**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: «Алгоритмы сортировки и поиска»**

**Вариант 1**

|  |  |  |
| --- | --- | --- |
| Студент гр. 0302 |  | Чёрный Я.И. |
| Преподаватель |  | Тутуева А.В. |

Санкт-Петербург

2021

**Оглавление**

[1 Постановка задачи 3](#_Toc28306331)

[1.1 Реализуемые классы и методы: 3](#_Toc28306332)

[1.1.1 Sort: 3](#_Toc28306333)

[2 Unit-тесты 4](#_Toc28306334)

[3 Сравнение временной сложности алгоритмов 4](#_Toc28306335)

[4 Пример работы 5](#_Toc28306336)

[4.1 Функция main, представляющая пример использования написанной структуры данных: 5](#_Toc28306337)

[4.2 Результат выполнения: 5](#_Toc28306338)

[5 Текст программы: 6](#_Toc28306339)

[6 Вывод 8](#_Toc28306340)

# Постановка задачи

Целью данной работы является реализация большинства распространённых алгоритмов сортировки (таких, как QuickSort, InsertionSort, BubbleSort и пр.) и сравнение их быстродействия.

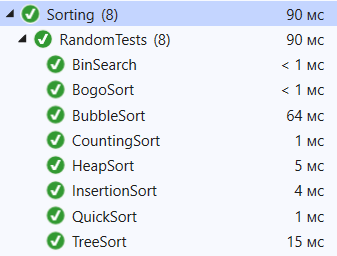
## Реализуемые классы и методы:

### Sort:

|  |  |  |
| --- | --- | --- |
| Функция | Назначение | Временная сложность |
| static size\_t BinarySearch(int\* data, int dataSize , int match) | Выполняет бинарный поиск по массиву | O(log2(n)) |
| static void QuickSort(int\* data, int first, int last) | Выполняет QuickSort для заданного массива | O(*n*log*n*) |
| static void InsertionSort(int\* data, size\_t dataSize) | Выполняет InsertionSort для заданного массива | О(*n*2) |
| static void BubbleSort(int\* data, size\_t dataSize) | Выполняет BubbleSort для заданного массива | О(*n*2) |
| static void TreeSort(int\* data, size\_t dataSize) | Выполняет TreeSort для заданного массива | O(n log n) |
| static void HeapSort(int\* data, size\_t dataSize) | Выполняет HeapSort для заданного массива | O(nlog n) |
| static void BogoSort(int\* data, size\_t dataSize) | Выполняет BogoSort для заданного массива | O((n+1)!) |
| static void CountingSort(char\* data, size\_t dataSize) | Выполняет CountingSort для заданного массива | O(n+k) , k – range of values |
| static bool isSorted(int\* data, size\_t dataSize) | Проверяет, отсортирован ли массив | O(n) |
| static void shuffle(int\* data, size\_t dataSize) | Случайным образом меняет местами элементы массива | O(n) |
| static void swap(int\* a, int\* b) | Меняет местами элементы массива | O(1) |
| static void heapify(int\* data, int n, int i) | Преобразование в кучу | O(log n) |

# Unit-тесты

Для проверки работоспособности программы были реализованы юнит-тесты. Каждый из данных тестов запускает N прогонов с массивами случайных чисел/символов.

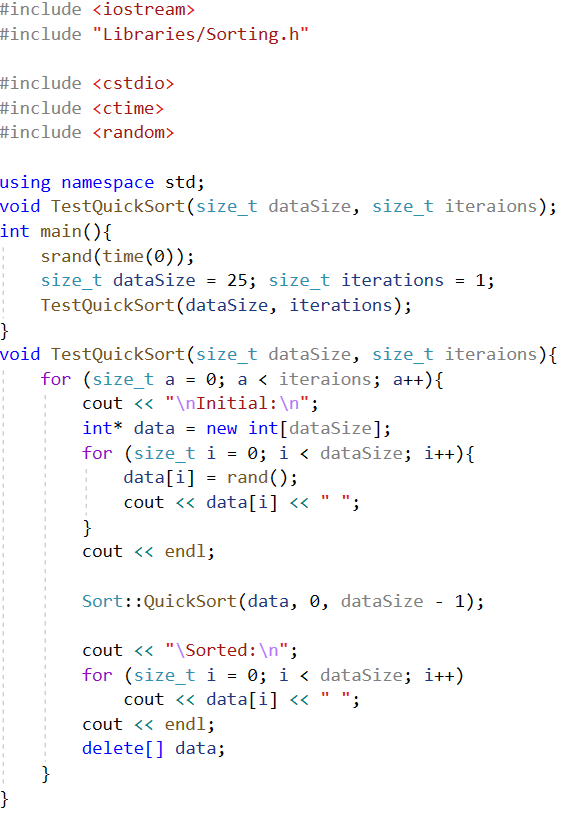


# Сравнение временной сложности алгоритмов

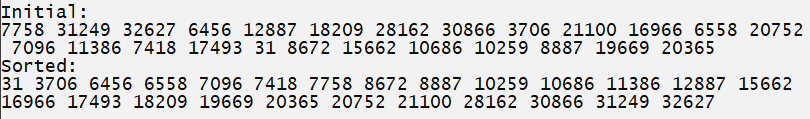
В качестве сравниваемых алгоритмов возьмём QuickSort и InsertionSort. Полученные временные результаты являются средними для десяти запусков алгоритма на заданном массиве данных.

# Пример работы

## Функция main, представляющая пример использования написанной структуры данных:



## Результат выполнения:



# Текст программы:

|  |
| --- |
| #pragma once  #include "BinTree.h";  class Sort  {  public:  static size\_t BinarySearch(int\* data, int dataSize , int match){  size\_t start = 0, end = dataSize - 1;  while (start <= end){  size\_t middleElement = start + (end - start) / 2;  if (data[middleElement] == match)  return middleElement;  if (data[middleElement] < match)  start = middleElement + 1; // shift start to right  else  end = middleElement - 1; // shift end to left  }  return -1;  }  static void QuickSort(int\* data, int first, int last){  if (first < last){  int left = first, right = last, middle = data[(left + right) / 2];  do{  while (data[left] < middle) left++;  while (data[right] > middle) right--;  if (left <= right){  swap(&data[left], &data[right]);  left++;  right--;  }  } while (left <= right);  QuickSort(data, first, right);  QuickSort(data, left, last);  }  }  static void InsertionSort(int\* data, size\_t dataSize){  for (int i = 1; i < dataSize; i++) {  int key = data[i];  int j = i - 1;  while (j >= 0 && data[j] > key) {  data[j + 1] = data[j];  j--;  }  data[j + 1] = key;  }  }  static void BubbleSort(int\* data, size\_t dataSize){  for (size\_t i = 0; i < dataSize - 1; i++){  for (size\_t j = 0; j < dataSize - i - 1; j++){  if (data[j + 1] < data[j])  swap(&data[j], &data[j+1]);  }  }  }  static void TreeSort(int\* data, size\_t dataSize){  Tree tree;  for (size\_t i = 0; i < dataSize; i++)  tree.Insert(data[i]);  Iterator\* dft = tree.CreateDFTIterator();  size\_t i = 0;  while (dft->HasNext()){  data[i] = dft->Next();  i++;  }    }  static void HeapSort(int\* data, size\_t dataSize){  for (int i = dataSize / 2 - 1; i >= 0; i--)  heapify(data, dataSize, i);  // extract  for (int i = dataSize - 1; i >= 0; i--){  // move current root to end  swap(&data[0], &data[i]);  // call max heapify on the heap  heapify(data, i, 0);  }  }  static void BogoSort(int\* data, size\_t dataSize){  while (!isSorted(data, dataSize))  shuffle(data, dataSize);  }  static void CountingSort(char\* data, size\_t dataSize){  char\* output = new char[dataSize];  int count[256];  for (int i = 0; i < 256; i++)  count[i] = 0;  for (int i = 0; i < dataSize; i++)  count[data[i]]++;  for (int i = 1; i < 256; i++)  count[i] += count[i - 1];  for (int i = dataSize-1; i >= 0; i--){  output[count[data[i]] - 1] = data[i];  count[data[i]]--;  }  for (int i = 0; i < dataSize; i++)  data[i] = output[i];  delete[] output;  }  static bool isSorted(int\* data, size\_t dataSize){  while (dataSize > 1){  dataSize--;  if (data[dataSize] < data[dataSize - 1])  return false;  }  return true;  }  static bool isSorted(char\* data, size\_t dataSize){  while (dataSize > 1){  dataSize--;  if (data[dataSize] < data[dataSize - 1])  return false;  }  return true;  }  private:  //helpers  static void shuffle(int\* data, size\_t dataSize){  for (size\_t i = 0; i < dataSize; i++)  swap(&data[i], &data[rand() % dataSize]);  }  static void swap(int\* a, int\* b){  int temp = \*a;  \*a = \*b;  \*b = temp;  }  static void heapify(int\* data, int n, int i){  // largest is root  int largest = i;  int l = 2 \* i + 1; // left = 2\*i + 1  int r = 2 \* i + 2; // right = 2\*i + 2  // left > root  if (l < n && data[l] > data[largest])  largest = l;  // right > largest  if (r < n && data[r] > data[largest])  largest = r;  if (largest != i){  swap(&data[i], &data[largest]);  //continue  heapify(data, n, largest);  }  }  }; |

# Вывод

В результате выполнения данной работы были рассмотрены и реализованы различные алгоритмы сортировки данных, был проведён сравнительный анализ временной сложности QuickSort и InsertionSort, и дана оценка временной сложности для каждого реализуемого алгоритма.