**ОТЧЕТ**

о выполнении долгосрочного домашнего задания

по дисциплине:

«Технологии и методы программирования»

на тему: «Поиск подстроки в строке. Алгоритм Рабина-Карпа»

Выполнил:

Лимасов С.В.\_\_\_\_\_\_\_\_

(подпись)

Проверил:

Кузнецов А.В.\_\_\_\_\_\_\_

(подпись)

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_

2019 г.

Оглавление

[**1. Описание задания 3**](#_Toc5307549)

[**2. Математическая модель, теоретическое описание построения и вычислительная сложность 3**](#_Toc5307550)

[**3. Тестирование структуры 4**](#_Toc5307551)

[**Заключение 6**](#_Toc5307552)

[Приложение Листинг файла BST.h 7](#_Toc5307553)

## Описание задания

Задача заключается в реализации алгоритма поиска в строке. А так же в оценке сложности операции поиска подстроки в алгоритме Рабина-Карпа.

Алгоритм Рабина — Карпа— это алгоритм [поиска строки](https://ru.wikipedia.org/wiki/Поиск_подстроки), который ищет шаблон, то есть подстроку, в тексте, используя [хеширование](https://ru.wikipedia.org/wiki/Хеширование). Он был разработан в [1987 году](https://ru.wikipedia.org/wiki/1987_год) [Майклом Рабином](https://ru.wikipedia.org/wiki/Рабин,_Майкл_Озер) и [Ричардом Карпом](https://ru.wikipedia.org/wiki/Карп,_Ричард).

## Математическая модель, теоретическое описание построения и вычислительная сложность

Алгоритм редко используется для поиска одиночного шаблона, но имеет значительную теоретическую важность и очень эиффективен в поиске совпадений множественных шаблонов одинаковой длины. Для текста длины n и шаблона длины m его среднее и лучшее время исполнения равно [O](https://ru.wikipedia.org/wiki/O_нотация)(n) при правильном выборе хеш-функции (смотрите ниже), но в худшем случае он имеет эффективность O(nm), что является одной из причин того, почему он не слишком широко используется. Для приложений, в которых допустимы ложные срабатывания при поиске, то есть, когда некоторые из найденных вхождений шаблона на самом деле могут не соответствовать шаблону, алгоритм Рабина — Карпа работает за гарантированное время O(n) и при подходящем выборе рандомизированной хеш-функции (смотрите ниже) вероятность ошибки можно сделать очень малой. Также алгоритм имеет уникальную особенность находить любую из заданных k строк одинаковой длины в среднем (при правильном выборе хеш-функции) за время O(n) независимо от размера k.

Ключами к производительности алгоритма Рабина — Карпа являются низкая вероятность коллизий и эффективное вычисление хеш-значения последовательных подстрок текста. Рабин и Карп[1] предложили использовать так называемый полиномиальный хеш (хотя любой другой кольцевой хеш также подойдёт). Для данного шаблона  такой хеш определён следующим образом:



где  q — некоторое простое число, а x — число от 0 до q-1. Хеш-значения последовательных подстрок s[i...i+m-1] и s[i+z..i+m] для полиномиального хеша вычисляются следующим образом (заметим, что для эффективности число  считается перед основной процедурой поиска алгоритма Рабина — Карпа):



Рассмотрим, на примере BST, основные методы, реализованные в дереве и вычислительные сложности их выполнения (см. таблицу 1.1).

Таблица 1.1 – Используемые методы

|  |  |  |
| --- | --- | --- |
| Прототип функции | Описание | Сложность |
| void Insert(int key,T value); | Вставляет элемент по ключу с правилами бинарного дерева | O(ln n) |
| int Find(int key); | Возвращает узел с заданным значением ключа key | O(ln n) |

## Тестирование структуры

Для проверки структуры бинарного дерева поиска (BST) проведем эксперименты по поиску элементов в дереве, состоящем из разного числа элементов.

Проведем пять экспериментов по поиску элементов в бинарном дереве поиска из n элементов (см. таблицу 2.1).

Таблица 2.1 – Измерение времени поиска элемента в дереве из n элементов.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Кол-во элементов | Измеренное  время заполнения  (с) | | | | |
| 1 эксперимент | 2 эксперимент | 3 эксперимент | 4 эксперимент | 5 эксперимент |
| 100 | 0,065 | 0,063 | 0,066 | 0,064 | 0,064 |
| 1000 | 0,103 | 0,093 | 0,093 | 0,094 | 0,095 |
| 10000 | 0,143 | 0,141 | 0,143 | 0,141 | 0,142 |
| 100000 | 0,182 | 0,182 | 0,183 | 0,185 | 0,183 |
| 200000 | 0,219 | 0,217 | 0,218 | 0,215 | 0,216 |

Результаты практических опытов наглядно представлены на графике (см. рисунок 2.1).

Рисунок 2.1 – График измерения времени поиска элемента.

График зависимости возрастает как O(ln n), что совпадает с теорией.

## Заключение

С точки зрения построения алгоритма, операции вставки тесно связана с поиском элемента в дереве. График функции выше показывает, что алгоритм работает с вычислительной сложность O(ln n), что совпадает с теоретическими значениями.

Выводы: в результате проделанной работы я получил навыки работы со средой CLion, повторил и подтвердил пройденный мною лекционный материал, освоил методы оценки сложности алгоритмов, приобрел опыт в анализе оценки времени работы алгоритмов.

# Приложение Листинг файла BST.h

#include <exception>

template <typename T>

struct Node{

int \_key;

T \_value;

Node<T>\* \_left;

Node<T>\* \_right;

Node(int key,T value)

{

\_key=key;\_value=value; \_left=nullptr; \_right=nullptr;

}

int size()

{

return (\_left == NULL ? 0 : \_left->size()) + (\_right == NULL ? 0 : \_right->size()) + 1;

}

};

template <typename T>

class BST

{

private:

Node<T> \*\_root;

public:

BST() { \_root = nullptr; }

BST(int key, T value) { Node<T> \*t = new Node<T>(key, value); \_root = t;}

T Find(int key);

void Insert(int key, T value);

int size()

{

return \_root == nullptr ? 0 : \_root->size();

}

};

template<typename T>

T BST<T>::Find(int key)

{

Node<T>\* target=\_root;

while((target!=nullptr)&&(key!=target->\_key))

{

if (key<target->\_key)

target=target->\_left;

else target=target->\_right;

}

if (target== nullptr)

throw "element not found";

return target->\_value;

}

template<typename T>

void BST<T>::Insert(int key, T value)

{

Node<T> \*buff = \_root;

if (buff == nullptr)

{

Node<T> \*element = new Node<T>(key, value);

\_root = element;

}

else

{

while (true)

{

if (buff->\_key <= key)

{

if (buff->\_right != nullptr)

buff = buff->\_right;

else

{

Node<T> \*element = new Node<T>(key, value);

buff->\_right = element;

break;

}

}

else

{

if (buff->\_left != nullptr)

buff = buff->\_left;

else

{

Node<T> \*element = new Node<T>(key, value);

buff->\_left = element;

break;

}

}

}

}

};