БУ ВО Ханты-Мансийского автономного округа – Югры

«Сургутский государственный университет»

Политехнический институт

Кафедра автоматики и компьютерных систем

ОТЧЁТ

по лабораторной работе №5

по дисциплине: «Алгоритмы и структуры данных»

Выполнил: студент(ка) группы №609-31,

Гаврилов Е.Е.

Принял: старший преподаватель кафедры АиКС

Назаров Е.В.

Сургут

2025г.

**Цель работы:** изучить базовые алгоритмы работы с деревьями: построение, обход, поиска элемента, удаление элемента, подсчет количества узлов, нахождение высоты дерева, исследовать свойства деревьев, закрепить навыки структурного программирования.

**Задание:**

1. Реализовать функции вставки, поиска и удаления узла, обхода дерева, вставки в корень, вывода дерева на экран, нахождения высоты дерева и количества узлов, а также функцию вставки, строящую рандомизированное дерево.

2. Построить зависимости высоты деревьев (обычного бинарного поиска и рандомизированного) от количества ключей (ключи – случайные и упорядоченные величины), полагая, что ключи целые числа.

3. Реализовать заданную функцию (в соответствии с вариантом: T – тип ключей, D – диапазон изменения значений ключей).

4. Составить отчет, в котором привести графики полученных зависимостей, анализ свойств алгоритмов, листинги функций вставки, поиска, удаления узла с комментариями и выводы по работе.

Ход работы:

1.Для реализации выбран алгоритм Кнута-Морриса-Пратта.

Алгоритм основан на предварительном создании префикс-функции (массива). Префикс-функция от строки s равна массиву π, где π[i] обозначает длину максимального префикса строки s[0..i], совпадающего с её суффиксом. Массив содержит информацию, о том, где в строке повторно встречается ее префикс. Таким образом можно избежать проверки заведомо недопустимых сдвигов.

2.Измерение времени работы алгоритмов:

Таблица 1. Среднее время поиска от размера текста при фиксированном размере образца для реализованного алгоритма и функции strstr, для успешного поиска.

|  |  |  |
| --- | --- | --- |
| Длинна текста | Время работы алгоритма (нс) | Время работы функции strstr (нс) |
| 5000 | 13380 | 4506 |
| 10000 | 23425 | 6827 |
| 15000 | 26319 | 6847 |
| 20000 | 33364 | 8708 |
| 25000 | 36066 | 8988 |
| 30000 | 35444 | 8681 |
| 35000 | 38122 | 8874 |
| 40000 | 36744 | 8773 |
| 45000 | 37598 | 8991 |
| 50000 | 39796 | 9997 |

Рис 1. Зависимость времени работы алгоритмов поиска от длинны текста (успешный поиск).

Таблица 2. Среднее время поиска от размера текста при фиксированном размере образца для реализованного алгоритма и функции strstr, для неуспешного поиска.

|  |  |  |
| --- | --- | --- |
| Длинна текста | Время работы алгоритма (нс) | Время работы функции strstr (нс) |
| 5000 | 25878 | 5939 |
| 10000 | 51090 | 10811 |
| 15000 | 69544 | 12033 |
| 20000 | 96606 | 16998 |
| 25000 | 120818 | 20281 |
| 30000 | 139297 | 21718 |
| 35000 | 163689 | 24267 |
| 40000 | 181210 | 26097 |
| 45000 | 206086 | 30724 |
| 50000 | 220961 | 39926 |

Рис 2. Зависимость времени работы алгоритмов поиска от длинны текста (неуспешный поиск).

Таблица 3. Среднее время поиска от размера искомой строки при фиксированном размере текста для реализованного алгоритма и функции strstr, для успешного поиска.

|  |  |  |
| --- | --- | --- |
| Длина строки | Время работы алгоритма(нс) | Время работы strstr(нс) |
| 5 | 38670 | 9682 |
| 10 | 119965 | 21263 |
| 15 | 125758 | 21840 |
| 20 | 122988 | 20565 |
| 25 | 117817 | 18381 |
| 30 | 116275 | 16299 |
| 35 | 118608 | 15529 |
| 40 | 118373 | 14112 |
| 45 | 120457 | 12957 |
| 50 | 123415 | 12061 |
| 55 | 125460 | 11931 |
| 60 | 118081 | 11087 |
| 65 | 124440 | 11217 |
| 70 | 122545 | 10416 |
| 75 | 117273 | 10236 |
| 80 | 119545 | 10004 |
| 85 | 125228 | 10145 |
| 90 | 120202 | 9610 |
| 95 | 126349 | 9778 |
| 100 | 126572 | 9675 |

Таблица 4. Среднее время поиска от размера искомой строки при фиксированном размере текста для реализованного алгоритма и функции strstr, для неуспешного поиска.

|  |  |  |
| --- | --- | --- |
| Длина строки | ВР алгоритма(нс) | ВР strstr(нс) |
| 5 | 149353 | 26325 |
| 10 | 144124 | 22999 |
| 15 | 146122 | 23229 |
| 20 | 140637 | 22687 |
| 25 | 146227 | 20524 |
| 30 | 142678 | 19408 |
| 35 | 145991 | 17727 |
| 40 | 142836 | 17673 |
| 45 | 145486 | 14256 |
| 50 | 148351 | 13595 |
| 55 | 145248 | 12835 |
| 60 | 143971 | 12856 |
| 65 | 144020 | 13400 |
| 70 | 146003 | 11976 |
| 75 | 144821 | 11370 |
| 80 | 146296 | 10954 |
| 85 | 149030 | 11494 |
| 90 | 154830 | 11388 |
| 95 | 150526 | 10416 |
| 100 | 148259 | 10357 |

Рис 3. Зависимость времени работы алгоритмов поиска от длинны строки (успешный поиск).

Рис 4. Зависимость времени работы алгоритмов поиска от длинны строки (неуспешный поиск).

3. Анализ

Алгоритм поиска Кнута-Мориса-Пратта показывает большее время, чем функция стандартной библиотеки, но похожую временную сложность при изменении длины текста и длины строки, при успешном и неуспешном поиске.

**Вывод:** была разработана функция для поиска подстроки в строке, реализующая алгоритм Кнута-Мориса-Пратта. Измерена зависимость времени работы алгоритма от длины текста и от длины строки. Произведено сравнение с аналогичными зависимостями для функции стандартной библиотеки strstr. Реализованный алгоритм показывает большее время поиска, но схожую временную сложность.

Листинг 1. arrgen.h

#ifndef ARRGEN\_H

#define ARRGEN\_H

#include <iostream>

#include <random>

#include <cmath>

#include <fstream>

namespace ArrGen

{

template <typename T>

void IncreasingSequence(T \*array, size\_t len, T min, T max);

template <typename T>

void DecreasingSequence(T \*array, size\_t len, T min, T max);

template <typename T>

void RandomSequence(T \*array, size\_t len, T min, T max);

template <typename T>

void SawtoothSequence(T \*array, size\_t len, T start, T end, size\_t period, size\_t offset);

template <typename T>

void SinSequence(T \*array, size\_t len, T start, T end, double period = 2 \* M\_2\_PI, T offset = 0);

template <typename T>

void StepSequence(T \*array, size\_t len, T start, T end, size\_t period = 10);

template <typename T>

void QuasiOrderedSequence(T \*array, size\_t len, T start, T end, T spread);

template <typename T>

void SawtoothSequence(T \*array, size\_t len, T start, T end);

template <typename T>

void SinSequence(T \*array, size\_t len, T start, T end);

template <typename T>

void StepSequence(T \*array, size\_t len, T start, T end);

template <typename T>

void QuasiOrderedSequence(T \*array, size\_t len, T start, T end);

extern void (\*GenFunArrInt[])(int \*, size\_t, int, int);

extern void (\*GenFunarrDouble[])(double \*, size\_t, double, double);

extern std::string GenFunNames[];

}

#endif

Листинг 2. arrgen.cpp

#include "arrgen.h"

#include <string>

namespace ArrGen

{

template <typename T>

void IncreasingSequence(T \* array, size\_t len, T min, T max)

{

if (nullptr == array || 0 == len)

return;

if (min > max)

{

T temp = min;

min = max;

max = temp;

}

double step = 1.0 \* (max - min) / len;

array[0] = (T)min;

double prev = (T)min;

for (size\_t i = 1; i < len; i++)

{

array[i] = prev + step;

prev += step;

}

}

template <typename T>

void DecreasingSequence(T \* array, size\_t len, T min, T max)

{

if (nullptr == array || 0 == len)

return;

if (min > max)

{

T temp = min;

min = max;

max = temp;

}

double step = 1.0 \* (max - min) / len;

array[0] = max;

double prev = (T)min;

for (size\_t i = 1; i < len; i++)

{

array[i] = prev + step;

prev += step;

}

}

template <typename T>

void RandomSequence(T \* array, size\_t len, T min, T max)

{

if (nullptr == array || 0 == len)

return;

if (min > max)

{

T temp = min;

min = max;

max = temp;

}

std::random\_device random\_device;

std::mt19937 generator(random\_device());

std::uniform\_real\_distribution distribution((double)min, (double)max);

for (size\_t i = 0; i < len; i++)

{

array[i] = distribution(generator);

}

}

template <typename T>

void SawtoothSequence(T \* array, size\_t len, T start, T end, size\_t period, size\_t offset)

{

if (nullptr == array || 0 == len)

return;

double local\_step = 1.0 \* (end - start) / period;

for (size\_t i = 0; i < len; i++)

{

array[i] = start + local\_step \* ((i + offset) % period);

}

}

template <typename T>

void SinSequence(T \* array, size\_t len, T start, T end, double period, T offset)

{

if (nullptr == array || 0 == len)

return;

for (size\_t i = 0; i < len; i++)

{

array[i] = sin(i \* M\_2\_PI \* len / period / period + offset) \* (end - start) / 2 + (end + start) / 2;

}

}

template <typename T>

void StepSequence(T \* array, size\_t len, T start, T end, size\_t period)

{

if (nullptr == array || 0 == len || 0 == period || len < period)

return;

std::random\_device random\_device;

std::mt19937 generator(random\_device());

std::uniform\_real\_distribution distribution((double)0, (double)(end - start) / (len / period));

for (size\_t i = 0; i < len; i++)

{

double ran = distribution(generator);

array[i] = (i / period) \* (end - start) / (len / period) + ran;

}

}

template <typename T>

void QuasiOrderedSequence(T \* array, size\_t len, T start, T end, T spread)

{

if (nullptr == array || 0 == len)

return;

double step = (end - start) / len;

std::random\_device random\_device;

std::mt19937 generator(random\_device());

T max = start > end ? start : end;

T min = start < end ? start : end;

double direct = start;

for (size\_t i = 0; i < len; i++, direct += step)

{

direct = start + 1.0 \* i \* (end - start) / len;

double up\_limit = max > direct + spread ? direct + spread : max;

double low\_limit = min < direct - spread ? direct - spread : min;

std::uniform\_real\_distribution distribution(low\_limit, up\_limit);

double ran = distribution(generator);

array[i] = ran;

}

}

template <typename T>

void SawtoothSequence(T \* array, size\_t len, T start, T end)

{

SawtoothSequence(array, len, start, end, 10, 0);

}

template <typename T>

void SinSequence(T \* array, size\_t len, T start, T end)

{

SinSequence<T>(array, len, start, end, 10 \* M\_2\_PI, 0);

}

template <typename T>

void StepSequence(T \* array, size\_t len, T start, T end)

{

StepSequence<T>(array, len, start, end, 10);

}

template <typename T>

void QuasiOrderedSequence(T \* array, size\_t len, T start, T end)

{

QuasiOrderedSequence<T>(array, len, start, end, 10);

}

}

Листинг 4. lab04.cpp

#include <chrono>

#include <cmath>

#include <cstring>

#include <fstream>

#include <iostream>

#include <random>

#include <vector>

#include <cstring>

using namespace std;

using namespace std::chrono;

char RandChar(char a, char b) { return rand() % (b - a + 1) + a; }

int RandInt(int a, int b) { return rand() % (b - a + 1) + a; }

void RandGen(char \*string, size\_t size) {

for (size\_t i = 0; i < size; i++) {

string[i] = RandChar('A', 'F');

}

}

char \*LasySearch(char \*string, int n, char \*substring, int m) {

int j;

int i;

for (i = 0; i < n - m; i++) {

for (j = 0; j < m && \*(substring + j) == \*(string + i + j); j++)

;

if (j == m)

return string + i;

}

return nullptr;

}

char \*KMPSearch(char \*string, int n, char \*substring, int m) {

int \*Next, i, j;

char \*result = nullptr;

Next = (int \*)malloc(m \* sizeof(int));

for (Next[0] = j = -1, i = 1; i < m; i++) {

for (; j > -1 && substring[j + 1] != substring[i]; j = Next[j])

;

if (substring[j + 1] == substring[i])

j++;

Next[i] = j;

}

for (j = -1, i = 0; i < n && nullptr == result; i++) {

for (; j > -1 && substring[j + 1] != string[i] && nullptr == result; j = Next[j])

;

if (substring[j + 1] == string[i])

j++;

if (j == m - 1) {

result = string + i - j;

}

}

free(Next);

return result;

}

int main() {

srand(time(0));

int repeat = 10000;

int str\_len = 5;

int text\_len;

ofstream output\_TY("ouput\_TY.txt"); //different text, find

ofstream output\_TN("output\_TN.txt"); //different text, not find

ofstream output\_SY("ouput\_SY.txt"); //different string, find

ofstream output\_SN("output\_SN.txt"); // different string, not find

ofstream log("log.txt");

auto start = steady\_clock::now();

auto end = steady\_clock::now();

size\_t fun\_time, str\_time;

size\_t fun\_time\_succ, fun\_time\_fail, str\_time\_succ, str\_time\_fail;

size\_t succ\_times, fail\_times;

char \*text;

char \*str;

for (int i = 5; i <= 50; i += 5) {

text\_len = i \* 1000;

text = (char \*)calloc(text\_len + 1, sizeof(char));

RandGen(text, text\_len);

str = (char \*)calloc(str\_len + 1, sizeof(char));

fun\_time\_succ = 0;

fun\_time\_fail = 0;

str\_time\_succ = 0;

str\_time\_fail = 0;

fail\_times = 0;

succ\_times = 0;

for (int j = 0; j < repeat; j++) {

fun\_time = 0;

str\_time = 0;

char \*fun\_res;

char \*str\_res;

RandGen(str, str\_len);

str[str\_len / 2] = 'F' + (j % 2);

start = steady\_clock::now();

str\_res = strstr(text, str);

end = steady\_clock::now();

str\_time = duration\_cast<nanoseconds>(end - start).count();

start = steady\_clock::now();

fun\_res = KMPSearch(text, text\_len, str, str\_len);

end = steady\_clock::now();

fun\_time = duration\_cast<nanoseconds>(end - start).count();

log << i \* 1000 << " " << fun\_time << " " << str\_time << " " << str << " "

<< (fun\_res == str\_res ? "SUCC" : "FAIL") << endl;

if (str\_res == nullptr) {

fun\_time\_fail += fun\_time;

str\_time\_fail += str\_time;

fail\_times++;

} else {

fun\_time\_succ += fun\_time;

str\_time\_succ += str\_time;

succ\_times++;

}

}

output\_TN << i \* 1000 << "|" << fun\_time\_fail / fail\_times << "|"

<< str\_time\_fail / fail\_times << "|" << fail\_times << endl;

output\_TY << i \* 1000 << "|" << fun\_time\_succ / succ\_times << "|"

<< str\_time\_succ / succ\_times << "|" << succ\_times << endl;

free(str);

free(text);

}

log<<"/\*text tests end\*/"<<endl;

text\_len = 50'000;

for (int i = 5; i <= 100; i += 5) {

str\_len = i;

text = (char \*)calloc(text\_len + 1, sizeof(char));

RandGen(text, text\_len);

str = (char \*)calloc(str\_len + 1, sizeof(char));

fun\_time\_succ = 0;

fun\_time\_fail = 0;

str\_time\_succ = 0;

str\_time\_fail = 0;

fail\_times = 0;

succ\_times = 0;

for (int j = 0; j < repeat; j++) {

fun\_time = 0;

str\_time = 0;

char \*fun\_res;

char \*str\_res;

int a = (rand()%text\_len -str\_len-10 -1);

// cout<<a<<" " << a + str\_len << endl;;

memcpy(str, text+a-1, str\_len);

str[str\_len / 2] = 'F' + (j % 2);

start = steady\_clock::now();

str\_res = strstr(text, str);

end = steady\_clock::now();

str\_time = duration\_cast<nanoseconds>(end - start).count();

start = steady\_clock::now();

fun\_res = KMPSearch(text, text\_len, str, str\_len);

end = steady\_clock::now();

fun\_time = duration\_cast<nanoseconds>(end - start).count();

log << i \* 1000 << " " << fun\_time << " " << str\_time << " " << str << " "

<< (fun\_res == str\_res ? "SUCC" : "FAIL") << endl;

if (str\_res == nullptr) {

fun\_time\_fail += fun\_time;

str\_time\_fail += str\_time;

fail\_times++;

} else {

fun\_time\_succ += fun\_time;

str\_time\_succ += str\_time;

succ\_times++;

}

}

output\_SN << i << "|" << fun\_time\_fail / fail\_times << "|"

<< str\_time\_fail / fail\_times << "|" << fail\_times << endl;

output\_SY << i << "|" << fun\_time\_succ / succ\_times << "|"

<< str\_time\_succ / succ\_times << "|" << succ\_times << endl;

free(str);

free(text);

}

// free(text);

output\_TY.close();

output\_TN.close();

output\_SY.close();

output\_SN.close();

log.close();

return 0;

}