**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №2**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «Списки и их реализация»**

**Вариант 19(1)**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 9302 |  | Плюснина Е.Ю. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

**Постановка задачи**

Реализовать алгоритмы:

1. Двоичный поиск (BinarySearch)
2. Быстрая сортировка (QuickSort)
3. Сортировка вставками (InsertionSort)
4. Глупая сортировка (BogoSort)
5. Сортировка подсчётом (CountingSort) для типа ***char***

# Описание реализуемого алгоритма, оценка временной сложности

1. Двоичный поиск (BinarySearch) – делим зону поиска (размер массива) пополам, находя тем самым средний элемент. Проверяем каждую из двух получившихся зон на наличие искомого элемента, постепенно сужая рамки (begin, end). Если они поменялись местами – элемент невозможно найти. Оценка временной сложности: O(log(n))
2. Быстрая сортировка (QuickSort) – берем средний элемент, и с разных сторон от него сортируем элементы (ближе к началу – меньшие, ближе к концу – большие). Далее алгоритм вызывает сам себя, «разбивая» зону поиска. Оценка временной сложности: О(n2)
3. Сортировка вставками (InsertionSort) – сохраняем второй элемент как искомый. Далее сравниваем с первым элементом и сортируем в зависимости от результата сравнения. Оценка временной сложности: О(n2)
4. Глупая сортировка (BogoSort) – «рандомом» перемешиваем элементы до тех пор, пока они не будут отсортированы. Оценка временной сложности: О(n2!)
5. Сортировка подсчётом (CountingSort) для типа ***char*** – считаем количество повторений, и сортируем в зависимости от индексов повторяющихся или не повторяющихся элементов. Оценка временной сложности: O(max + n)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| QuickSort | | | | | InsertionSort | | | | |
| 10 | 100 | 1000 | 10000 | 100000 | 10 | 100 | 1000 | 10000 | 100000 |
| 0 | 0 | 0 | 2 | 26 | 0 | 0 | 2 | 116 | 8823 |
| 0 | 0 | 0 | 2 | 32 | 0 | 0 | 1 | 125 | 9153 |
| 0 | 0 | 0 | 2 | 26 | 0 | 0 | 1 | 128 | 8709 |
| 0 | 0 | 1 | 1 | 24 | 0 | 0 | 3 | 107 | 8703 |
| 0 | 0 | 1 | 2 | 27 | 0 | 0 | 2 | 108 | 8700 |
| 0 | 0 | 0 | 2 | 25 | 0 | 0 | 1 | 99 | 9706 |
| 0 | 0 | 0 | 1 | 25 | 0 | 0 | 1 | 119 | 8779 |
| 0 | 0 | 0 | 2 | 28 | 0 | 0 | 0 | 124 | 8672 |
| 0 | 0 | 0 | 4 | 25 | 0 | 0 | 1 | 104 | 8627 |
| 0 | 0 | 0 | 3 | 29 | 0 | 0 | 1 | 123 | 8644 |
| 0 | 0 | 0.2 | 2.1 | 26.7 | 0 | 0 | 1.3 | 115.3 | 8851.6 |

# Описание реализованных unit-тестов

TEST\_METHOD(check\_Sorted) – проверка Sorted, неотсортированный массив

TEST\_METHOD(check\_Sorted\_2) – проверка Sorted, отсортированный изначально массив

TEST\_METHOD(check\_BinarySearch\_notfound) – проверка BinarySearch в случае, если элемент не найдется

TEST\_METHOD(check\_BinarySearch\_found) – проверка BinarySearch в случае, если элемент найдется

TEST\_METHOD(check\_QuickSort) – проверка QuickSort на работоспособность

TEST\_METHOD(check\_QuickSort\_2) – проверка QuickSort на правильность

TEST\_METHOD(check\_InsertionSort) – проверка InsertionSort на работоспособность

TEST\_METHOD(check\_InsertionSort\_2) – проверка InsertionSort на правильность

TEST\_METHOD(check\_BogoSort) – проверка BogoSort на работоспособность

TEST\_METHOD(check\_BogoSort\_2) – проверка BogoSort на правильность

TEST\_METHOD(check\_CountingSort) – проверка CountingSort на работоспособность

TEST\_METHOD(check\_CountingSort\_2) – проверка CountingSort на правильность

# Листинг

**Header.h**

#pragma once

bool Sorted(int\* Array, int size);

bool Sorted(char\* Array, int size);

int BinarySearch(int\* Array, int FindElement, int Size);

void QuickSort(int\* Array, int size);

void InsertionSort(int\* Array, int size);

void BogoSort(int\* Array, int size);

void CountingSort(char\* Array, int size);

int main();

**Prog.cpp**

#include <iostream>

#include <ctime>

#include "Header.h"

using namespace std;

bool Sorted(int\* Array, int Size) {

int i = 0;

for (i = 0; i < (Size - 1); i++) {

if (Array[i] <= Array[i + 1]) {

return true;

}

}

return false;

}

bool Sorted(char\* Array, int Size) {

int i = 0;

for (i = 0; i < (Size - 1); i++) {

if (Array[i] <= Array[i + 1]) {

return true;

}

}

return false;

}

int BinarySearch(int\* Array, int FindElement, int Size) {

int begin = 0, medium = Size/2, end = Size - 1;

while (Array[medium] != FindElement) {

medium = begin + end; //find size

if (FindElement > Array[medium]) { //start with the medium element

begin = medium + 1;

}

else if (FindElement < Array[medium]) {

end = medium - 1;

}

else {

return medium;

}

if (end < begin) { //if the "end" has reached the beginning.

throw out\_of\_range("Impossible to find");

}

}

return medium;

}

void QuickSort(int\* Array, int Size) {

int begin = 0, medium = Array[Size / 2], end = Size - 1;

int a = 0;

while (begin <= end) {

while (Array[begin] < medium) {

begin++;

}

while (Array[end] > medium) {

end--;

}

if (begin <= end) {

a = Array[begin];

Array[begin] = Array[end];

Array[end] = a;

begin++;

end--;

}

}

if (begin <= Size - 1) {

QuickSort(&Array[begin], Size - begin);

}

if (end > 0) {

QuickSort(Array, end + 1);

}

}

void InsertionSort(int\* Array, int Size) {

int begin = 0, end = 0, FindElement = 0;

for (begin = 1; begin < Size; begin++) {

FindElement = Array[begin];

end = begin - 1;

while ((end >= 0)&&(Array[end] > FindElement)) {

Array[end + 1] = Array[end];

end = end - 1;

}

Array[end + 1] = FindElement;

}

}

void BogoSort(int\* Array, int Size) {

srand(time(NULL));

int a = 0;

while (!Sorted(Array, Size)) {

for (int begin = 0; begin < Size; begin++) {

a = Array[begin];

Array[begin] = Array[rand() % Size];

Array[rand() % Size] = a;

}

}

}

void CountingSort(char\* Array, int Size) {

int index1 = 0, index2 = 0, max = 0;

for (index1 = 0; index1 < Size; index1++) {

if (max < int(Array[index1])) {

max = int(Array[index1]);

}

}

int\* count = new int[max + 1];

for (index1 = 0; index1 < max + 1; index1++) {

count[index1] = 0;

}

for (index1 = 0; index1 < Size; index1++) {

count[int(Array[index1])]++;

}

while (index1 < max + 1) {

while (count[index1] != 0) {

Array[index2] = char(index1);

count[index1]--;

index2++;

}

index1++;

}

}

int main()

{

int\* array;

int Size = 10;

clock\_t time = 0;

double Array\_time[10];

double max\_time = 0;

for (int i = 0; i < 5; i++)

{

for (int j = 0; j < 10; j++)

{

array = new int[Size];

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

{

array[i] = rand();

}

time = clock();

QuickSort(array, Size - 1);

time = clock() - time;

Array\_time[j] = (double)time / CLOCKS\_PER\_SEC;

cout << Array\_time[j] << endl;

max\_time += Array\_time[j];

delete[] array;

}

cout << "Average time: " << max\_time / 10 << endl;

Size = Size \* 10;

max\_time = 0;

}

Size = 10;

for (int i = 0; i < 5; i++)

{

for (int j = 0; j < 10; j++)

{

array = new int[Size];

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

{

array[i] = rand();

}

time = clock();

InsertionSort(array, Size);

time = clock() - time;

Array\_time[j] = (double)time / CLOCKS\_PER\_SEC;

cout << Array\_time[j] << endl;

max\_time += Array\_time[j];

delete[] array;

}

cout << "Average time: " << max\_time / 10 << endl; Size = Size \* 10; max\_time = 0;

} return 0;

}

**UnitTest.cpp**

#include "CppUnitTest.h"

#include "..\АлгСтр, лабораторная 2, сем 3\Header.h"

#include "..\АлгСтр, лабораторная 2, сем 3\Prog.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1

{

TEST\_CLASS(UnitTest1)

{

public:

int\* Array;

TEST\_METHOD\_INITIALIZE(setUp) {

Array = new int[10];

for (int i = 0; i < 10; i++) {

Array[i] = rand();

}

}

TEST\_METHOD\_CLEANUP(cleanUp) {

delete[] Array;

}

TEST\_METHOD(check\_Sorted) {

Assert::IsFalse(Sorted(Array, 10));

}

TEST\_METHOD(check\_Sorted\_2) {

for (int i = 0; i < 10; i++) {

Array[i] = i;

}

Assert::IsTrue(Sorted(Array, 10));

}

TEST\_METHOD(check\_BinarySearch\_notfound) {

for (int i = 0; i < 10; i++) {

Array[i] = i;

}

try {

BinarySearch(Array, 21, 10);

}

catch (std::out\_of\_range e) {

Assert::AreEqual("Impossible to find", e.what());

}

}

TEST\_METHOD(check\_BinarySearch\_found) {

for (int i = 0; i < 10; i++) {

Array[i] = i;

}

Assert::AreEqual(BinarySearch(Array, 10, 5), 5);

}

TEST\_METHOD(check\_QuickSort) {

for (int i = 9; i >= 0; i--) {

Array[i] = i;

}

QuickSort(Array, 10);

Assert::IsTrue(Sorted(Array, 10));

}

TEST\_METHOD(check\_QuickSort\_2) {

for (int i = 9; i >= 0; i--) {

Array[i] = i;

}

Array[8] = 7;

QuickSort(Array, 10);

Assert::IsTrue(Sorted(Array, 10));

}

TEST\_METHOD(check\_InsertionSort) {

for (int i = 9; i >= 0; i--) {

Array[i] = i;

}

InsertionSort(Array, 10);

Assert::IsTrue(Sorted(Array, 10));

}

TEST\_METHOD(check\_InsertionSort\_2) {

for (int i = 9; i >= 0; i--) {

Array[i] = i;

}

Array[8] = 7;

InsertionSort(Array, 10);

Assert::IsTrue(Sorted(Array, 10));

}

TEST\_METHOD(check\_BogoSort) {

for (int i = 9; i >= 0; i--) {

Array[i] = i;

}

BogoSort(Array, 10);

Assert::IsTrue(Sorted(Array, 10));

}

TEST\_METHOD(check\_BogoSort\_2) {

for (int i = 0; i < 10; i++) {

Array[9 - i] = i;

}

BogoSort(Array, 10);

Assert::AreEqual(Array[0], 0);

}

TEST\_METHOD(check\_CountingSort) {

char array\_char[] = { '3', '0', '1', '2', '3', '4', '1', 'a' };

CountingSort(array\_char, 8);

Assert::IsTrue(Sorted(array\_char, 7));

}

TEST\_METHOD(check\_CountingSort\_2) {

char array\_char[] = { '3', '0', '1', '2', '3', '4', '1', 'a' };

CountingSort(array\_char, 8);

Assert::AreEqual(array\_char[0], '0');

}

};

}

# Вывод

Я научилась использовать алгоритмы сортировки и поиска.