МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

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

Кафедра защиты информации

**

**ОТЧЁТ**

**по практической работе №1**

**«**Разработка СУБД**»**

**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.

СУБД(dataBase.out) запускается с аргументами: command, data, key, и предназначена для работы в качестве подпрограммы. Для взаимодействия с СУБД был написан консольный интерфейс(program).

Таблица 1 – Код программы

|  |  |
| --- | --- |
| main.c | #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <sys/types.h>  #define \_SIZE\_ 25  void call\_dataBase(char\* command, char\* data, char\* key)  {  pid\_t pid = -2;  switch(pid = fork())  {  case -1:  printf("ERROR: fork()");  break;  case 0:  char\* path = "/home/dmitriy/NSTU\_proga/PR-1/dataBase.out\0";  execl(path, command, data, key, NULL);  break;  default:  break;  }  }  int main()  {  char command[\_SIZE\_];  printf("Enter the command: ");  scanf("%s", command);  char data[\_SIZE\_];  char key[\_SIZE\_];  if (strcmp(command, "SADD") == 0)  {  printf("Enter the data: ");  scanf("%s", data);  strcpy(key, "key");  }  else if (strcmp(command, "SPUSH") == 0)  {  printf("Enter the data: ");  scanf("%s", data);  strcpy(key, "key");  }  else if (strcmp(command, "QPUSH") == 0)  {  printf("Enter the data: ");  scanf("%s", data);  strcpy(key, "key");  }  else if (strcmp(command, "HSET") == 0)  {  printf("Enter the data: ");  scanf("%s", data);  printf("Enter the key: ");  scanf("%s", key);  }  else if (strcmp(command, "SREM") == 0)  {  printf("Enter the key: ");  scanf("%s", key);  strcpy(data, "data");  }  else if (strcmp(command, "SPOP") == 0)  {  strcpy(data, "data");  strcpy(key, "key");  }  else if (strcmp(command, "QPOP") == 0)  {  strcpy(data, "data");  strcpy(key, "key");  }  else if (strcmp(command, "HDEL") == 0)  {  printf("Enter the key: ");  scanf("%s", key);  strcpy(data, "data");  }  else if (strcmp(command, "SISMEMBER") == 0)  {  printf("Enter the key: ");  scanf("%s", key);  }  else if (strcmp(command, "HGET") == 0)  {  printf("Enter the key: ");  scanf("%s", key);  }  else  {  printf("ERROR: incorrect command\n");  }  call\_dataBase(command, data, key);  } |
| programm.c | #include "queue.h"  #include "stack.h"  #include "set.h"  #include "hash\_table.h"  #include "file.c"  #include <stdio.h>  #include <string.h>  #define \_SIZE\_ 15  #define \_PATH\_ "data.txt"  void writer(HT\* table, SET\* my\_set)  {  clear\_file(\_PATH\_);  char\* data = pop\_queue();  while(strcmp(data, "List\_is\_empty.\n"))  {  write\_to\_file(\_PATH\_, "Queue", data);  data = pop\_queue();  }  data = pop\_stack();  while(strcmp(data, "List\_is\_empty.\n"))  {  write\_to\_file(\_PATH\_, "Stack", data);  data = pop\_stack();  }  save\_set(my\_set, \_PATH\_);  save\_table(table, \_PATH\_);  }  int main(int argc, char\* argv[])  {  char\* command = strdup(argv[0]);  char\* data = strdup(argv[1]);  char\* key = strdup(argv[2]);  HT\* table = create\_table(\_SIZE\_);  SET\* my\_set = create\_set(\_SIZE\_);  read\_file(\_PATH\_, "Queue");  read\_file(\_PATH\_, "Stack");  read\_set\_file(\_PATH\_, my\_set, "Set");  read\_ht\_file(\_PATH\_, table, "HT");  if (strcmp(command, "SADD") == 0)  {  printf("\nYour data: %s\n", data);  set\_insert(my\_set, data, "");  }  else if (strcmp(command, "SPUSH") == 0)  {  printf("\nYour data: %s\n", data);  push\_stack(data);  }  else if (strcmp(command, "QPUSH") == 0)  {  printf("\nYour data: %s\n", data);  push\_queue(data);  }  else if (strcmp(command, "HSET") == 0)  {  printf("\nYour key: %s data: %s\n", key, data);  ht\_insert(table, key, data);  }  else if (strcmp(command, "SREM") == 0)  {  printf("\nYour key: %s\n", key);  set\_delete(table, key);  key = strcat(key, "\n");  }  else if (strcmp(command, "SPOP") == 0)  {  printf("%s\n", pop\_stack());  }  else if (strcmp(command, "QPOP") == 0)  {  printf("%s\n", pop\_queue());  }  else if (strcmp(command, "HDEL") == 0)  {  printf("\nYour key: %s\n", key);  ht\_delete(table, key);  }  else if (strcmp(command, "SISMEMBER") == 0)  {  char\* tmp;  key = strcat(key, "\n");  if((tmp = set\_search(my\_set, key)) == NULL)  {  printf("---NULL---\n");  }  else  {  printf("Exist\n");  }  }  else if (strcmp(command, "HGET") == 0)  {  printf("\nYour key: %s\n", key);  print\_search(table, key);  }  else  {  printf("ERROR: incorrect command\n");  }  writer(table, my\_set);  } |
| set.c | #include "set.h"  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #define CAPACITY 10 // Size of the Hash Table  unsigned long hash(char\* str)  {  unsigned long i = 0;  for (int j = 0; str[j]; j++)  {  i += str[j];  }  return i % CAPACITY;  }  node\* create\_node(char\* key, char\* value)  {  node\* item = (node\*)malloc(sizeof(node));  item->key = (char\*)malloc(strlen(key) + 1);  if (key != 0)  {  strcpy(item->key, key);  }  return item;  }  SET\* create\_set(int size)  {  SET\* table = (SET\*)malloc(sizeof(SET));  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;  }  return table;  }  void free\_node(node\* item)  {  free(item->key);  free(item);  }  void free\_set(SET\* table)  {  for (int i = 0; i < table->size; i++)  {  node\* item = table->items[i];  if (item != NULL)  {  free\_node(item);  }  }  free(table->items);  free(table);  }  void set\_insert(SET\* table, char\* key, char\* value)  {  if (table == NULL)  {  return;  }  node\* item = create\_node(key, value);  int index = hash\_function(key);  node\* current\_item = table->items[index];  if (current\_item == NULL)  {  if (table->count == table->size)  {  printf("Insert Error: Hash Table is full\n");  return;  }  table->items[index] = item;  table->count++;  }  else  {  printf("Insert Error: The element already exists\n");  return;  }  }  char\* set\_search(SET\* table, char\* key)  {  printf("\t%s\n", key);  int index = hash(key);  node\* item = table->items[index];  if (item == NULL)  {  return NULL;  }  else if (!strcmp(item->key, key))  {  return item->key;  }  printf("\t%s, %s", item->key, key);  return NULL;  }  void search\_print(SET\* table, char\* key)  {  char\* val;  if ((val = set\_search(table, key)) == NULL)  {  printf("Key:%s does not exist\n", key);  return;  }  else  {  printf("Key:%s, Value:%s\n", key, val);  }  }  void print\_set(SET\* 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\n", i, tmp->key);  }  }  printf("-------------------\n\n");  }  void save\_set(SET\* my\_set, char\* path)  {  for (int i = 0; i < my\_set->size; i++)  {  if (my\_set->items[i])  {  node\* tmp = my\_set->items[i];  write\_to\_file(path, "Set", tmp->key);  }  }  }  void set\_delete(SET\* table, char\* key)  {  int index = hash(key);  if (table->items[index] == NULL)  {  return;  }  table->items[index] = NULL;  } |
| set.h | #pragma once  typedef struct node {  char\* key;  } node;  typedef struct SET {  struct SET\*\* items;  int size;  int count;  }SET;  SET\* create\_set(int size);  void set\_insert(SET\* table, char\* key, char\* value);  char\* set\_search(SET\* table, char\* key);  void search\_print(SET\*, char\*);  void print\_set(SET\*);  void set\_delete(SET\*, char\*);  void free\_set(SET\*); |
| queue.c | #include "queue.h"  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  q\_node\* START = NULL;  q\_node\* END = NULL;  bool is\_empty()  {  return START == NULL;  }  int count(q\_node\* list\_copy)  {  int x = 0;  for (; list\_copy != NULL; list\_copy = list\_copy->next)  {  x++;  }  return x;  }  void push\_to\_start(q\_node\*\* list, char\* data)  {  q\_node\* tmp = (q\_node\*)malloc(sizeof(q\_node));  //tmp->data = data;  tmp->data = strdup(data);  tmp->next = \*list;  \*list = tmp;  }  void push\_to\_end(q\_node\* list\_copy, char\* data)  {  if (START == NULL)  {  push\_to\_start(&START, data);  END = START;  return;  }  q\_node\* tmp = (q\_node\*)malloc(sizeof(q\_node));  tmp->data = strdup(data);  tmp->next = NULL;  list\_copy->next = tmp;  END = tmp;  }  char\* pop\_from\_start(q\_node\*\* list)  {  if (\*list == NULL)  {  return "List\_is\_empty.\n";  }  q\_node\* tmp = \*list;  char\* res = tmp->data;  START = tmp->next;  \*list = tmp->next;  free(tmp);  return res;  }  void push\_queue(char\* data)  {  push\_to\_end(END, data);  }  char\* pop\_queue()  {  return pop\_from\_start(&START);  } |
| queue.h | #pragma once  #include <stdbool.h>  typedef struct q\_node  {  char\* data;  struct q\_node\* next;  } q\_node;  void push\_queue(char\*);  char\* pop\_queue();  bool is\_empty(); |
| stack.c | #include "stack.h"  #include <stdio.h>  #include <stdlib.h>  s\_node\* S\_HEAD;  s\_node\* S\_TAIL;  void push\_to\_head(s\_node\*\* list, char\* data)  {  s\_node\* tmp = (s\_node\*)malloc(sizeof(s\_node));  tmp->data = strdup(data);  tmp->next = \*list;  \*list = tmp;  }  char\* pop\_from\_head(s\_node\*\* list)  {  if (\*list == NULL)  {  return "List\_is\_empty.\n";  }  s\_node\* tmp = \*list;  char\* res = strdup(tmp->data);  \*list = tmp->next;  free(tmp);  return res;  }  void push\_stack(char\* data)  {  push\_to\_head(&S\_HEAD, data);  }  char\* pop\_stack()  {  return pop\_from\_head(&S\_HEAD);  } |
| stack.h | #pragma once  typedef struct s\_node  {  char\* data;  struct s\_node\* next;  } s\_node;  void push\_stack(char\*);  char\* pop\_stack(); |
| hash\_table.c | #include "hash\_table.h"  #include "file.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--------------------  ht\_node\* create\_item(char\* key, char\* value)  {  ht\_node\* item = (ht\_node\*)malloc(sizeof(ht\_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 = (ht\_node\*\*)calloc(table->size, sizeof(ht\_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->ht\_node->key);  free(temp->ht\_node->value);  free(temp->ht\_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(ht\_node\* 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++)  {  ht\_node\* item = table->items[i];  if (item != NULL)  {  free\_item(item);  }  }  free\_overflow(table);  free(table->items);  free(table);  }  List\* list\_insert(List\* list, ht\_node\* item)  {  if (list == NULL)  {  list = (List\*)malloc(sizeof(List));  list->ht\_node = item;  list->next = NULL;  return list;  }  else if (list->next == NULL)  {  List\* tmp = (List\*)malloc(sizeof(List));  tmp->ht\_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->ht\_node = item;  tmp->next = NULL;  list->next = tmp;  return list;  }  }  void handle\_collision(HT\* table, unsigned long index, ht\_node\* item)  {  List\* list = table->overflow[index];  if (list == NULL)  {  list = (List\*)malloc(sizeof(List));  list->ht\_node = item;  list->next = NULL;  table->overflow[index] = list;  return;  }  else {  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;  }  ht\_node\* item = create\_item(key, value);  int index = hash\_function(key);  ht\_node\* current\_item = table->items[index];  if (current\_item == NULL)  {  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);  return;  }  else  {  handle\_collision(table, index, item);  return;  }  }  }  char\* ht\_search(HT\* table, char\* key)  {  int index = hash\_function(key);  ht\_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->ht\_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])  {  ht\_node\* tmp = table->items[i];  printf("Index:%d, Key:%s, Value:%s\n", i, tmp->key, tmp->value);  }  }  printf("-------------------\n\n");  }  void save\_table(HT\* table, char\* path)  {  for (int i = 0; i < table->size; i++)  {  if (table->items[i])  {  ht\_node\* tmp = table->items[i];  char\* data = strdup(tmp->key);  data = strcat(data, ":");  data = strcat(data, tmp->value);  write\_to\_file(path, "HT", data);  }  }  }  void ht\_delete(HT\* table, char\* key)  {  int index = hash\_function(key);  ht\_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)  {  if (strcmp(item->key, key) == 0)  {  free\_item(item);  List\* list = head;  head = head->next;  list->next = NULL;  table->items[index] = create\_item(list->ht\_node->key, list->ht\_node->value);  free\_list(list);  table->overflow[index] = head;  return;  }  List\* curr = head;  List\* prev = NULL;  while (curr)  {  if (strcmp(curr->ht\_node->key, key) == 0)  {  if (prev == NULL)  {  free\_list(head);  table->overflow[index] = NULL;  return;  }  else  {  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 ht\_node {  char\* key;  char\* value;  } ht\_node;  typedef struct HT {  struct HT\*\* items;  struct List\*\* overflow;  int size;  int count;  }HT;  typedef struct List {  ht\_node\* ht\_node;  struct List\* next;  } List;  HT\* create\_table(int);  void ht\_insert(HT\*, char\* , char\*);  void print\_search(HT\*, char\*);  void print\_table(HT\*);  void ht\_delete(HT\*, char\*);  void free\_table(HT\*);  void save\_table(HT\* table, char\* path); |
| file.c | #include "queue.h"  #include "stack.h"  #include "set.h"  #include "hash\_table.h"  #include "file.h"  #include <stdio.h>  #include <string.h>  char\* decapitation(char\* input, char\* struc)  {  char\* output = strtok(input, ":");  if (strcmp(output, struc) == 0)  {  return strtok(NULL, ":");  }  else  {  return NULL;  }  }  void read\_file(char\* path, char\* struc)  {  FILE\* mf = fopen(path, "r");  if (!mf)  {  printf("ERROR: The file does not exist\n");  return;  }  char mstr[100] = "Start";  char\* estr = "Start";  while (estr != NULL)  { estr = fgets(mstr, sizeof(mstr), mf);  char\* tmp = decapitation(mstr, struc);  if (tmp != NULL)  {  if (!strcmp(struc, "Queue"))  {  push\_queue(tmp);  }  else if (!strcmp(struc, "Stack"))  {  push\_stack(tmp);  }  }  }  fclose(mf);  }  void read\_ht\_file(char\* path, HT\* table, char\* struc)  {  FILE\* mf = fopen(path, "r");  if (!mf)  {  printf("ERROR: The file does not exist\n");  return;  }  char mstr[100] = "Start";  char\* estr = "Start";  while (estr != NULL)  {  estr = fgets(mstr, sizeof(mstr), mf);  char\* tmp = decapitation(mstr, struc);  if (tmp != NULL)  {  if (!strcmp(struc, "HT"))  {  ht\_insert(table, tmp, tmp);  }  }  }  fclose(mf);  }  void read\_set\_file(char\* path, SET\* my\_set, char\* struc)  {  FILE\* mf = fopen(path, "r");  if (!mf)  {  printf("ERROR: The file does not exist\n");  return;  }  char mstr[100] = "Start";  char\* estr = "Start";  while (estr != NULL)  {  estr = fgets(mstr, sizeof(mstr), mf);  char\* tmp = decapitation(mstr, struc);  if (tmp != NULL)  {  if (!strcmp(struc, "Set"))  {  set\_insert(my\_set, tmp, "");  }  }  }  fclose(mf);  }  void write\_to\_file(char\* path, char\* struc, char\* data)  {  FILE\* mf = fopen(path, "a");  if (!mf)  {  printf("ERROR: Can't open the file\n");  return;  }  char\* result = strdup(struc);  char\* tmp = strdup(data);  fputs(result, mf);  fputs(":", mf);  fputs(tmp, mf);  fputs("\n", mf);  fclose(mf);  }  void clear\_file(char\* path)  {  FILE\* mf = fopen(path, "w");  if (!mf)  {  printf("ERROR: Can't open the file\n");  return;  }  fputs("", mf);  } |
| file.h | #pragma once  #include "set.h"  #include "hash\_table.h"  void read\_file(char\*, char\*);  void write\_to\_file(char\*, char\*, char\*);  void clear\_file(char\* path);  void read\_set\_file(char\* path, SET\* my\_set, char\* struc);  void read\_ht\_file(char\* path, HT\* table, char\* struc); |

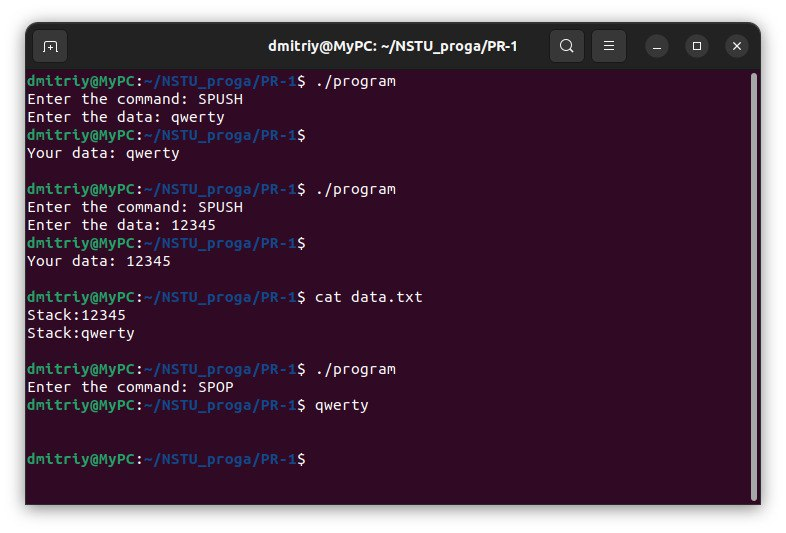


Рисунок 1 – Результат работы программы

Вывод: Практическая работа по созданию СУБД позволила на практике освоить работу со структурами данных, их сохранении в файл и извлечения из него. Были получены навыки работы с вызовом дочерних процессов и передачей аргументов программы.