МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм Ахо-Корасик

Студент гр. 8303 ____ Хохлов Г.О.

Преподаватель ____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы

Изучение алгоритма Ахо-Корасик для решения задач точного поиска набора образцов и поиска образца с джокером.

Задание 1

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст |T|,1≤|T|≤100000).

Вторая - число n |1≤n≤3000), каждая следующая из n строк содержит шаблон из набора P={p1,…,pn}1≤|pi|≤75

Все строки содержат символы из алфавита {A,C,G,T,N} }

Выход:

Все вхождения образцов из P в T.

Каждое вхождение образца в текст представить в виде двух чисел - i p

Где i - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p

Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

Sample Input:

CCCA

1

CC

Sample Output:

1 1

2 1

Индивидуализация

Вариант 4

Реализовать режим поиска, при котором все найденные образцы не пересекаются в строке поиска (т.е. некоторые вхождения не будут найдены; решение задачи неоднозначно).

Описание алгоритма задания 1

В программе используется алгоритм Ахо-Корасик. Поэтапное выполнение алгоритма выглядит следующим образом:

- Из всех строк-шаблонов, вхождения которых необходимо найти в тексте строится бор — особую структура данных для хранения строк, которая представляет собой дерево, каждая вершина которого представляет один символ. Проходя от корня к терминальной вершине получаем строку-шаблон.
- Затем необходимо построить суффиксные ссылки, которые из конкретной вершины направляют в вершину,представляющую наибольший суффикс текущей строки.
- Запускается поиск, который заключается в том, что проходятся все символы текста по порядку и происходит спуск по бору. Если у текущей вершины есть вершина-ребенок с текущим символом, происходит переход в эту вершину-ребенка. Если же нет, находится максимальный суффикс из которого мы можем перейти в следующую вершину, содержащую текущий символ. Если произошел переход в вершину, помеченную как терминальная, это означает что найдено вхождение шаблона. Для выполнения требования индивидуализации после нахождения очередного вхождения шаблона, текущей вершиной становится корень, для того чтобы избежать пересечений строк-шаблонов в тексте.

Описание функций и структур данных

struct Vertex — структура вершины бора.

Поля Vertex:

- char ch – символ строки;
- Vertex* parent – ссылка на родителя;
- Vertex* suffix - ссылка на суффикс;
- unordered_map<char, Vertex*> child – карта ссылок на детей;
- int terminateId – индекс строки если вершина терминальная,иначе равен -1;
- int stringSize – размер строки если вершина терминальная;
- vector<int> substringStart – индекс вхождения подстроки в шаблон;

class Trie – структура бора

Поля Trie:

- Vertex root – корень бора;
- map<int, set<int>> foundedStrings – карта найденных вхождений строк-шаблонов;

Методы Trie:

- void readStrings() – метод считывания строк-шаблонов из потока ввода.
- void insert(const string& newString, int id) – метод добавления строки в бор. Принимает строку и ее индекс.
- void findStrings(string text) – метод поиска строк в тексте, принимает текст, найденные вхождения записывает в foundedStrings.
- void initSuffixRef() - метод построения суффиксных ссылок в боре, ничего не принимает, изменяет вершины бора.
- void printFoundedStrings() - метод вывода найденных строк в стандартный поток вывода.

Сложность алгоритма

Построение бора выполняется за O(m), где m — суммарная длина всех шаблонов. Для построения суффиксных ссылок используется поиск в ширину. Его сложность O(E+V), но так как количество ребер линейно зависит от количества вершин, а вершины представляют собой одиночные символы, то можем считать сложность как O(2m) = O(m). Поиск строки в тексте линейно проходит текст, следовательно сложность O(n), где n — длина текста.
Итого сложность по времени составляет O(m+n).
Для хранения бора используется не больше, чем O(m) памяти, так как каждый символ представляет собой одну вершину бора. Также хранится карта найденных строк, которая не превышает O(n*k), где k — количество строк-шаблонов. Итого сложность по памяти составляет O(m+n*k).

Задание 2

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с *джокером*.

В шаблоне встречается специальный символ, именуемый джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу P необходимо найти все вхождения P в текст T.

Например, образец ab??c? с джокером ? встречается дважды в тексте xabvccbababcax.

Символ джокер не входит в алфавит, символы которого используются в T. Каждый джокер соответствует одному символу, а не подстроке неопределённой длины. В шаблон входит хотя бы один символ не джокер, т.е. шаблоны вида ??? недопустимы.

Все строки содержат символы из алфавита {A,C,G,T,N} }

Вход:

Текст ‖T,1≤‖T‖≤100000)

Шаблон ‖P,1≤‖P‖≤40)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Sample Input:

ACTAN}CA

A$$A$

$

Sample Output:

1

Описание алгоритма задания 2

Для задачи нахождения шаблона с джокером в текстом, алгоритм действует следующим образом:

- Шаблон разбивается на подстроки, разделенные джокером и они записываются в бор как отдельные строки.
- Также как в первом задании строятся суффиксные ссылки.
- После этого происходит поиск каждой подстроки в тексте. Тогда появление подстроки *Qi* в тексте на позиции *j* будет означать

возможное появление шаблона на позиции $j-li+1$, где li — индекс начала подстроки в шаблоне. Создается массив в котором при нахождении подстроки значение элемента под номером $j-li+1$ увеличивается на единицу. Соответственно индексы таких элементов этого массива, которые равны количеству подстрок и являются вхождениями шаблона.

Сложность алгоритма

Построение бора выполняется за O(m), где m — суммарная длина всех шаблонов (в случае задачи с джокером, m — сумма подстрок в шаблоне, разделенных джокером). Для построения суффиксных ссылок используется поиск в ширину. Его сложность O(E+V), но так как количество ребер линейно зависит от количества вершин, а вершины представляют собой одиночные символы, то можем считать сложность как O(2m) = O(m). Поиск строки в тексте линейно проходит текст, следовательно сложность O(n), где n — длина текста. В алгоритме поиска строки с джокером после прохождения текста еще просматривается конечный массив, но его размер равен n, что не повлияет на ассимптотическую сложность. Итого сложность по времени составляет O(m+n).

Для хранения бора используется не больше, чем O(m) памяти, так как каждый символ представляет собой одну вершину бора. Также хранится карта найденных строк, которая не превышает O(n*k), где k — количество строк-шаблонов. Итого сложность по памяти составляет O(m+n*k).

Описание функций и структур данных

struct Vertex — структура вершины бора.

Поля Vertex:

- char ch – символ строки;
- Vertex* parent – ссылка на родителя;
- Vertex* suffix - ссылка на суффикс;
- unordered_map<char, Vertex*> child – карта ссылок на детей;
- int terminateId – индекс строки если вершина терминальная, иначе равен -1;
- int stringSize – размер строки если вершина терминальная;

Для задачи нахождения строки с джокером дополнительное поле:

- vector<int> substringStart – индекс вхождения подстроки в шаблон;

class Trie – структура бора

Поля Trie:

- Vertex root – корень бора;
- map<int, set<int>> foundedStrings – карта найденных вхождений строк-шаблонов;

Для задачи нахождения строки с джокером дополнительные поля:

- char joker – символ джокера;
- int tempSize – длина строки-шаблона;
- int substringsNum – количество подстрок, разделенных джокером;

Методы Trie:

- void readStrings() – метод считывания строк-шаблонов из потока ввода.
- void insert(const string& newString, int id) – метод добавления строки в бор. Принимает строку и ее индекс.
- void findStrings(string text) – метод поиска строк в тексте, принимает текст, найденные вхождения записывает в foundedStrings.
- void initSuffixRef() - метод построения суффиксных ссылок в боре, ничего не принимает, изменяет вершины бора.
- void printFoundedStrings() - метод вывода найденных строк в стандартный поток вывода.

Тестирование

Задание 1:

Тест 1:

```
ABCASDTEAD
5
ABC
CAS
ASD
TEA
EAD
```

Вывод:

```
1 1
4 3
7 4
```

Тест 2:

```
CATNATCAT
3
ATN
NAT
CAT
```

Вывод:

```
1 3
4 2
7 3
```

Тест 3:

```
CCCA
1
CC
```

Вывод:

```
1 1
```

Тест 4:

```
ABABABABABABAB
1
ABA
```

Вывод:

```
1 1
5 1
9 1
```

Тест с подробным промежуточным выводом:

```
NATCADCAT
3
NAT
TCA
CAT

add new string "NAT" to trie
path in trie : root->(New)N->(New)A->(New)T

add new string "TCA" to trie
path in trie : root->(New)T->(New)C->(New)A

add new string "CAT" to trie
path in trie : root->(New)C->(New)A->(New)T
initialization of suffix links:
root:
    childs: CNT
C   childs: A
    suffixLink: root
N   childs: A
    suffixLink: root
T   childs: C
    suffixLink: root
CA  childs: T
    suffixLink: root
NA  childs: T
    suffixLink: root
TC  childs: A
    suffixLink: C
CAT childs: none
    suffixLink: T
NAT childs: none
    suffixLink: T
TCA childs: none
    suffixLink: CA

Answer:
1 1
7 3
```

Задание 2:

Тест 1:

```
ACTANCA
A$$A
$
```

Вывод:

```
1
```

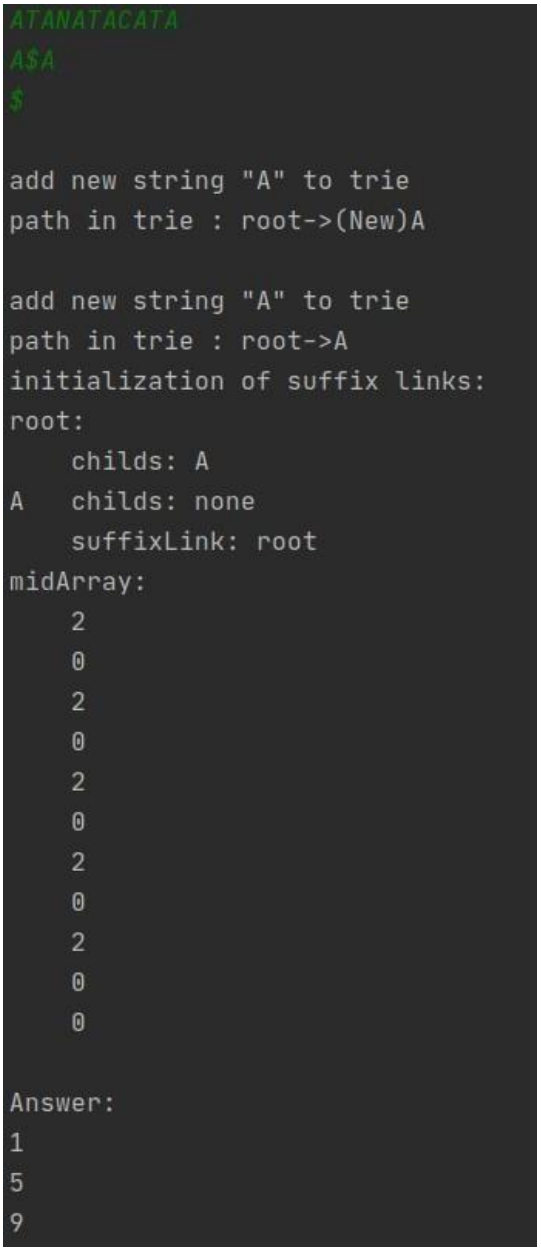
Тест 2:

```
CATNATCAT
#AT
#
```

Вывод:

```
1
4
7
```

Тест с подробным промежуточным выводом:



```
ATANATACATA
A$A
$

add new string "A" to trie
path in trie : root->(New)A

add new string "A" to trie
path in trie : root->A
initialization of suffix links:
root:
    childs: A
A   childs: none
    suffixLink: root
midArray:
    2
    0
    2
    0
    2
    0
    2
    0
    2
    0
    0

Answer:
1
5
9
```

Вывод

В ходе выполнения лабораторной работы был изучен алгоритм Ахо-Корасик и использован для нахождения вхождений множества строк в тексте, а также для нахождения шаблона с джокером.

Приложение А

Исходный код piaa_5_1.cpp

```
#include <iostream>
#include <unordered_map>
#include <map>
#include <set>
#include <queue>
#include <string>
#define DBG
using namespace std;

//структура вершины бора. Хранит символ, ссылки на родителя, на суффикс и на детей,
//а также индекс шаблона. Если строка не терминальная, хранится -1. Также для удобства в терминальных вершинах
//хранится длина шаблона.
struct Vertex {
    char ch;
    Vertex* parent = nullptr;
    Vertex* suffix = nullptr;
    unordered_map<char, Vertex*> child;
    int terminateId = -1;
    int stringSize;
};

//класс бора
class Trie {
    Vertex root;
    map<int, set<int>> foundedStrings;
public:
    //метод для поиска строк в тексте, принимает текст.
    void findStrings(string text) {
        //начинаем поиск от корня бора
        Vertex* tmp = &root;
        for (int i = 0; i < text.length(); i++) {
            //если можем перейти из текущей вершины, то делаем это, иначе пытаемся перейти из суффиксов
            if (tmp->child.find(text[i]) != tmp->child.end()) {
                tmp = tmp->child[text[i]];
                  //если вершина терминальная, записываем индекс ее начала в тексте и возвращаемся
                      //к корню чтобы избежать пересечения вхождений шаблонов
```

```
                    if (tmp->terminateId != -1) {
                                        foundedStrings[i + 2 - tmp-
>stringSize].insert(tmp->terminateId);
                        tmp = &root;
                    }
                } else {
                        //если мы не можем перейти из текущей вершины,
проверяем, можем ли перейти из ее суффиксов
                    while (tmp->child.find(text[i]) == tmp->child.end()
&& tmp != &root) {
                        tmp = tmp->suffix;
                    }
                    if (tmp->child.find(text[i]) != tmp->child.end())
                        i--;
                }
            }
        }
        //метод для вывода найденных строк в тексте
        void printFoundedStrings() {
    #ifdef DBG
            cout << "\nAnswer:\n";
    #endif
            for (auto i : foundedStrings) {
                for (auto j : i.second) {
                    cout << i.first << ' ' << j << '\n';
                }
            }
        }

        //метод построения суффиксных ссылок в боре
        void initSuffixRef() {
            Vertex* vert;
            queue<Vertex*> vertexToVisit;
            vertexToVisit.push(&root);
             //запускается поиск в ширину, на каждом шаге для родителя
текущей вершины суффиксная ссылка уже построена,
             //а значит остается только проверить можем ли из суффикса
родителя перейти по символу текущей вершины,
             //если не можем то уменьшаем суффикс родителя и проверяем
так пока не дойдем до корня
    #ifdef DBG
            cout << "initialization of suffix links:\n";
    #endif
```

```
            while (!vertexToVisit.empty()) {
                vert = vertexToVisit.front();
                vertexToVisit.pop();
    #ifdef DBG
                if (vert == &root) {
                    cout << "root:\n";
                } else {
                    Vertex* t = vert;
                    vector<char> str;
                    while (t!=&root) {
                        str.push_back(t->ch);
                        t = t->parent;
                    }
                    for (auto k = str.rbegin(); k!=str.rend(); k++){
                        cout << *k;
                    }
                }
                cout << "\tchilds: ";
                if (vert->child.empty()) {
                    cout << "none";
                }
    #endif
                for (auto &i : vert->child) {
                    vertexToVisit.push(i.second);
    #ifdef DBG
                    cout << i.first;
    #endif
                }
                Vertex* tmp = vert->parent;
    #ifdef DBG
                cout << endl;
                if (vert != &root)
                    cout << "\tsuffixLink: ";
    #endif
                while (tmp) {
                    tmp = tmp->suffix;
                         if (tmp && tmp->child.find(vert->ch) !=  tmp-
>suffix->child.end()) {
                        vert->suffix = tmp->child[vert->ch];
    #ifdef DBG
                        if (vert->suffix != &root) {
                            vector<char> suffix;
                            suffix.push_back(vert->ch);
```

```
                                while (tmp!=&root) {
                                    suffix.push_back(tmp->ch);
                                    tmp = tmp->parent;
                                }
                                     for (auto k = suffix.rbegin();  k!
=suffix.rend(); k++){
                                    cout << *k;
                                }
                                cout << endl;
                            }
      #endif
                            break;
                        }
                    }
      #ifdef DBG
                    if (vert->suffix == &root)
                        cout << "root\n";
      #endif
                }
            }
            //метод добавления новой строки в бор
            void insert(const string& newString, int id) {
      #ifdef DBG
                 cout << "\nadd new string \"" << newString << "\" to  trie\
n";
                cout << "path in trie : root";
      #endif
                Vertex* tmp = &root;
                 //спускаемся по бору для каждого символа, если вершина  не
существует, создаем ее
                for (auto i : newString) {
      #ifdef DBG
                    cout << "->";
      #endif
                    if (tmp->child.find(i) == tmp->child.end()) {
                        tmp->child[i] = new Vertex;
                        tmp->child[i]->parent = tmp;
                        tmp->child[i]->ch = i;
                        tmp->child[i]->suffix = &root;
      #ifdef DBG
                        cout << "(New)";
      #endif
                    }
```

```
                tmp = tmp->child[i];

    #ifdef DBG
                cout << tmp->ch;
    #endif
            }

    #ifdef DBG
            cout << endl;
    #endif
               //выставляем для терминальной вершины индекс шаблона и
записываем длину шаблона
                tmp->terminateId = id;
                tmp->stringSize = newString.length();
            }
            //метод считывания строк-шаблонов из потока ввода
            void readStrings(){
                int numOfWords;
                vector<string> strings;
                cin >> numOfWords;
                for (int i = 0; i < numOfWords; i++) {
                    string newString;
                    cin >> newString;
                    strings.push_back(newString);
                }
                for (int i = 0; i < numOfWords; i++) {
                    insert(strings[i], i + 1);
                }
            }

    };

    int main() {
        string text;
        cin >> text;
        Trie trie;
        trie.readStrings();
        trie.initSuffixRef();
        trie.findStrings(text);
        trie.printFoundedStrings();
        return 0;
    }
```

Исходный код piaa_5_2.cpp

```
#include <iostream>
#include <unordered_map>
#include <set>
#include <queue>
#include <string>
#define DBG
using namespace std;

//структура вершины бора. Хранит символ, ссылки на родителя, на суффикс и на детей,
//а также индекс шаблона. Если строка не терминальная, хранится -1. Также для удобства в терминальных вершинах
//хранится длина шаблона.
struct Vertex {
    char ch;
    Vertex* parent = nullptr;
    Vertex* suffix = nullptr;
    unordered_map<char, Vertex*> child;
    int terminate = 0;
    int  stringSize;
    vector<int> substringStart;
};

//класс бора
class Trie {
    Vertex root;
    char joker;
    int tempSize;
    int substringsNum = 0;
public:
    //метод для поиска строк в тексте, принимает текст.
    void findStrings(string text) {
        //начинаем поиск от корня бора
        Vertex* tmp = &root;
        int substringsEntries[text.size()+1];
        for (int i = 0; i <= text.size(); i++) {
            substringsEntries[i] = 0;
        }
        for (int i = 0; i < text.length(); i++) {
            //если можем перейти из текущей вершины, то делаем это, иначе пытаемся перейти из суффиксов
            if (tmp->child.find(text[i]) != tmp->child.end()) {
```

```
                    tmp = tmp->child[text[i]];
                     //если вершина терминальная, записываем индекс ее
начала в тексте и возвращаемся
                        //к корню чтобы избежать пересечения вхождений
шаблонов
                    if (tmp->terminate) {
                        for (auto j : tmp->substringStart) {
                            if (text.size() - i + j + tmp->stringSize >
tempSize)
                                substringsEntries[i - tmp->stringSize -
j + 2] += 1;
                        }
                    }
                    Vertex* suff = tmp;
                    //также проверяем, не является ли терминальным один
из суффиксов
                    while (suff != &root) {
                        suff = suff->suffix;
                        if (suff->terminate) {
                            for (auto j : suff->substringStart) {
                                    if (text.size() - i + j + suff-
>stringSize > tempSize)
                                        substringsEntries[i - suff-
>stringSize - j + 2] += 1;
                            }
                        }
                    }
                } else {
                        //если мы не можем перейти из текущей вершины,
проверяем, можем ли перейти из ее суффиксов
                    while (tmp->child.find(text[i]) == tmp->child.end()
&& tmp != &root) {
                        tmp = tmp->suffix;
                    }
                    if (tmp->child.find(text[i]) != tmp->child.end())
                        i--;
                }
            }

            printEntries(substringsEntries, text.size());
        }
        //метод для вывода найденных вхождений в тексте
```

```
          void printEntries(const int* substringsEntries, int textSize)
const {
     #ifdef DBG
             cout << "midArray:\n";
             for (int i = 1; i <= textSize; i++) {
                 cout << '\t' << substringsEntries[i] << endl;
             }
             cout << "\nAnswer:\n";
     #endif
             int prev = 0;
             for (int i = 1; i <= textSize; i++) {
                  if (substringsEntries[i] == substringsNum && (prev==0
|| prev+tempSize <= i)) {
                      prev = i;
                      cout << i << endl;
                  }
             }
         }
         //метод построения суффиксных ссылок в боре
         void initSuffixRef() {
             Vertex* vert;
             queue<Vertex*> vertexToVisit;
             vertexToVisit.push(&root);
              //запускается поиск в ширину, на каждом шаге для родителя
текущей вершины суффиксная ссылка уже построена,
              //а значит остается только проверить можем ли из суффикса
родителя перейти по символу текущей вершины,
              //если не можем то уменьшаем суффикс родителя и проверяем
так пока не дойдем до корня
     #ifdef DBG
             cout << "initialization of suffix links:\n";
     #endif
             while (!vertexToVisit.empty()) {
                 vert = vertexToVisit.front();
                 vertexToVisit.pop();
     #ifdef DBG
                 if (vert == &root) {
                     cout << "root:\n";
                 } else {
                     Vertex* t = vert;
                     vector<char> str;
                     while (t!=&root) {
                         str.push_back(t->ch);
```

```
                        t = t->parent;
                    }
                    for (auto k = str.rbegin(); k!=str.rend(); k++){
                        cout << *k;
                    }
                }
                cout << "\tchilds: ";
                if (vert->child.empty()) {
                    cout << "none";
                }
    #endif
                for (auto &i : vert->child) {
                    vertexToVisit.push(i.second);
    #ifdef DBG
                    cout << i.first;
    #endif
                }
                Vertex* tmp = vert->parent;
    #ifdef DBG
                cout << endl;
                if (vert != &root)
                    cout << "\tsuffixLink: ";
    #endif
                while (tmp) {
                    tmp = tmp->suffix;
                          if (tmp && tmp->child.find(vert->ch) !=  tmp-
>suffix->child.end()) {
                        vert->suffix = tmp->child[vert->ch];
    #ifdef DBG
                        if (vert->suffix != &root) {
                            vector<char> suffix;
                            suffix.push_back(vert->ch);
                            while (tmp!=&root) {
                                suffix.push_back(tmp->ch);
                                tmp = tmp->parent;
                            }
                                    for (auto k = suffix.rbegin();  k!
=suffix.rend(); k++){
                                cout << *k;
                            }
                            cout << endl;
                        }
    #endif
```

```
                        break;
                    }
                }
    #ifdef DBG
                if (vert->suffix == &root)
                    cout << "root\n";
    #endif
            }
        }
        void insert(const string& newString, int start) {
    #ifdef DBG
             cout << "\nadd new string \"" << newString << "\" to  trie\
n";
            cout << "path in trie : root";
    #endif
            Vertex* tmp = &root;
            for (auto i : newString) {
    #ifdef DBG
                cout << "->";
    #endif
                if (tmp->child.find(i) == tmp->child.end()) {
                    tmp->child[i] = new Vertex;
                    tmp->child[i]->parent = tmp;
                    tmp->child[i]->ch = i;
                    tmp->child[i]->suffix = &root;
    #ifdef DBG
                    cout << "(New)";
    #endif
                }
                tmp = tmp->child[i];

    #ifdef DBG
                cout << tmp->ch;
    #endif
            }

    #ifdef DBG
            cout << endl;
    #endif
             //выставляем для терминальной вершины флаг о том, что  она
терминальная
            //записываем длину подстроки и индекс вхождения в шаблон
            tmp->terminate = 1;
```

```
            tmp->stringSize = newString.size();
            tmp->substringStart.push_back(start);
        }
        //метод считывания строк-шаблонов из потока ввода
        void readStrings() {
            string temp;
            cin >> temp;
            cin >> joker;
            tempSize = temp.size();
            int start = 0;
            bool sub = false;
            for (int i = 0; i < temp.size(); i++) {
                if (temp[i] != joker && !sub) {
                    sub = true;
                    start = i;
                    if (i == temp.size() - 1) {
                        insert(temp.substr(start, 1), start);
                        substringsNum++;
                    }
                 } else if ((temp[i] == joker || i == temp.size() -  1)
&& sub) {
                    sub = false;
                          insert(temp.substr(start, i-start+(temp[i] ==
joker?0:1)), start);
                    substringsNum++;
                }
            }

        }
    };

    int main() {
        string text;
        cin >> text;
        Trie trie;
        trie.readStrings();
        trie.initSuffixRef();
        trie.findStrings(text);
        return 0;
    }
```