**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 2.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9302 |  | Давтян С.Д. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2020

## Постановка задачи и описание реализуемых алгоритмов.

Задание: Реализовать алгоритмы сортировки массивов.

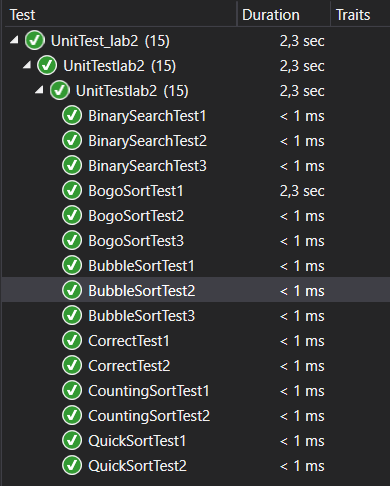
Для выполнения задания используются функции:

|  |  |  |
| --- | --- | --- |
| **Функция** | **Описание** | **Оценка временной сложности** |
| void swap(int& a, int& b) | Меняет местами значения | O(1) |
| void PrintArray(int\*, s) | Вывод массива типа int | O(n) |
| void PrintArray(char\*, int) | Вывод массива типа char | O(n) |
| bool correct(int\*, int) | Проверяет отсортированность списка для алгоритма Bogosort | O(n) |
| int BinarySearch(int\*, int, int) | Выполняет двоичный поиск | O(log\_2(n)) |
| void BubbleSort(int\*, int) | Выполняет сортировку пузырьком | O(n^2) |
| 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) |

## Описание реализованных unit-тестов.

|  |  |
| --- | --- |
| Название теста | Что проверяет |
| CorrectTest1 | Проверка функции correct в отсортированном массиве |
| CorrectTest2 | Проверка функции correct в неотсортированом массиве |
| BinarySearchTest1 | Проверка функции BinarySearch в отсоритрованном массиве |
| BinarySearchTest2 | Проверка функции BinarySearch в неотсоритрованном массиве |
| BinarySearchTest3 | Проверка функции BinarySearch с некорректными данными |
| BubbleSortTest1 | Проверка функции BubbleSort в неотсортированном массиве |
| BubbleSortTest2 | Проверка функции BubbleSort в отсортированном массиве |
| BubbleSortTest3 | Проверка функции BubbleSort с некорректными данными |
| QuickSortTest1 | Проверка функции QuickSort в отсортированном массиве |
| QuickSortTest2 | Проверка функции QuickSort в неотсортированном массиве |
| BogoSortTest1 | Проверка функции BogoSort в неотсортированном массиве |
| BogoSortTest2 | Проверка функции BogoSort в отсортированном массиве |
| BogoSortTest3 | Проверка функции BogoSort с некорректными данными |
| CountingSortTest1 | Проверка функции CountingSort в неотсортированном массиве |
| CountingSortTest2 | Проверка функции CountingSort с некорректными данными |

## 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 | 0,2 | 5,9 |
| 10000 | 1,2 | 637,4 |
| 100000 | 19,9 | 69260.7 |

Данные приведены в миллисекундах. С помощью таблицы можно отметить, что функция QuickSort работает намного быстрее. Подробнее о результатах замеров можно посмотреть в файле “Time\_test\_result.txt”.

## Листинг

|  |
| --- |
| algorithms.h |
| #pragma once  void swap(int& a, int& b); // O(1)  void PrintArray(int\* A, int size); // O(n)  void PrintArray(char\* A, int size); // O(n)  bool correct(int\* A, int size); // O(n)  int BinarySearch(int\* A, int size, int search\_num); // O(log\_2(n))  void BubbleSort(int\* A, int size); // O(n^2)  void QuickSort(int\* A, int low, int high); // O(n\*log(n))  void BogoSort(int\* A, int size); // O(n\*n!)  void CountingSort(char\* A, int size); // O(max+n) |
| algorithms.cpp |
| #include "algorithms.h"  #include <iostream>  void swap(int& a, int& b)  {  int temp = a;  a = b;  b = temp;  }  //the function checks if the array is sorted  bool correct(int\* A, int size)  {  if (size <= 0)  throw "Error! Incorrect data";  while (--size > 0)  if (A[size - 1] > A[size])  return false;  return true;  }  //the function performs a binary search  int BinarySearch(int\* A, int size, int search\_num)  {  if (size <= 0)  throw "Error! Incorrect data";  if (!(correct(A, size)))  throw "Error! Array is not sorted.";  int middle, left, right;  left = 0; right = size - 1;  while (true)  {  if (left > right) return (-1); // value not found  middle = left + (right - left) / 2;  if (A[middle] < search\_num) left = middle + 1;  if (A[middle] > search\_num) right = middle - 1;  if (A[middle] == search\_num) return middle;  }  }    //the function outputs an array  void PrintArray(int\* A, int size)  {  if (size <= 0)  throw "Error! Incorrect data";  for (int i = 0; i < size; i++)  {  std::cout << A[i] << ' ';  }  std::cout << std::endl;  }  void PrintArray(char\* A, int size)  {  if (size <= 0)  throw "Error! Incorrect data";  for (int i = 0; i < size; i++)  {  std::cout << A[i] << ' ';  }  std::cout << std::endl;  }  //the function performs a bubble sort  void BubbleSort(int\* A, int size)  {  if (size <= 0)  throw "Error! Incorrect data";  bool control = true;  int temp;  for (int i = 0; i < size - 1; i++)  {  for (int j = 0; j < size - i - 1; j++)  {  if (A[j] > A[j + 1])  {  if (control == true)  control = false;  swap(A[j], A[j + 1]);  }  }  if (control == true) break;  }  }  //the function performs a quick sort  // low - min index in the array  // high - max index in the array  void QuickSort(int\* A, int low, int high)  {  int i = low;  int j = high;  int pivot = A[high];    while (i <= j)  {  while (A[i] < pivot)  i++;  while (A[j] > pivot)  j--;  if (i <= j)  {  swap(A[i], A[j]);  i++;  j--;  }  }  if (j > low)  QuickSort(A, low, j);  if (i < high)  QuickSort(A, i, high);  }  //the function performs a bogo sort  void BogoSort(int\* A, int size)  {  if (size <= 0)  throw "Error! Incorrect data";  while (!correct(A, size))  {  for (int i = 0; i < size; ++i)  swap(A[i], A[(rand() % size)]);  }  }  //the function performs a counting sort  void CountingSort(char\* A, int size)  {  if (size <= 0)  throw "Error! Incorrect data";  char max=char(0);  for (int i = 0; i < size; i++)  {  if (A[i] > max)  {  max = A[i];  }  }  int count\_size = int(max+1);  int\* count = new int[count\_size];  std::memset(count, 0, count\_size \* sizeof(\*count));    for (int i = 0; i < size; i++)  {  if (A[i] != '\0')  {  count[int(A[i])]++;  }  }  int i = 0, control = 0;  for (int j = 0; j < count\_size; j++)  {  if (count[j] != 0)  {  while (i < count[j]+control)  {  A[i] = char(j);  i++;  }  control = i;  }  }  delete[]count;  } |
| UnitTest\_lab2.cpp |
| #include "pch.h"  #include "CppUnitTest.h"  #include "../lab2/algorithms.h"  using namespace Microsoft::VisualStudio::CppUnitTestFramework;  namespace UnitTestlab2  {  TEST\_CLASS(UnitTestlab2)  {  public:  int A[10] = { 2,0,-1,10,-7,21,11,-3,24,-34 };  char B[10] = { 'f','s','d','s','a','d','u','z','k','l' };  int C[10] = { -3,-2,-1,0,1,2,3,4,5,6 };  TEST\_METHOD(CorrectTest1)  {  Assert::IsTrue(correct(C,10));  }  TEST\_METHOD(CorrectTest2)  {  int control[] = { 1,2,-3 };  Assert::IsFalse(correct(control, 3));  }  TEST\_METHOD(BinarySearchTest1)  {  Assert::IsTrue(BinarySearch(C, 10, 3) == 6);  }  TEST\_METHOD(BinarySearchTest2)  {  try  {  int control = BinarySearch(A, 10, 3);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Array is not sorted.");  }  }  TEST\_METHOD(BinarySearchTest3)  {  try  {  int control = BinarySearch(A, -10, 3);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Incorrect data");  }  }  TEST\_METHOD(BubbleSortTest1)  {  BubbleSort(A, 10);  Assert::IsTrue(A[0] == -34, L"Assert 1");  Assert::IsTrue(A[9] == 24, L"Assert 2");  }  TEST\_METHOD(BubbleSortTest2)  {  BubbleSort(C, 10);  Assert::IsTrue(C[0] == -3, L"Assert 1");  Assert::IsTrue(C[9] == 6, L"Assert 2");  }  TEST\_METHOD(BubbleSortTest3)  {    try  {  BubbleSort(A, -10);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Incorrect data");  }  }  TEST\_METHOD(QuickSortTest1)  {  QuickSort(A, 0, 9);  Assert::IsTrue(A[0] == -34, L"Assert 1");  Assert::IsTrue(A[9] == 24, L"Assert 2");  }  TEST\_METHOD(QuickSortTest2)  {  QuickSort(C, 0, 9);  Assert::IsTrue(C[0] == -3, L"Assert 1");  Assert::IsTrue(C[9] == 6, L"Assert 2");  }  TEST\_METHOD(BogoSortTest1)  {  BogoSort(A, 10);  Assert::IsTrue(A[0] == -34, L"Assert 1");  Assert::IsTrue(A[9] == 24, L"Assert 2");  }  TEST\_METHOD(BogoSortTest2)  {  BogoSort(C, 10);  Assert::IsTrue(C[0] == -3, L"Assert 1");  Assert::IsTrue(C[9] == 6, L"Assert 2");  }  TEST\_METHOD(BogoSortTest3)  {  try  {  BogoSort(A, -10);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Incorrect data");  }  }  TEST\_METHOD(CountingSortTest1)  {  CountingSort(B, 10);  Assert::IsTrue(B[0] == 'a', L"Assert 1");  Assert::IsTrue(B[9] == 'z', L"Assert 2");  }  TEST\_METHOD(CountingSortTest2)  {  try  {  CountingSort(B, -10);  }  catch (const char\* error)  {  Assert::AreEqual(error, "Error! Incorrect data");  }  }  };  } |
| Source.cpp |
| #include <iostream>  #include "algorithms.h"  #include <time.h>  #include <fstream>  using namespace std;  int main()  {  srand(time(0));  int size = 10, minus;  double average;  int\* array\_for\_tests;  clock\_t start\_time, end\_time, result\_time;  ofstream of("Time\_test\_result.txt");  double result[10];  of << "Result for QuickSort: " << endl;    for (int i = 0; i < 5; i++)  {  for (int j = 0; j < 10; j++)  {  array\_for\_tests = new int[size];  for (int k = 0; k < size; k++)  {  minus = -1;  if (rand() % 10 > 4)  minus = 1;  array\_for\_tests[k] = rand() \* minus;  }  start\_time = clock();  QuickSort(array\_for\_tests, 0, size - 1);  end\_time = clock();  result\_time = end\_time - start\_time;  result[j] = ((double)result\_time \* 1000 / CLOCKS\_PER\_SEC);  of << result[j] << " ";  delete[] array\_for\_tests;  }  average = 0;  for (int j = 0; j < 10; j++)  average += result[j];  average /= 10;  of << "for " << size << " elements. " << " Average: " << average << '\n';  size \*= 10;  }  size = 10;  of << "Result for BubbleSort: " << endl;  for (int i = 0; i < 5; i++)  {  for (int j = 0; j < 10; j++)  {  array\_for\_tests = new int[size];  for (int k = 0; k < size; k++)  {  array\_for\_tests[k] = rand();  }  start\_time = clock();  BubbleSort(array\_for\_tests, size);  end\_time = clock();  result\_time = end\_time - start\_time;  result[j] = ((double)result\_time \* 1000 / CLOCKS\_PER\_SEC);  of << result[j] << " ";  delete[] array\_for\_tests;  }  average = 0;  for (int j = 0; j < 10; j++)  average += result[j];  average /= 10;  of << "for " << size << " elements. " << " Average: " << average << '\n';  size \*= 10;  }  of.close();  } |