Санкт-Петербургский политехнический университет Петра Великого

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

**Отчёт по лабораторной работе №5**

**Дисциплина**: Низкоуровневое программирование

**Тема**: Программирование на языке C

Выполнил студент гр. 3530901/90003 В.Д. Коротких

(подпись)

Преподаватель А. О. Алексюк

(подпись)

“ ” 2021 г.

Санкт-Петербург

2021

Оглавление

[1. Техническое задание 3](#_Toc71541752)

[2. Метод решения 3](#_Toc71541753)

[3. Решение 3](#_Toc71541754)

[4. Работа программы 11](#_Toc71541755)

[5. Тестирование программы 13](#_Toc71541756)

[6. GitHub 16](#_Toc71541757)

# Техническое задание

Реализовать на языке C префиксное дерево.

# Метод решения

Trie (оно же префиксное дерево) – структура данных, позволяющая хранить ассоциативный массив, ключами которого являются строки. Представляет собой корневое дерево, каждое ребро которого помечено каким-то символом так, что для любого узла все рёбра, соединяющие этот узел с его сыновьями, помечены разными символами.

Реализация осуществлена с помощью узлов дерева, каждый из которых хранит массив следующих узлов, а также информацию о том, является ли этот узел концом слова. Само дерево содержит указатель на корень (корневой узел) и размер дерева (количество слов). Функции дерева (добавление, удаление, поиск, печать) основаны на поэтапном спуске от узла к узлу.

# Решение

main.c

|  |
| --- |
| #include <stdio.h> #include <stdlib.h> #include <fcntl.h> #include "trie.h" #include "cmd\_parser.h"  int main() {  \_setmode(\_fileno(stdout), \_O\_U16TEXT);  Trie \*trie = create\_Trie();  char \*line;  line = malloc(MAX\_WORD\_LENGTH \* sizeof(char));  parse\_command("-h\n", NULL);  while (1) {  line = fgets(line, MAX\_WORD\_LENGTH, stdin);  if (!parse\_command(line, trie)) return 0;  } } |

trie.c

|  |
| --- |
| #include "trie.h" #include <stdio.h> #include <stdlib.h> #include <ctype.h>  struct TrieNode {  char data; // символ  TrieNode \*children[N\_CHARACTERS]; // дети  int is\_leaf; // этот узел - конец слова };  struct Trie {  TrieNode \*root;  int size; };  Trie \*create\_Trie() {  // создание дерева  Trie \*trie = (Trie \*) malloc(sizeof(Trie));  trie->root = make\_trieNode(' ');  trie->size = 0;  return trie; }  Trie \*clear\_trie(Trie \*trie) {  trie->size = 0;  free\_trieNode(trie->root);  trie->root = NULL;  return trie; }  TrieNode \*make\_trieNode(char data) {  // Создание узла  TrieNode \*node = (TrieNode \*) malloc(sizeof(TrieNode));  for (int i = 0; i < N\_CHARACTERS; i++)  node->children[i] = NULL;  node->is\_leaf = 0;  node->data = data;  return node; }   void free\_trieNode(TrieNode \*node) {  //Освобождение памяти, занимаемой узлом  if (node == NULL) {  free(node);  return;  }  for (int i = 0; i < N\_CHARACTERS; i++) {  if (node->children[i] != NULL) {  // очищае рукурсивно  free\_trieNode(node->children[i]);  } else {  continue;  }  }  //node = NULL;  free(node); }  int get\_char\_position(char ch) {  int ch\_ = (int) tolower(ch);  if (isdigit(ch)) return ch\_ - 48;  //Русские буквы не работают  //if (ch\_ < 0) return ch\_ + 58;  return (int) ch\_ - 'a' + 10; }  int trie\_add\_word(Trie \*trie, char \*word) {  // вставка слова в дерево  if (trie->root == NULL) trie->root = make\_trieNode(' ');  TrieNode \*temp = trie->root;   for (int i = 0; word[i] != '\0'; i++) {  int idx = get\_char\_position(word[i]); // получаем позицию для символа  if (temp->children[idx] == NULL) {  // если при попытке спуститься вниз по дереву для вставляемого слова  // не оказывается узла, то создаём этот узел  temp->children[idx] = make\_trieNode(word[i]);  }  // если же узел есть, то просто спускаемся дальше вниз  temp = temp->children[idx];  }  if (temp->is\_leaf)  return 0;  else {  trie->size++;  temp->is\_leaf = 1;// обозначаем, что это слово  }  return 1; }  int trie\_search(Trie \*trie, char \*word) {  // Поиск слова в дереве: 1 - найдено, 0 - не найдено  if (trie->root == NULL) return 0;  TrieNode \*temp = trie->root;   for (int i = 0; word[i] != '\0'; i++) {  // спускаемся вниз по дереву  int position = get\_char\_position(word[i]);  if (temp->children[position] == NULL) {  // если при спуске не оказывается следующего узла, то значит слова нет  return 0;  }  temp = temp->children[position];  }  if (temp != NULL && temp->is\_leaf == 1) {  // проверяем удался ли спуск и является ли узел концом слова  return 1;  }  return 0; }  int trie\_delete\_word(Trie \*trie, char \*word) {  if (trie->size == 0) {  return 2;  }   TrieNode \*temp = trie->root;  TrieNode \*previous;  TrieNode \*previousDel = NULL;   TrieNode \*deleteFrom = NULL; // узел, который можно удалить, не повредив дерево   for (int i = 0; word[i] != '\0'; i++) {  //спускаемся вниз по дереву  int position = get\_char\_position(word[i]);  if (temp->children[position] == NULL) {  // такого слова нет  return 0;  } else {  // проверяем, можно ли удалить этот узел, не повредив дерево  previous = temp;  int j = 0;  for (; j < N\_CHARACTERS; j++) {  if (temp->children[j] != NULL) {  if (temp->children[j]->data != word[i]) break;  }  }  if (j == N\_CHARACTERS) {  if (deleteFrom == NULL && !temp->is\_leaf) {  previousDel = previous;  deleteFrom = temp;  } else if (temp->is\_leaf) {  deleteFrom = NULL;  }  } else deleteFrom = NULL;  // спускаемся  temp = temp->children[position];  }  }  int j = 0;  for (; j < N\_CHARACTERS; j++) {  if (temp->children[j] != NULL) {  break;  }  }  if (j == N\_CHARACTERS) {  if (deleteFrom == NULL) {  previousDel = previous;  deleteFrom = temp;  }  } else deleteFrom = NULL;  if (temp == NULL) {  return 0;  }  if (!temp->is\_leaf) {  return 0;  }   temp->is\_leaf = 0; // удаляем слово  if (deleteFrom != NULL) {  if (deleteFrom == trie->root) deleteFrom = deleteFrom->children[get\_char\_position(word[0])];  int i = 0;  for (; previousDel->children[i] != deleteFrom; i++);  previousDel->children[i] = NULL;  free\_trieNode(deleteFrom);// очищаем память, если возможно  }  trie->size--;  return 1; }  wchar\_t \*concat(wchar\_t \*s1, wchar\_t \*s2) {  const size\_t len1 = wcslen(s1);  const size\_t len2 = wcslen(s2);  wchar\_t \*result = malloc((len1 + len2 + 1) \* sizeof(result)); // +1 для '\0'  wcscpy(result, s1);  wcscat(result, s2);  return result; }  void trie\_print\_(wchar\_t \*prefix, int isTail, TrieNode \*node) {  wchar\_t \*suffix;  wchar\_t \*fix;  if (isTail) {  suffix = L" ";  fix = L"└── ";  } else {  suffix = L"│ ";  fix = L"├── ";  }   wchar\_t \*data = L"";  if (node->is\_leaf) {  data = L"<-";  }   wprintf(L"%s%s%c%s\n", prefix, fix, node->data, data);   int size = 0;  for (int i = 0; i < N\_CHARACTERS; i++) {  if (node->children[i] != NULL) size++;  }   for (int i = 0; i < N\_CHARACTERS && size > 1; i++) {  if (node->children[i] != NULL) {  size--;  wchar\_t \*new\_prefix = concat(prefix, suffix);  trie\_print\_(new\_prefix, 0, node->children[i]);  free(new\_prefix);  }  }  if (size != 0) {  int last = 0;  for (int i = 0; i < N\_CHARACTERS; i++) {  if (node->children[i] != NULL) {  last = i;  }  }  wchar\_t \*new\_prefix = concat(prefix, suffix);  trie\_print\_(new\_prefix, 1, node->children[last]);  free(new\_prefix);  } }  void trie\_print(Trie \*trie) {  if (trie->size == 0) {  wprintf(L"Trie is empty!\n\n");  return;  }  wchar\_t \*prefix = L"";  trie\_print\_(prefix, 1, trie->root); } |

trie.h

|  |
| --- |
| #ifndef LOW\_LEVEL\_PROGRAMMING\_TRIE\_H #define LOW\_LEVEL\_PROGRAMMING\_TRIE\_H  #include <string.h>  // 26 English and 10 digits #define N\_CHARACTERS 36 #define MAX\_WORD\_LENGTH 1024 typedef struct Trie Trie;  typedef struct TrieNode TrieNode;  Trie \*create\_Trie();  Trie \*clear\_trie(Trie \*trie);  TrieNode \*make\_trieNode(char data);  void free\_trieNode(TrieNode \*node);  int get\_char\_position(char ch);  int trie\_add\_word(Trie \*trie, char \*word);  int trie\_search(Trie \*trie, char \*word);  int trie\_delete\_word(Trie \*trie, char \*word);  wchar\_t \*concat(wchar\_t \*s1, wchar\_t \*s2);  void trie\_print\_(wchar\_t \*prefix, int isTail, TrieNode \*node);  void trie\_print(struct Trie \*trie);  #endif |

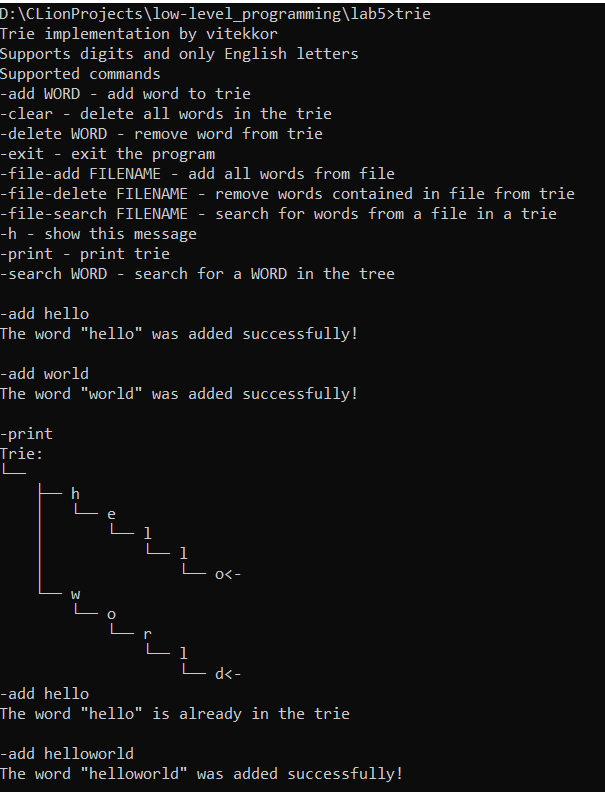
cmd\_parser.c

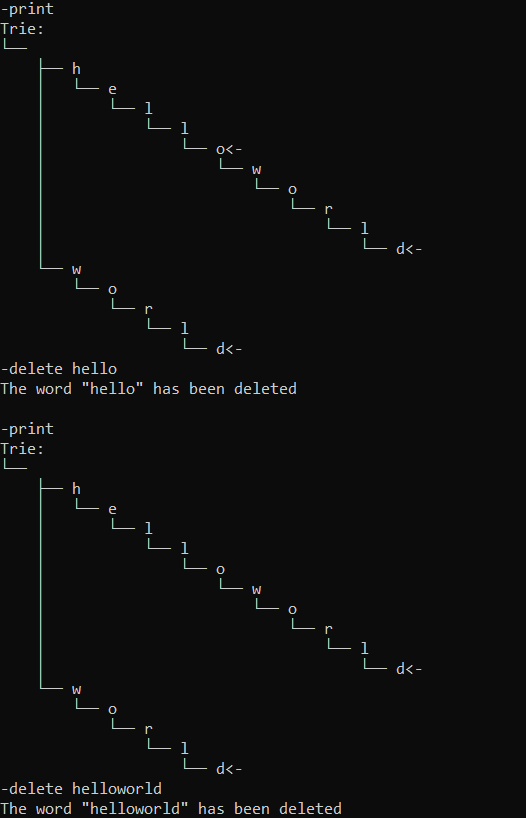
|  |
| --- |
| #include "cmd\_parser.h" #include "stdio.h" #include <stdlib.h> #include <string.h>  char \*readArgument(int startIndex, const char \*line) {  //чтение аргумента командной строки  if (startIndex > (int) strlen(line)) return NULL;  char \*argument;  argument = calloc((MAX\_WORD\_LENGTH - startIndex), sizeof(char));  int j = 0;  for (int i = startIndex; line[i] != '\n' && line[i] != ' '; i++) {  argument[j] = line[i];  j++;  }  return argument; }  void from\_file(int (\*action)(Trie \*, char \*), void (\*action\_handler)(int, char \*), Trie \*trie, char \*line, int i) {  //выполнение дейстивй с файлом  char \*fileName = readArgument(i, line);  if (fileName[0] == '\0') {  wprintf(L"Incomplete command. Enter a name for the file.\n\n");  return;  }  FILE \*file = fopen(fileName, "r");  if (file == NULL) {  wprintf(L"Error! Unable to open file!\n\n");  return;  }  char word[MAX\_WORD\_LENGTH];  // читаем файл  while (fgets(word, MAX\_WORD\_LENGTH, file) != NULL) {  word[strlen(word) - 1] = '\0';  //выполняем действие. action handler - метод, обрабатывающий результат action  action\_handler(action(trie, word), word);  } }  void add\_handler(int add, char \*word) {  // обработка результата trie\_add\_word  if (add)  wprintf(L"The word \"%S\" was added successfully!\n\n", (wchar\_t \*) word);  else  wprintf(L"The word \"%S\" is already in the trie\n\n", word); }  void delete\_handler(int delete, char \*word) {  // обработка результата trie\_delete\_word  switch (delete) {  case 0:  wprintf(L"The trie does not contain a word \"%S\"\n\n", word);  break;  case 1:  wprintf(L"The word \"%S\" has been deleted\n\n", word);  break;  default:  wprintf(L"Trie is empty!\n\n");  } }  void search\_handler(int search, char \*word) {  // обработка результата trie\_search  if (search)  wprintf(L"The word \"%S\" found\n\n", word);  else  wprintf(L"The word \"%S\" not found\n\n", word); }  int parse\_command(char \*line, Trie \*trie) {  // парсинг командной строки  char \*command = calloc(12, sizeof(char));  int i = 0;  if (line[i] != '-') {  wprintf(L"Invalid command\nEnter -h to see the available commands\n\n");  return 1;  }  int j = 0;  // считываем команду  for (; line[i] != '\n' && line[i] != ' '; i++) {  command[j] = line[i];  j++;  }  // определяем, что за команда и выполняем соответсвующие действия  if (strcmp(command, "-add") == 0) {  i++;  char \*word = readArgument(i, line);  if (word[0] == '\0') {  wprintf(L"Incomplete command. You must enter the WORD to be inserted\n\n");  free(command);  return 1;  }  add\_handler(trie\_add\_word(trie, word), word);  free(command);  return 1;  }  if (strcmp(command, "-clear") == 0) {  wprintf(L"Removing all words from the trie ...\n");  clear\_trie(trie);  wprintf(L"The trie has been cleared!\n\n");  free(command);  return 1;  }  if (strcmp(command, "-delete") == 0) {  i++;  char \*word = readArgument(i, line);  if (word[0] == '\0') {  wprintf(L"Incomplete command. You must enter the WORD to be deleted\n\n");  free(command);  return 1;  }  delete\_handler(trie\_delete\_word(trie, word), word);  free(command);  return 1;  }  if (strcmp(command, "-exit") == 0) {  wprintf(L"Removing all words from the trie ...\n\n");  clear\_trie(trie);  wprintf(L"The trie has been cleared!\nExit.\n");  free(command);  free(trie);  return 0;  }  if (strcmp(command, "-file-add") == 0) {  i++;  from\_file(&trie\_add\_word, &add\_handler, trie, line, i);  free(command);  return 1;  }  if (strcmp(command, "-file-delete") == 0) {  i++;  from\_file(&trie\_delete\_word, &delete\_handler, trie, line, i);  free(command);  return 1;  }  if (strcmp(command, "-file-search") == 0) {  i++;  from\_file(&trie\_search, &search\_handler, trie, line, i);  free(command);  return 1;  }  if (strcmp(command, "-h") == 0) {  wprintf(L"Trie implementation by vitekkor\nSupports digits and only English letters\nSupported commands\n");  wprintf(L"-add WORD - add word to trie\n");  wprintf(L"-clear - delete all words in the trie\n");  wprintf(L"-delete WORD - remove word from trie\n");  wprintf(L"-exit - exit the program\n");  wprintf(L"-file-add FILENAME - add all words from file\n");  wprintf(L"-file-delete FILENAME - remove words contained in file from trie\n");  wprintf(L"-file-search FILENAME - search for words from a file in a trie\n");  wprintf(L"-h - show this message\n");  wprintf(L"-print - print trie\n");  wprintf(L"-search WORD - search for a WORD in the tree\n\n");  free(command);  return 1;  }  if (strcmp(command, "-print") == 0) {  wprintf(L"Trie:\n");  trie\_print(trie);  free(command);  return 1;  }  if (strcmp(command, "-search") == 0) {  i++;  char \*word = readArgument(i, line);  if (word[0] == '\0') {  wprintf(L"Incomplete command. Enter your search WORD\n\n");  free(command);  return 1;  }  wprintf(L"Searching a word \"%S\": ", (wchar\_t \*) word);  search\_handler(trie\_search(trie, word), word);  free(command);  return 1;  }  wprintf(L"Unknown command\n\n");  return 1; } |

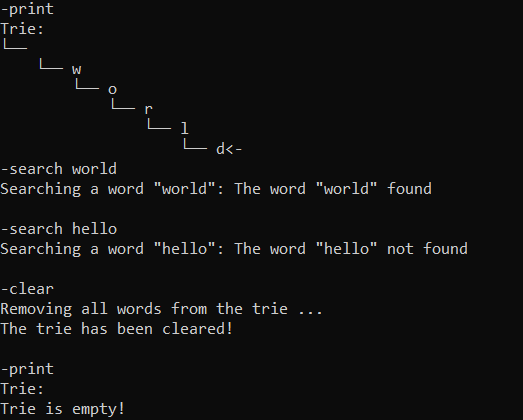
cmd\_parser.h

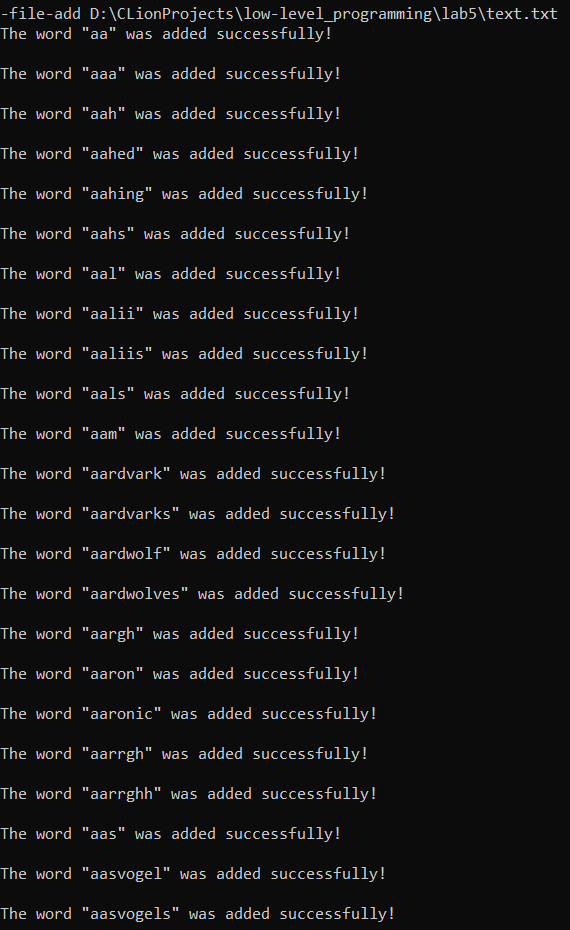
|  |
| --- |
| #ifndef LOW\_LEVEL\_PROGRAMMING\_CMD\_PARSER\_H #define LOW\_LEVEL\_PROGRAMMING\_CMD\_PARSER\_H  #include "trie.h"  char \*readArgument(int startIndex, const char \*line);  void from\_file(int (\*action)(Trie \*, char \*), void (\*action\_handler)(int, char \*), Trie \*trie, char \*line, int i);  void add\_handler(int add, char \*word);  void delete\_handler(int delete, char \*word);  void search\_handler(int search, char \*word);  int parse\_command(char \*line, Trie \*trie);  #endif |

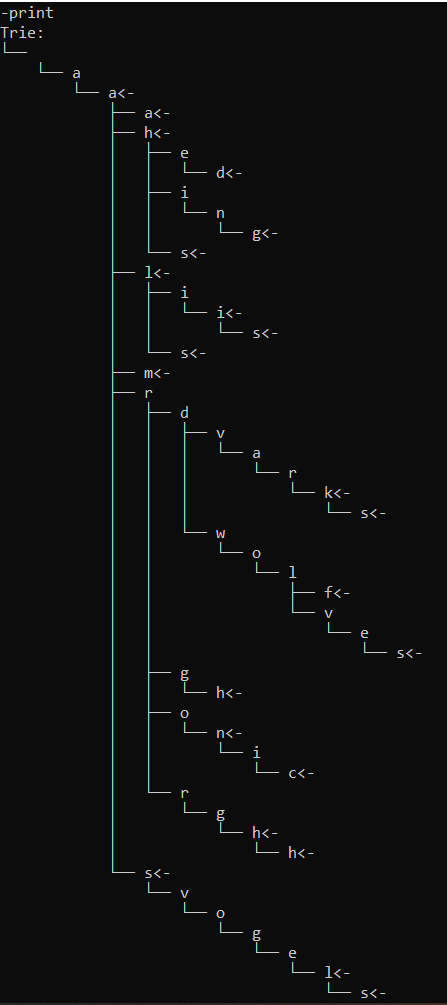
# Работа программы

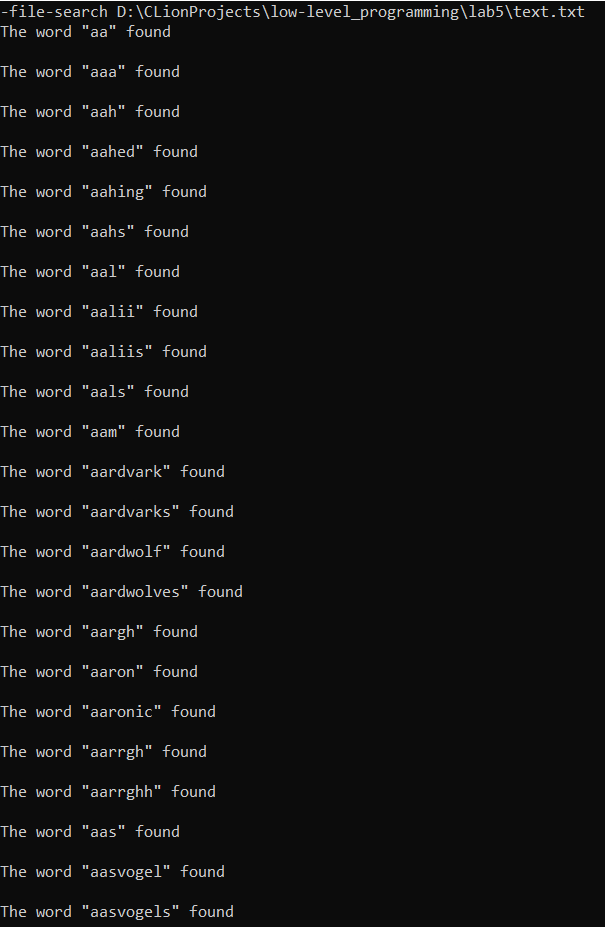


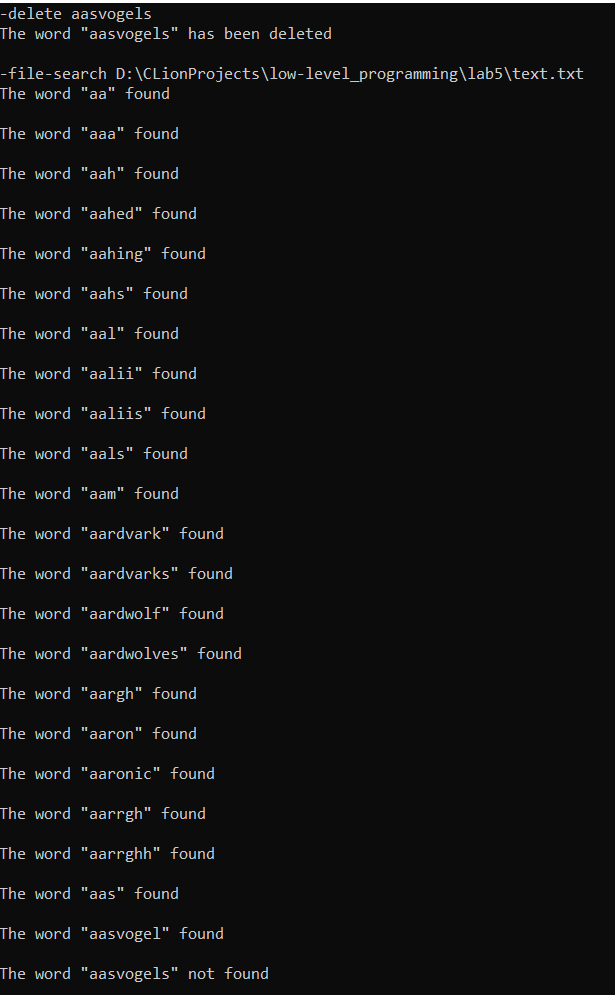




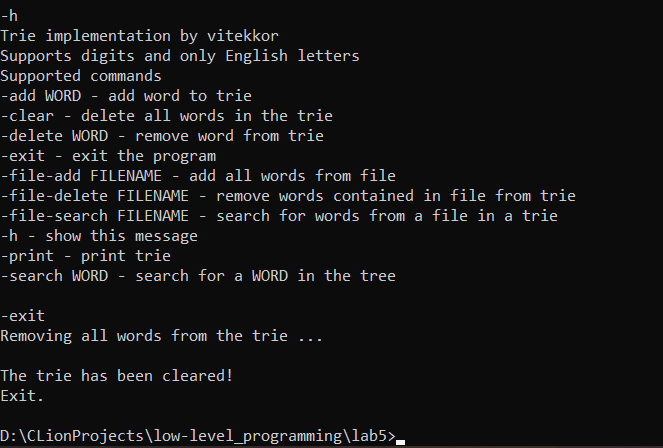












# Тестирование программы

Для проверки правильности работы программы были написаны автоматические тесты.

test.c

|  |
| --- |
| #include <stdio.h> #include <string.h> #include <stdlib.h> #include "test.h" #include "../trie.h"  int run\_all\_tests() {  printf("Running tests...\n");  return test\_add() + test\_delete() + test\_search(); }  int test\_add() {  Trie \*trie = create\_Trie();  char path\_in[] = "..\\test\_data\\test\_add.txt";  char path\_out[] = "..\\test\_data\\test\_add\_expected.txt";  FILE \*file\_in = fopen(path\_in, "r");  FILE \*file\_out = fopen(path\_out, "r");  char word[MAX\_WORD\_LENGTH];  while (fgets(word, MAX\_WORD\_LENGTH, file\_in) != NULL) {  word[strlen(word) - 1] = '\0';  int result = trie\_add\_word(trie, word);  fgets(word, MAX\_WORD\_LENGTH, file\_out);  if (result != atoi(word)) {  printf("Add test failed\n");  return 1;  }  }  clear\_trie(trie);  free(trie);  fclose(file\_in);  free(file\_in);  fclose(file\_out);  free(file\_out);  printf("Add test passed\n");  return 0; }  int test\_delete() {  Trie \*trie = create\_Trie();  char path\_in[] = "..\\test\_data\\test\_add.txt";  FILE \*file\_in = fopen(path\_in, "r");  char word[MAX\_WORD\_LENGTH];  while (fgets(word, MAX\_WORD\_LENGTH, file\_in) != NULL) {  word[strlen(word) - 1] = '\0';  trie\_add\_word(trie, word);  }  fclose(file\_in);  char path\_delete\_in[] = "..\\test\_data\\test\_delete.txt";  char path\_delete\_out[] = "..\\test\_data\\test\_delete\_expected.txt";  FILE \*file\_delete\_in = fopen(path\_delete\_in, "r");  FILE \*file\_delete\_out = fopen(path\_delete\_out, "r");  while (fgets(word, MAX\_WORD\_LENGTH, file\_delete\_in) != NULL) {  word[strlen(word) - 1] = '\0';  int result = trie\_delete\_word(trie, word);  fgets(word, MAX\_WORD\_LENGTH, file\_delete\_out);  if (result != atoi(word)) {  printf("Delete test failed\n");  return 1;  }  }  clear\_trie(trie);  free(trie);  fclose(file\_delete\_in);  fclose(file\_delete\_out);  free(file\_delete\_out);  free(file\_delete\_in);  free(file\_in);  printf("Delete test passed\n");  return 0; }  int test\_search() {  Trie \*trie = create\_Trie();  char path\_in[] = "..\\test\_data\\test\_add.txt";  FILE \*file\_in = fopen(path\_in, "r");  char word[MAX\_WORD\_LENGTH];  while (fgets(word, MAX\_WORD\_LENGTH, file\_in) != NULL) {  word[strlen(word) - 1] = '\0';  trie\_add\_word(trie, word);  }  fclose(file\_in);  char path\_delete\_in[] = "..\\test\_data\\test\_search.txt";  char path\_delete\_out[] = "..\\test\_data\\test\_search\_expected.txt";  FILE \*file\_delete\_in = fopen(path\_delete\_in, "r");  FILE \*file\_delete\_out = fopen(path\_delete\_out, "r");  int result;  while (fgets(word, MAX\_WORD\_LENGTH, file\_delete\_in) != NULL) {  word[strlen(word) - 1] = '\0';  if (strcmp(word, "delete") == 0) {  result = trie\_search(trie, word);  fgets(word, MAX\_WORD\_LENGTH, file\_delete\_out);  if (result != atoi(word)) {  printf("Search test failed\n");  return 1;  }  trie\_delete\_word(trie, word);  }  result = trie\_search(trie, word);  fgets(word, MAX\_WORD\_LENGTH, file\_delete\_out);  if (result != atoi(word)) {  printf("Search test failed\n");  return 1;  }  }  clear\_trie(trie);  free(trie);  fclose(file\_delete\_in);  fclose(file\_delete\_out);  free(file\_delete\_out);  free(file\_delete\_in);  free(file\_in);  printf("Search test passed\n");  return 0; } |

test.h

|  |
| --- |
| #ifndef LOW\_LEVEL\_PROGRAMMING\_TEST\_H #define LOW\_LEVEL\_PROGRAMMING\_TEST\_H  int run\_all\_tests();  int test\_add();  int test\_delete();  int test\_search();  #endif |

Входные данные находятся в файлах test\_method.txt. В файлах test\_method\_expected.txt находятся ожидаемые значения для каждого вызова метода. Суть заключается в сравнении ожидаемого значения вызова метода с получившимся значением вызова метода.

Метод run\_all\_tests запускает все тесты.

# GitHub

<https://github.com/vitekkor/low-level_programming/tree/master/lab5>