МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра Автоматизированных систем управления



**Расчетно-графическая работа**

**По «Алгоритмам и структурам данных»**

**на тему: «***Алгоритм Ахо-Корасик - поиск***»**

Выполнил: Проверил: Студент гр. *АВТ - 813*, *АВТФ Доцент*

*Чернаков К.О. Ландовский В.В.*

Новосибирск,

2020

Оглавление

[**Задание:** 3](#_Toc26836431)

[**Теоретическая часть:** 3](#_Toc26836432)

[**Вывод теоретических оценок затрат времени и памяти** 7](#_Toc26836433)

[**Оценка памяти для поиска подстроки:** 7](#_Toc26836434)

[**Пример входных и выходных данных:** 8](#_Toc26836435)

[**Экспериментальная часть:** 9](#_Toc26836436)

[**Оценка затрат времени при поиске подстроки:** 9](#_Toc26836437)

[**Вывод:** 11](#_Toc26836438)

[**Листинг программы:** 12](#_Toc26836439)

[**Литература:** 17](#_Toc26836440)

# **Задание:**

Провести анализ алгоритма поиска подстроки с помощью суффиксного дерева. Разработать программную реализацию алгоритма для оценки затрачиваемого объема времени и памяти. Построить зависимости этих величин от объема входных данных. Рассмотреть различные («хорошие» и «плохие») варианты входного потока, т.е. попытаться дать оценку в лучшем и в худшем случае. Сравнить с теоретическими оценками.

# **Теоретическая часть:**

**Суффиксное дерево** – это нагруженное дерево (бор), содержащее все суффиксы некоторой строки. Можно сказать, что это специальный способ представления строки, позволяющий быстро решать множество задач.

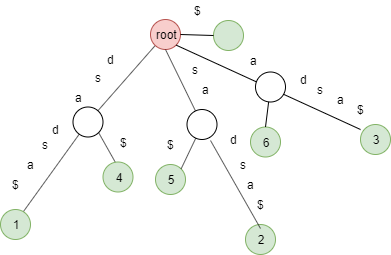
Автором идеи является Питер Вейнер, применение суффиксных деревьев для поиска впервые опубликовано им в 1973 г. в работе. Суффиксное дерево занимает большой объем памяти.

Следует отметить одно важное свойство – каждый суффикс должен заканчиваться в своем листе, для его выполнения последний символ строки не должен совпадать с каким-либо другим символом. Для гарантированного выполнения указанного свойства к строке, не глядя на ее содержимое, добавляют специальный уникальный символ, которого нет в исходном алфавите.

Эдвард МакКрейт в 1976 году предложил использовать сжатое суффиксное дерево, в котором присутствуют только те вершины, в которых есть ветвления.

Наивный алгоритм построения суффиксного дерева заключается в последовательном добавлении суффиксов, начиная с самого длинного. Добавление каждого суффикса предполагает попытку прочитать его от корня и в момент, когда чтение невозможно (пути по очередному символу нет) создание ответвления и нового листа.

Рассмотрим построение дерева на примере строки dsadsa$.



*Рис. 1. Построение дерева для строки dsadsa$*

Алгоритм, представленный в виде псевдокода:

Создать корень дерева;

n=|t|;

**Для** i от 0 до (n-1) **Цикл (3)**

w= «Корень дерева»;

j=i;

**Если** Есть путь из w по символу t[j] **Тогда** (6)

Для k от «метка начала ребра» до «метка окончания ребра» **Цикл**

**Если** t[j]= t[k] **Тогда**

j=j+1;

**Иначе**

Создать новую внутреннюю вершину v;

Добавить лист u с номером i;

Добавить ребро из v в u помеченное t[j..n];

Перейти к следующей итерации цикла (3);

**Конец если**

**Конец цикла**

w = «очередная вершина»;

Перейти к шагу (6);

**Конец если**

Добавить лист u с номером i;

Добавить ребро из w в u помеченное t[j..n];

**Конец цикла**

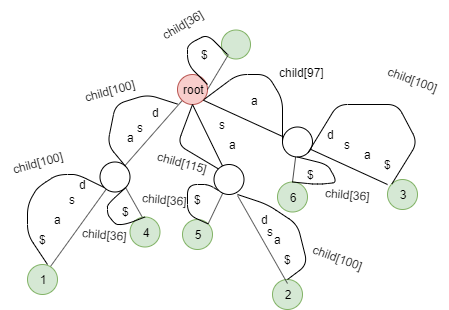
К данному алгоритму нужно добавить поиск подстроки: в общем случае данная задача формулируется как поиск всех позиций в некоторой строке t, начиная с которых можно прочитать строку p (поиск всех вхождений p в t).

Суффиксное дерево при условии, что оно уже построено для строки t, поможет ответить на вопрос о том, встречается ли образец p в тексте t за время линейное относительно |p|. Для получения ответа достаточно попытаться прочитать искомый образец в дереве.

Описание алгоритма поиска подстроки:

На вход подается некоторая строка, берем первый символ этой строки и находим у нее ASCII-код для первого элемента строки, который является индексом для ребенка данного корня, затем находим самого ребенка по ASCII-код, после этого пытаемся сравнить элементы данной подстроки с ребром ребенка.

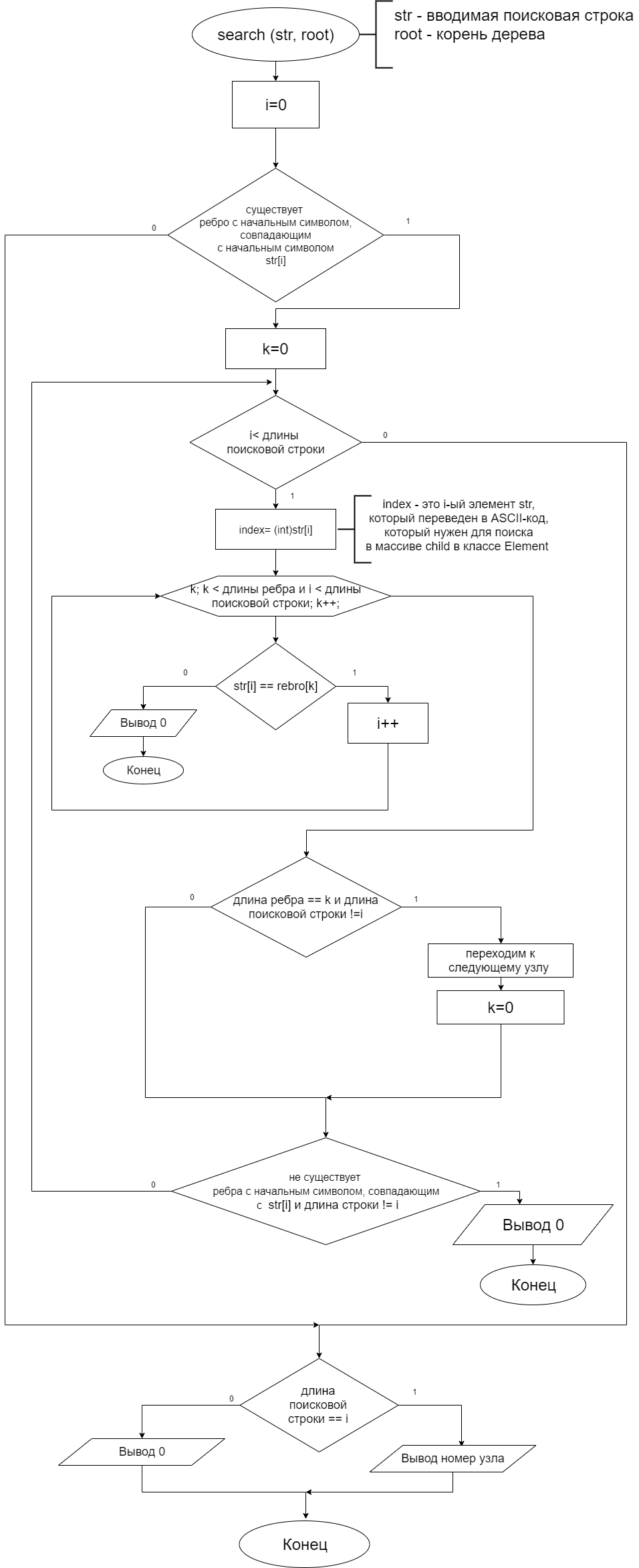
1. Если данная подстрока полностью входит в ребро ребенка и данный ребенок не имеет своих детей, то мы нашли подстроку, иначе подстрока не найдена;
2. Если данная подстрока частично совпала с ребром ребенка, у этого ребенка находим еще ребенка, который может дополнить совпадение с нашей данной строкой, иначе подстрока не найдена;



*Рис. 2. Поиск подстроки dsadsa$ в суффиксном дереве*

**Блок-схема для алгоритма поиска подстроки:**

**Пояснение:** index - это переменная int равная ASCII-коду str[i], это переменная нужна нам для быстрого нахождения child в узле родителя, вместо того, чтобы перебирать все узлы, тем самым эта переменная сохраняет линейность для поиска по дереву



# **Вывод теоретических оценок затрат времени и памяти**

**Оценка времени при построении дерева:**

Основной цикл делает шагов, на каждом из которых он производит сравнений символов. Значит итоговая оценка времени работы данного алгоритма равна

**Оценка времени при поиске подстроки:**

Основной цикл делает переходов. Значит итоговая оценка времени работы данного алгоритма равна .

**Оценка памяти для построения дерева:**

Можно рассмотреть два крайних случая:

1. Строка состоит из неповторяющихся символов:

Тогда количество узлов будет равно количеству листьев в дереве плюс корень.

Так как каждый суффикс заканчивается в своем листе их количество будет равно , но так как мы расширяем исходную строку уникальным символом добавится еще один лист. Следовательно получаем лист.

Общее число узлов в этом случае равно

1. Строка состоит из повторяющихся символов:

Тогда количество узлов будет рано количеству листьев в дереве плюс количество ветвлений.

Так как каждый суффикс заканчивается в своем листе их количество будет равно , но так как мы расширяем исходную строку уникальным символом добавится еще один лист. Следовательно получаем лист.

Ветвления появляться в процессе ответвления каждого суффикса кроме самого длинного (идущего из корня в первый лист) и самого короткого (ребро в последний лист помечено одним символом). Таким образом количество вершин ветвления включая корень – , но так как мы расширяем строку одним символом, получаем , так же корень тоже является ветвлением. Получаем общее число ветвлений равное .

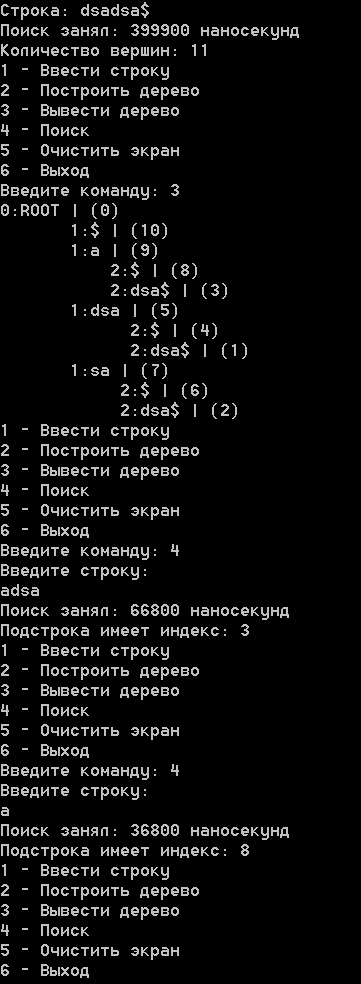
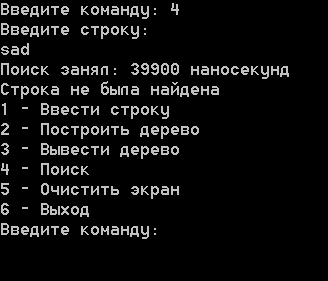
Тогда общее число узлов равно

## **Оценка памяти для поиска подстроки:**

Так как происходит только поиск подстроки, это значит, что затраты памяти не происходит, потому что при поиске нет создания новых элементов.

## **Пример входных и выходных данных:**

На вход программа принимает строку, по которой строит суффиксное дерево, после чего на консоль можно вывести это дерево, затем пользователь вводит подстроку, и программа выводит время поиска подстроки и индекс, который указывает, в какой строке находится искомая подстрока.

*Рис. 4. Пример входных и выходных данных*

# **Экспериментальная часть:**

Проведем серию экспериментов и на их основе составим таблицу результатов, позволяющую оценить затраты памяти и времени. По полученным данным построим графики, чтобы сравнить полученные затраты времени и памяти с их теоретической оценкой.

## **Оценка затрат времени при поиске подстроки:**

Строка состоит из *повторяющихся* символов:

В качестве входных данных будут выступать две строки: строка, по которой строится дерево, и подстрока, которую мы ищем в вводимой изначально строке, длина подстроки будет n=1, 20, 40, 60, 80, 100, длина вводимой строки для построения дерева будет n=100.

|  |  |  |  |
| --- | --- | --- | --- |
| ***№*** | ***Длина строки*** | ***Длина подстроки*** | ***Затраченное время (наносекунды)*** |
| 1 | 100 | 1 | 44100 |
| 2 | 100 | 20 | 111200 |
| 3 | 100 | 40 | 176100 |
| 4 | 100 | 60 | 242400 |
| 5 | 100 | 80 | 305600 |
| 6 | 100 | 100 | 365100 |

*Таблица 1. Зависимость временных затрат от длинны строки, в “лучшем” случае.*

*График 1. Зависимость временных затрат от длинны строки, в “лучшем” случае.*

Строка состоит из *неповторяющихся* символов:

В качестве входных данных будут выступать две строки: строка, по которой строится дерево, и подстрока, которую мы ищем в вводимой изначально строке, длина подстроки будет n=1, 20, 40, 60, 80, 100, длина вводимой строки для построения дерева будет n=100.

|  |  |  |  |
| --- | --- | --- | --- |
| ***№*** | ***Длина строки*** | ***Длина подстроки*** | ***Затраченное время (наносекунды)*** |
| 1 | 100 | 1 | 42100 |
| 2 | 100 | 20 | 80600 |
| 3 | 100 | 40 | 152600 |
| 4 | 100 | 60 | 209200 |
| 5 | 100 | 80 | 240000 |
| 6 | 100 | 100 | 298100 |

*Таблица 2. Зависимость временных затрат от длинны строки, в “лучшем” случае.*

*График 2. Зависимость временных затрат от длинны строки, в “лучшем” случае.*

# **Вывод:**

Из графиков можно сделать вывод, что зависимость экспериментальной оценки в большей степени совпадает с установленной теоретической оценкой.

На графиках можно заметить, что при использовании сжатого дерева для поиска подстроки отклонение между теоретическим и экспериментальным значениями различно для поиска по дереву с неповторяющимся и повторяющимся символами. Это происходит, потому что поиск по дереву с повторяющимися элементами похож на поиск по несжатому дереву, так как каждый узел имеет ребро в один символ, тем самым увеличивает время поиска подстроки, но не влияет на линейность. В обоих случаях выполняется поиск подстроки за , что и требовалось реализовать.

Оценивая временные затраты программы, можно сказать, что при увеличении длины строки в целом увеличивается время выполнения алгоритма.

# **Листинг программы:**

// ConsoleApplication15.cpp: Поиск подстроки с помощью суффиксного дерева

#include "pch.h"

#include <iostream>

#include <fstream>

#include <list>

#include <iostream>

#include <Windows.h>

#include <ctime>

#include <string>

#include <chrono>

#include <queue>

using namespace std;

#define N 256

#define LENGHT 100

class Element;

int count0;

class Element

{

public:

Element \*child[N];

Element \*parent;

string rebro;

int number;

Element(int number)

{

this->number = number;

parent = NULL;

for (int i = 0; i < N; i++)

{

child[i] = NULL;

}

}

};

bool getElement(char sign, Element \*vertex)

{

if (vertex->child[(int)sign] != NULL)

{

return true;

}

return false;

}

void SufixTree(string str, Element \*root)

{

Element \*PTTR;

bool ToNewVertex = false;

bool AddNewVertex = false;

int length = str.length();

int i = 0;

while (i < length)

{

string bufString = str;

PTTR = root;

int j = i;

int PosChild;

AddNewVertex = true;

while (getElement(bufString[j], PTTR))

{

AddNewVertex = false;

PosChild = bufString.at(j);// переменная находит индекс дочернего элемента

int k = 0;

for (k; k < PTTR->child[PosChild]->rebro.length(); k++)

{

if (bufString[j] == PTTR->child[PosChild]->rebro[k])

{

j++;

ToNewVertex = true;

}

else

{

count0++;

Element \*list = new Element(count0);

list->rebro = PTTR->child[PosChild]->rebro.substr(0, k);

PTTR->child[PosChild]->rebro.erase(0, k);

PTTR->child[PosChild]->parent = list;

list->child[(int)PTTR->child[PosChild]->rebro[0]] = PTTR->child[PosChild];

list->parent = PTTR;

PTTR->child[PosChild] = list;

PTTR = list;

count0++;

Element \*vertex = new Element(count0);

vertex->rebro = bufString.erase(0, j);

vertex->parent = PTTR;

PTTR->child[(int)vertex->rebro[0]] = vertex;

ToNewVertex = false;

int listNumber = list->number;

list->number = vertex->number;

vertex->number = listNumber;

break;

}

}

if (ToNewVertex)

{

PTTR = PTTR->child[PosChild];

PosChild = bufString.at(j);

count0++;

Element\* list = new Element(count0);

list->rebro = bufString.erase(0, j);

list->parent = PTTR;

PTTR->child[PosChild] = list;

}

i++;

break;

}

if (AddNewVertex)

{

PosChild = bufString.at(j);

count0++;

Element \*list = new Element(count0);

list->rebro = bufString.erase(0, j);

list->parent = PTTR;

PTTR->child[PosChild] = list;

i++;

}

}

}

void ShowThree(Element root, int num, int kol)

{

if (root.number != 0)

{

kol += root.parent->rebro.size() + 3;

}

for (int i = 0; i < kol; i++)

{

cout << " ";

}

cout << num << ":";

cout << root.rebro << " | (" << root.number << ")" << endl;

num++;

for (int i = 0; i < 256; i++)

{

if (root.child[i] != NULL) {

ShowThree(\*root.child[i], num, kol);

}

}

}

int search(string str, Element\* root)

{

int length = str.length();

int i = 0;

int index;

Element\* PTTR = root;

if (getElement(str[i], PTTR)) {

int k = 0;

while (i < length) {

index = (int)str[i];

for (k; k < PTTR->child[index]->rebro.length() && i < length; k++) {

if (str[i] == PTTR->child[index]->rebro[k])

{

i++;

}

else {

return 0;

}

}

if (PTTR->child[index]->rebro.length() == k && length != i) {

PTTR = PTTR->child[index];

k = 0;

}

if (getElement(str[i], PTTR) == false && length != i) {

return 0;

}

}

}

if (length == i) {

return PTTR->child[index]->number;

}

else {

return 0;

}

}

string generateString(int lenght, bool repeat) {

string str;

if (repeat) {

char ch = 97 + rand() % 26;

for (int i = 0; i < lenght; i++) {

str += ch;

}

}

else {

for (int i = 0; i < lenght; i++) {

str += 97 + rand() % 26;

}

}

return str + "$";

}

void printMenu()

{

cout << "1 - Ввести строку" << endl

<< "2 - Сгенерировать строку" << endl

<< "3 - Построить дерево" << endl

<< "4 - Вывести дерево" << endl

<< "5 - Поиск" << endl

<< "6 - Очистить экран" << endl

<< "7 - Выход" << endl;

}

int main()

{

char menu;

int index;

int countChar = 100;

int repeat = 0;

string str = "abaabk$";

string searchStr = "$";

Element \*root = new Element(0);

bool exit = false;

while (!exit)

{

printMenu();

cout << "Введите команду: ";

cin >> menu;

switch (menu)

{

case '1':

{

cout << "Введите строку:\n";

cin >> str;

str += "$";

break;

}

case '2':

{

cout << "Введите длину строки для гененрации:" << endl;

cin >> countChar;

cout << "Повторяющие символы (да[1], нет[0]):" << endl;

cin >> repeat;

str = generateString(countChar, (bool)repeat);

break;

}

case '3':

{

system("cls");

cout << "Строка: " << str << endl;

root = new Element(0);

root->rebro = "ROOT";

auto start\_time = std::chrono::steady\_clock::now();

SufixTree(str, root);

auto end\_time = std::chrono::steady\_clock::now();

auto elapsed\_ns = std::chrono::duration\_cast<std::chrono::nanoseconds>(end\_time - start\_time);

cout << "Поиск занял: " << elapsed\_ns.count() << " наносекунд" << endl;

cout << "Количество вершин: " << count0 + 1 << endl;

count0 = 0;

break;

}

case '4':

{

ShowThree(\*root, 0, 0);

break;

}

case '5':

{

cout << "Введите строку:\n";

cin >> searchStr;

auto start\_time = std::chrono::steady\_clock::now();

index = search(searchStr, root);

auto end\_time = std::chrono::steady\_clock::now();

auto elapsed\_ns = std::chrono::duration\_cast<std::chrono::nanoseconds>(end\_time - start\_time);

cout << "Поиск занял: " << elapsed\_ns.count() << " наносекунд\n";

if (search(searchStr, root) == 0) {

cout << "Cтрока не была найдена" << endl;

}

else {

cout << "Подстрока имеет индекс: " << search(searchStr, root) << endl;

}

break;

}

case '6':

{

system("cls");

break;

}

case '7':

{

exit = true;

break;

}

default:

break;

}

}

system("pause");

return 0;

}

# **Литература:**

1. Ландовский В.В. Алгоритмы обработки данных: учеб. пособие. / В.В. Ландовский – Новосибирск: Изд-во НГТУ, 2018. – 64 с.

2. Роберт, Седжвик Алгоритмы на C++. Анализ структуры данных. Сортировка. Поиск. Алгоритмы на графах. Руководство / Седжвик Роберт. - М.: Диалектика / Вильямс, 2016. - 697 c.