\*tree)->right = tmp->left;  tmp->left = \*tree;  \*tree = tmp;  }  balance\_tree(&((\*tree)->left));  balance\_tree(&((\*tree)->right));  }  }  void balance()  {  if (ROOT == NULL)  {  return;  }  balance\_tree(&ROOT);  balance\_tree(&ROOT);  }  bool isFullBinaryTree(node\* root)  {  if (root == NULL)  {  return true;  }  if (root->left == NULL && root->right == NULL)  {  return true;  }  if ((root->left) && (root->right))  {  return (isFullBinaryTree(root->left) && isFullBinaryTree(root->right));  }  return false;  } |
| trees.h | #pragma once  #include <stdbool.h>  typedef struct node  {  int key;  void\* data;  struct node\* right;  struct node\* left;  struct node\* parent;  unsigned char level;  } node;  extern node\* ROOT;  node\* search\_the\_node(node\* tree, int key);  void add\_the\_node(node\* tree, int key, int data);  void create\_node(int key, void\* data);  void printTree(node\* tree);  node\* finde\_parend(node\* parent, int key);  void delete\_node(int key);  void balance(); |

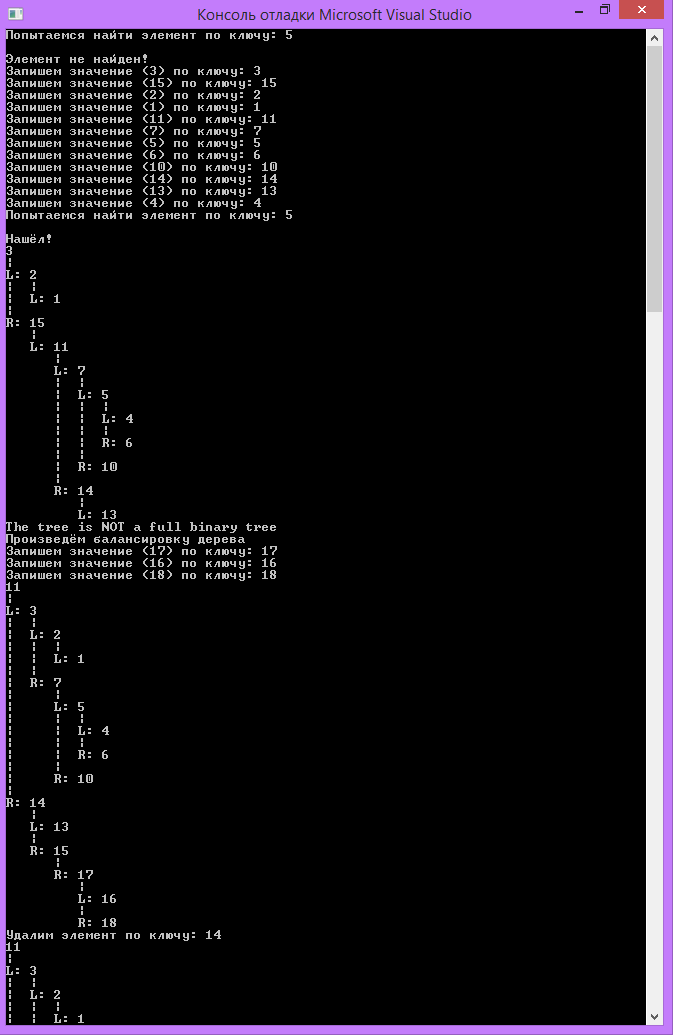


Рисунок 15 - Результат работы программы

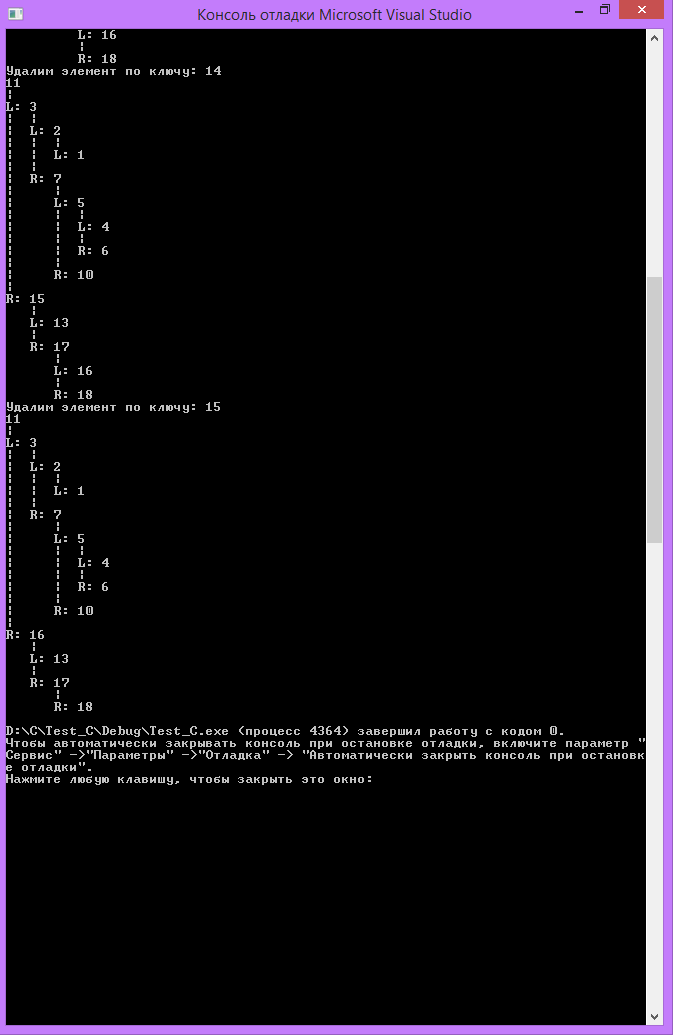


Рисунок 16 - Результат работы программы

Вывод: В ходе выполнения работы были изучены различные структуры данных и их основные свойства. Массивы являются самым простым и быстрым способом хранения данных, но они ограничены размером и не позволяют добавлять или удалять элементы без перестройки всего массива. Списки позволяют хранить данные в любом порядке и легко добавлять и удалять элементы, но при обращении к конкретным элементам эффективность списков ниже, чем у массивов. Деревья позволяют хранить данные в иерархической структуре, что может быть полезно для организации больших объемов данных. Они также позволяют быстро выполнять различные операции, такие как поиск данных. Однако деревья требуют больше памяти и времени на обработку, чем массивы и списки. Хеш-таблицы и связанные списки являются эффективными структурами данных для хранения и поиска элементов в больших наборах данных. Они обеспечивают быстрый доступ к данным и позволяют обрабатывать большие объемы информации. Но как и деревья очень требовательны к объёму памяти и времени на обработку данных.