**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

ОТЧЁТ

**по лабораторной работе №2**

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

**Тема: «Программирование двунаправленных связных списков»**

**Вариант 1.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Горбанев А.В |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

## Постановка задачи и описание реализуемых алгоритмов.

Задание: Реализовать алгоритмы сортировки массивов.

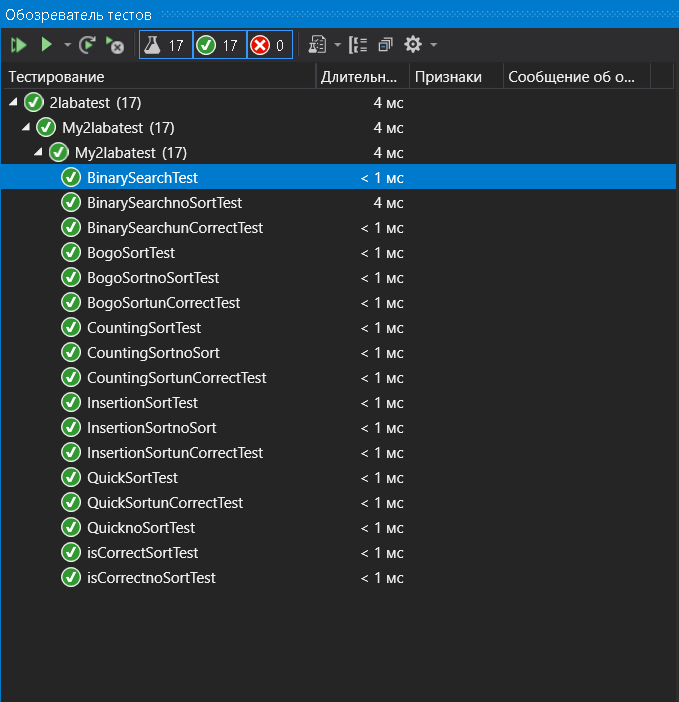
Для выполнения задания используются функции:

|  |  |  |
| --- | --- | --- |
| **Функция** | **Описание** | **Оценка временной сложности** |
| int BinarySearch(int\*, int, int) | Выполняет двоичный поиск | O(log\_2(n)) |
| void QuickSort(int\*, int, int) | Выполняет быструю сортировку | O(n\*log(n)) |
| void BogoSort(int\*, int) | Выполняет глупую сортировку | O(n\*n!) |
| void CountingSort(char\*, int) | Выполняет сортировку подсчетом | O(max+n) |
| Void InsertionSort(int\*a,int size) | Выполняет сортировку вставками | O(n^2) |
| Bool isSorted(int\*a,int size) | Проверяет массив на отсортированность | O(n) |
| Void shuffle(int \*a,int size) | Меняет местами элементы массива рандомно | O(n) |

## Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| Название теста | Что проверяет |
| isCorrectSortTest | Проверка функции isSorted в отсортированном массиве |
| isCorrectnoSortTest | Проверка функции isSorted в неотсортированном массиве |
| BinarySearchTest | Проверка функции BinarySearch в отсортированном массиве |
| BinarySearchunCorrectTest | Проверка функции BinarySearch с некоректными данными |
| BinarySearchnoSortTest | Проверка функции BinarySearch в неотсортированном массиве |
| QuickSortTest | Проверка функции QuickSort в отсортированном массиве |
| QuickSortunCorrectTest | Проверка функции QuickSort с некоректными данными |
| QuicknoSortTest | Проверка функции QuickSort в отсортированном массиве |
| InsertionsSortTest | Проверка функции InsertionSort в неотсортированном массиве |
| InsertionSortunCorrectTest | Проверка функции InsertionSort с некоректными данными |
| InsertionSortnoSort | Проверка функции InsertionSort в отсортированном массиве |
| BogoSortTest | Проверка функции BogoSort в неотсортированном массиве |
| BogoSortunCorrectTest | Проверка функции BogoSort с некорректными данными |
| BogoSortnoSortTest | Проверка функции BogoSort в отсортированном массиве |
| CountingSortTest | Проверка функции CountingSort в неотсортированном массиве |
| CountingSortunCorrectTest | Проверка функции CountingSort с некоректными данными |
| CountingSortnoSort | Проверка функции CountingSort в отсортированном массиве |

## Пример работы unit-тестов.

****

## Cравнение временной сложности алгоритмов

Сравнение временной сложности алгоритмов BubbleSort и QuickSort в виде таблицы.

|  |  |  |
| --- | --- | --- |
| Размерность данных | Время, среднее для 10 запусков | |
| QuickSort | BubbleSort |
| 10 | 0 | 0 |
| 100 | 0 | 0,1 |
| 1000 | 0 | 0.5 |
| 10000 | 1,1 | 59 |
| 100000 | 11.6 | 5788.3 |

Из полученных данных видно, что на больших размерах массива алгоритм быстрой сортировки в разы быстрее сортировки вставками.

## Листинг

|  |
| --- |
| Header.h |
| #pragma once  int Binary\_Search(int\* a, int size, int value);  void QuickSort(int\* a, int size);  void InsertionSort(int\* a, int size);  bool isSorted(int\* a, int size);  void shuffle(int\* a, int size);  void BogoSort(int\* a, int size);  void CountingSort(char\* a, char\* b, int const size); |
| algorithms.cpp |
| #include<iostream>  #include "Header.h"  using namespace std;  int Binary\_Search(int\* a,int size, int value) {  if (size < 0) throw "Uncorrect size";  if (isSorted(a, size) == 0) throw "Array is not Sorted!";  int right = size - 1;  int left = 0;  while (left <= right) {  int mid = (left + right) / 2;  if (a[mid] == value) return mid;  else if (a[mid] > value) right = mid - 1;  else if (a[mid] < value) left = mid + 1;  }  throw "Not Find Value";  }  void QuickSort(int\* a, int size) {  if (size < 0) throw "Uncorrect Size!";  int i = 0;  int j = size - 1;  int mid = a[size / 2];  do {  while (a[i] < mid) {  i++;  }  while (a[j] > mid) {  j--;  }  if (i <= j) {  int tmp = a[i];  a[i] = a[j];  a[j] = tmp;  i++;  j--;  }  } while (i <= j);  if (j > 0) {  QuickSort(a, j + 1);  }  if (i < size) {  QuickSort(&a[i], size - i);  }  }  void InsertionSort(int\* a,int size) {  int temp, itemp;  if (size < 0) throw "Uncorrect Size!";  for (int i = 1; i < size; i++) {  temp = a[i];  itemp = i - 1;  while (itemp >= 0 && a[itemp] > temp) {  a[itemp + 1] = a[itemp];  a[itemp] = temp;  itemp--;  }  }  }  bool isSorted(int\* a, int size) {  while (size > 1) {  if (a[size - 1] < a[size - 2]) return false;  size--;  }  return true;  }  void shuffle(int\* a, int size) {  for (int i = 0; i < size; i++) {  int swap = a[i];  int index = (rand() % size);  a[i] = a[index];  a[index] = swap;  }  }  void BogoSort(int\* a, int size) {  while (!isSorted(a, size)) {  shuffle(a, size);  }  }  void CountingSort(char\* a, char\* b, int const size) {  if (size < 0) throw "Uncorrect Size!";  int c[26];  for (int i = 0; i < 26; i++) {  c[i] = 0;  }  for (int i = 0; i < size; i++) {  c[int(a[i]) - 97]++;  }  int t = 0;  for (int i = 0; i < 26; i++) {  while (c[i] > 0) {  b[t] = char(i + 97);  t++;  c[i]--;  }  }  } |
| 2labatest.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "../2laba/algorithm.cpp"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace My2labatest  {  TEST\_CLASS(My2labatest)  {  public:    TEST\_METHOD(isCorrectSortTest) {  int a[] = { 1,2,3,4,5 };  Assert::IsTrue(isSorted(a, 5) == 1);  }  TEST\_METHOD(isCorrectnoSortTest) {  int a[] = { 1,7,2,1,7 };  Assert::IsTrue(isSorted(a, 5) == 0);  }  TEST\_METHOD(BinarySearchTest)  {  int a[] = { 1,2,3,4,5,6 };  int index;  index = Binary\_Search(a, 6, 4);  Assert::IsTrue(index == 3);  }  TEST\_METHOD(BinarySearchunCorrectTest) {  int a[] = { 1,2,3,4,5,6 };  int index;  try {  index = Binary\_Search(a, -1, 5);  }  catch(const char\* warning){  Assert::AreEqual(warning, "Uncorrect size");  }  }  TEST\_METHOD(BinarySearchnoSortTest) {  int a[] = { 1,6,2,7,2 };  int index;  try {  index = Binary\_Search(a, 5, 6);  }  catch(const char\* warning){  Assert::AreEqual(warning, "Array is not Sorted!");  }  }  TEST\_METHOD(QuickSortTest) {  int a[] = { 1,6,2,7,8,1 };  QuickSort(a, 6);  Assert::IsTrue(isSorted(a, 6) == 1);  }  TEST\_METHOD(QuickSortunCorrectTest) {  int a[] = { 1,2,5,4,7,2 };  try {  QuickSort(a, -1);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "Uncorrect Size!");  }  }  TEST\_METHOD(QuicknoSortTest) {  int a[] = { 1,2,3,4,5 };  QuickSort(a, 5);  Assert::IsTrue(isSorted(a, 5) == 1);  }  TEST\_METHOD(InsertionSortTest) {  int a[] = { 1,6,7,5,4 };  InsertionSort(a, 5);  Assert::IsTrue(isSorted(a, 5) == 1);  }  TEST\_METHOD(InsertionSortunCorrectTest) {  int a[] = { 1,5,2,6,2 };  try {  InsertionSort(a, -1);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "Uncorrect Size!");  }  }  TEST\_METHOD(InsertionSortnoSort) {  int a[] = { 1,2,3,4,5 };  InsertionSort(a, 5);  Assert::IsTrue(isSorted(a, 5) == 1);  }  TEST\_METHOD(BogoSortTest) {  int a[] = { 1,5,2,6,7 };  BogoSort(a, 5);  Assert::IsTrue(isSorted(a, 5) == 1);  }  TEST\_METHOD(BogoSortunCorrectTest) {  int a[] = { 1,6,2,4,6 };  try {  BogoSort(a, -1);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "Uncorrect Size!");  }  }  TEST\_METHOD(BogoSortnoSortTest) {  int a[] = { 1,2,3,4,5 };  BogoSort(a, 5);  Assert::IsTrue(isSorted(a,5) == 1);  }  TEST\_METHOD(CountingSortTest) {  char a[] = { 'b','c','d','a' };  char b[4];  CountingSort(a, b, 4);  Assert::IsTrue((b[0] == 'a') && (b[3] == 'd'));  }  TEST\_METHOD(CountingSortunCorrectTest) {  char a[] = { 'b','c','d','a' };  char b[4];  try {  CountingSort(a, b, -1);  }  catch (const char\* warning) {  Assert::AreEqual(warning, "Uncorrect Size!");  }  }  TEST\_METHOD(CountingSortnoSort) {  char a[] = { 'a','b','c','d' };  char b[4];  CountingSort(a, b, 4);  Assert::IsTrue((b[0] == 'a') && (b[1] == 'b') && (b[2] == 'c') && (b[3] == 'd'));  }  };  } |
| Source.cpp |
| #include<iostream>  #include "Header.h"  #include <ctime>  #define ArraySize 100000  using namespace std;  void elapsedTime(void (\*funcName)(int\*, int), int\* arr, int array\_size)  {  int i, j;  int t1, t2;  double seconds, res = 0;  srand(time(0));  for (i = 0; i < 10; i++)  {  for (j = 0; j < array\_size; j++)  {  arr[j] = rand() % ArraySize;  }  t1 = clock();  funcName(arr, array\_size);  t2 = clock() - t1;  seconds = double(t2);  res += seconds \* 1000 / CLOCKS\_PER\_SEC;  cout << seconds \* 1000 / CLOCKS\_PER\_SEC << " ";  }  cout << endl;  res /= 10;  cout << res << endl;  }  int main()  {  int mass[ArraySize];  void (\*kind[2])(int\*, int);  kind[0] = QuickSort;  kind[1] = InsertionSort;  elapsedTime(kind[0], mass, 10);  elapsedTime(kind[0], mass, 100);  elapsedTime(kind[0], mass, 1000);  elapsedTime(kind[0], mass, 10000);  elapsedTime(kind[0], mass, 100000);  elapsedTime(kind[1], mass, 10);  elapsedTime(kind[1], mass, 100);  elapsedTime(kind[1], mass, 1000);  elapsedTime(kind[1], mass, 10000);  elapsedTime(kind[1], mass, 100000);  return 0;  } |