Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Языки программирования (ЯП)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе

на тему:

«АНАЛИЗ И СРАВНЕНИЕ АЛГОРИТМОВ СОРТИРОВКИ»

БГУИР КП 1-40 01 01  ПЗ

Студент: Киселев Н.И.

Руководитель: Шостак Е.В.

Минск 2019

Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

УТВЕРЖДАЮ

Заведующий кафедрой ПОИТ

––––––––––––––––––––––––

(подпись)

Лапицкая Н.В. 2019 г.

ЗАДАНИЕ

по курсовому проектированию

Студенту Киселеву Николаю Игоревичу

1. Тема работы Анализ и сравнение алгоритмов сортировки

2. Срок сдачи студентом законченной работы 23.12.2019

3. Исходные данные к работе Провести анализ и сравнение различных алгоритмов сортировки, разработатть программное средство для их реализации и тестирования.

4. Содержание расчётно-пояснительной записки (перечень вопросов, которые подлежат разработке)

*Введение.*

*1. Аналитический обзор литературы и существующих аналогов;*

*2. Разработка алгоритма;*

*3. Разработка программного средства;*

*4. Тестирование, экспериментальные исследования и анализ полученных результатов;*

*5. Руководство пользователя программы;*

*Заключение, список литературы, ведомость, приложения.*

5. Перечень графического материала (с точным обозначением обязательных чертежей и графиков)

1. Схема программы

6. Консультант по курсовому проекту Шостак Е.В.

7. Дата выдачи задания 01.09.2019 г.––   –

8. Календарный график работы над проектом на весь период проектирования (с обозначением сроков выполнения и процентом от общего объёма работы):

раздел 1, введение к 15.09.2019 – 10 % готовности работы;

разделы 2 к 15.10.2019 – 30 % готовности работы;

разделы 3,4 к 15.11.2019 – 60 % готовности работы;

раздел 5, 6 к 05.12.2019 – 90 % готовности работы;

оформление пояснительной записки и графического материала к 23.12.2019– 100 % готовности работы.

Защита курсового проекта с 23.12.2019 по 12.01.2020 г.

РУКОВОДИТЕЛЬ Шостак Е.В.

(подпись)

Задание принял к исполнению –––\_\_\_\_––     01.09.2019 г.

(дата и подпись студента)

**СОДЕРЖАНИЕ**

Введение 5

1 Анализ алгоритмов сортировки 6

1.1 Задача сортировки 6

1.2 Классификация алгоритмов сортировки 6

1.3 Оценка алгоритма сортировки 7

2 Основные алгоритмы сортировки и их реализация …........................................8

2.1 Обменные 8

2.1.1 Сортировка пузырьком 8

2.1.2 Сортировка перемешиванием 9

2.1.3 Сортировка расчёской 10

2.1.4 Быстрая сортировка, сортировка Хоара 11

2.2 Выбором 12

2.2.1 Сортировка выбором 12

2.2.2 Пирамидальная сортировка 13

2.3 Вставками 15

2.3.1 Сортировка вставками 15

2.3.2 Сортировка Шелла 16

2.3.3 Сортировка деревом 18

2.4 Сортировка слиянием 19

3 Разработка программного средства для исследования алгоритмов сортировки

3.1 Требования к программному средству 21

3.2 Архитектура классов сортировки 21

3.3 Пользовательский интерфейс программы 22

4 Тестирование и анализ полученных результатов 24

5 Руководство пользователя программы 27

Заключение 28

Список использованных источников 29

Приложение А 30

Приложение Б 38

# ВВЕДЕНИЕ

Сортировка является одной из фундаментальных и часто встречающихся на практике алгоритмических задач в программировании. Весьма часто сортировка данных перед обработкой позволяет значительно сократить сложность и время работы многих алгоритмов. Поэтому, возможно, никакая другая задача не породила такого количества разнообразнейших решений. Алгоритмы сортировки всегда были тесно связаны с развитием вычислительной техники, являясь одной из основных задач ее работы. Поэтому к настоящему времени теория сортировки достаточно развита, содержит описание и анализ математических свойств достаточно большого числа алгоритмов.

В рамках данной работы будут рассмотрены наиболее известные алгоритмы сортировки, их свойства, достоинства и недостатки. Также будет спроектировано и реализовано программное средство для сортировки числовых массивов данными методами с оценкой времени выполнения и визуализацией шагов процесса. С его помощью будет проведено тестирование и сравнение алгоритмов для различных видов и размеров входной последовательности.

Разработка будет вестись на платформе .NET на языке C++/CLI с использованием принципов ООП и SOLID. Управляемый код и другие возможности .NET будут использоваться в модуле, ответственном за пользовательский интерфейс, созданный с помощью библиотеки WinForms. В остальных модулях планируется использовать неуправлямый код на чистом С++ со стандартной библиотекой языка.

В первом разделе будет поставлена задача сортировки, определены основные свойства и параметры оценки алгоритмов сортировки. Во втором разделе будут описаны наиболее популярные методы с приведением псевдокода, основных свойств, преимуществ и недостатков. Третий раздел будет посвящен проектированию и реализации программного средства с использованием вышеприведенных алгоритмов. В четвертом разделе будут приведены результаты тестирования и сравнения реализованных методов. Пятый раздел будет содержать краткое руководство пользователя программы.

## 1 АНАЛИЗ АЛГОРИТМОВ СОРТИРОВКИ

### Задача сортировки

Сортировку можно определить как преобразование входной последовательности элементов в неубывающую последовательность. Последовательность элементов {Ai} называют неубывающей, если для любых i и j, таких что i < j выполняется неравенство Ai <= Aj. Обычно на вход подается последовательность записей, каждый элемент которой представляет из себя какую-либо информацию и ключ, по которому выполняется сортировка. В качестве ключа часто выступает число, а поле данных никак не влияет на работу алгоритма.

### Классификация алгоритмов сортировки

Основные алгоритмы сортировки можно классифицировать на наличию или отсутствию следующих свойств: устойчивость, естественность поведения, сфера применения, потребность в дополнительной памяти, вычислительная сложность и необходимость в знании предварительной информации о сортируемых данных.

Устойчивость — свойство алгоритма сортировки не менять взаимного расположения элементов с одинаковыми ключами. Неустойчивые методы таким образом переупорядочивают массив, что равные элементы после сортировки могут идти в порядке, отличающемся от исходного.

Естественность поведения — свойство, показывающее насколько эффективнее алгоритм обрабатывает упорядоченные или частично упорядоченные массивы по сравнению с неупорядоченными. Неестественный метод может работать так же долго на упорядоченных данных или даже хуже.

По сфере применения алгоритмы сортировки можно разделить на внутренние и внешние. Внутренняя сортировка работает с данными в оперативной памяти с произвольным доступ к любой ячейке. Обычно данные алгоритмы не требуют дополнительных затрат памяти. Внешняя сортировка работает с внешними устройствами памяти, характеризующимися обычно большим объемом, последовательным доступом и небольшой скоростью работы по сравнению с оперативной памятью.

* 1. Оценка алгоритма сортировки

Основными параметрами оценки алгоритмов сортировки являются вычислительная сложность и дополнительная память.

Вычислительная сложность характеризует быстродействие алгоритма. В теории алгоритмов она определяется как количество элементарных операций, затрачиваемых алгоритмом для решения конкретной задачи и задается функцией, которая каждой входной длине n ставит в соответствие минимальное время, затрачиваемое алгоритмом на решение всех задач этой длины. Эта функция часто записывается в O-нотации. Из математического анализа известно, что функция f(n) есть О(g(n)), если существует константа С такая, что для всех .

Скорость сортировки определяется количеством сравнений и количеством обменов (обмены требуют больше времени). Для полной оценки важно определять поведение алгоритма в худшем, среднем и наилучшем случае. Идеальным поведением является O(n), хорошим можно считать O(n log n), примером плохого поведения будет O(n2). В теории алгоритмов для сортировки доказывается следующая теорема: Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений C ≥ O(n⋅log2(n)).

При оценки производительности метода сортировки важно учитывать, что в современных архитектурах компьютеров широко применяется подкачка и кэширование памяти. Не все алгоритмы хорошо сочетаются с данными механизмами.

Некоторые алгоритмы требуют выделения дополнительной памяти, помимо той, что выделена на хранение исходного массива. Для своего выполнения они обычно требуют O(log n) или O(n) памяти. Алгоритмы, которые не требуют дополнительной памяти называют алгоритмами сортировки на месте.

**2 ОСНОВНЫЕ АЛГОРИТМЫ СОРТИРОВКИ И ИХ РЕАЛИЗАЦИЯ**

**2.1** Обменные

**2.1.1** Сортировка пузырьком

Сортировка пузырьком является одним из самых простейших алгоритмов сортировки. Основная идея состоит в последовательных проходах по массиву с попарным сравнением и перестановкой соседних элементов. Название происходит от того факта, что за каждую итерацию наибольший элемент массива «всплывает» в конец списка. Несмотря на свою простоту, алгоритм используется практически только в учебных целях. Тем не менее на его основе создан ряд усовершенствований, активно использующихся на практике.

**Алгоритм:**

1. На каждой итерации производится полный проход по массиву с последовательным сравнением соседних элементов. Если порядок в паре неверный, то элементы меняют местами.
2. Если за проход не произошло не произошло ни одного обмена, то массив уже отсортирован, иначе перейти на следующую итерацию.

Так как за каждый проход по массиву хотя бы один элемент перемещается на свое место, то достаточно не более n−1 проходов для сортировки массива. Если учесть, что после i-ой итерации внешнего цикла i последних элементов уже находятся на своих местах , то достаточно n−i−1 проходов.

**Псевдокод**:

function bubbleSort(array a):

i = 0

swapped = true

while swapped

swapped = false

for j = 0 to n - i - 2

if a[j] > a[j + 1]

swap(a[j], a[j + 1])

swapped = true

i = i + 1

**Свойства:**

* Устойчивый
* Естественный
* В среднем и худшем случае вычислительная сложность O(n2), в наилучшем случае (упорядоченный массив) O(n).
* Требует дополнительно O(1) памяти для выполнения обмена

**Преимущества:**

* Простая реализация, в связи с чем часто используется в учебных целях.
* Так как при малых n алгоритм не сильно уступает в производительности другим, то иногда допустимо его использование в малозагруженных системах.
* Алгоритм лежит в основе других более эффективных методов сортировки.

**Недостатки:**

* Затраты быстро растут с ростом размера массива, причем сортировка пузырьком уступает в эффективности многим другим алгоритмам со сложностью O(n2), таким как сортировка вставками.
* На современных процессорах алгоритм вызывает как минимум в два раза болье кэш-промахов и операций записи, чем сортировка вставками, а также частые неверные предсказания переходов.

**2.1.2** Сортировка перемешиванием

Сортировка перемешиванием (шейкерная сортировка) — усовершенствование сортировки пузырьком. Идея состоит в том, чтобы на каждой итерации сортировать массив передвигаясь сначала в прямом, затем в обратном направлении.

**Псевдокод**:

function shakerSort(array a):

begin = -1

end = n - 2

while swapped

swapped = false

begin++

for i = begin to end

if a[i] > a[i + 1]

swap(a[i], a[i + 1])

swapped = true

if !swapped

break

swapped = false

end--

for i = end downto begin

if a[i] > a[i + 1]

swap(a[i], a[i + 1])

swapped = true

**Свойства:**

* Устойчивый
* Естественный
* В среднем, работает в два раза быстрее сортировки пузырьком.
* В среднем и худшем случае (отсортированный в обратном порядке массив) вычислительная сложность O(n2), в наилучшем случае (упорядоченный массив) O(n).
* Требует дополнительно O(1) памяти для выполнения обмена

**2.1.3** Сортировка расчёской

Сортировка расческой — усовершенствование сортировки пузырьком, основная идея которой в сравнении элементов массива на различных расстояниях, а не только соседних.

**Алгоритм:**

1. Задать начальное значение расстояния между сравниваемыми элементами делением размера массива на число k, называемое фактором уменьшения (часто берут экспериментально определенное k=1.3)
2. Провести проход пузырьковой сортировки сравнивая элементы на текущем расстоянии.
3. Уменьшить текущее расстояние, поделив его на фактор уменьшения.
4. Если расстояние больше единицы, то перейти на шаг 2, иначе досортировать массив обычным пузырьком.

**Псевдокод**:

function combSort(array a):

k = 1.3

jump = n

bool swapped = true

while jump > 1 or swapped

if jump > 1

jump /= k

swapped = false

for i = 0 to size - jump - 1

if a[i + jump] < a[i]

swap(a[i], a[i + jump])

swapped = true

**Свойства:**

* Неустойчивый
* Естественный
* В среднем и худшем случае (отсортированный в обратном порядке массив) вычислительная сложность O(n2), в наилучшем случае (упорядоченный массив) O(n).
* Требует дополнительно O(1) памяти для выполнения обмена.
* Самый быстрый алгоритм из класса O(n2), асимптотически стремится к O(nlogn).
  + 1. Быстрая сортировка, сортировка Хоара

Быстрая сортировка – один из самых применяемым и эффективных алгоритмов сортировки. Был разработан в 1960 году английским информатиком Чарльзом Хоаром. Быстрая сортировка является модификацией алгоритмов прямого обмена, таких как пузырьковая и шейкерная сортировка с использованием принципа "разделяй и властвуй". Изначально обмены происходят на наибольшем возможном расстоянии, затем массив делится пополам и процедура сортировки рекурсивно вызывается для каждой части.

**Алгоритм:**

1. Выбрать опорный элемент p - середину массива и разделить массив по этому элементу (разбиение Хоара)
   1. Берутся два индекса, один из которых указывает на начало массива, другой – на конец.
   2. Индексы последовательно сближаются, пока не найдётся пара элементов, где один больше опорного и расположен перед ним, а второй меньше и расположен после.
   3. Поменять найденные элементы местами.
   4. Если индексы не пересеклись, то перейти на шаг 1.2.
2. Если левый подмассив от p содержит более одного элемента, рекурсивно вызвать процедуру для него.
3. Если правый подмассив от p содержит более одного элемента, рекурсивно вызвать процедуру для него.

Помимо вышеприведенного способа разбиения, называемого схема Хоара, существуют и другие, например схема Ломуто. Также для простоты можно выбирать опорным первый, случайный или средний элемент. Однако схема Хоара будет эффективней данных вариантов. Оптимальным опорным элементом является медиана всей последовательности, но её вычисление слишком затратно.

**Псевдокод:**

function quicksort(array a, low, high):

if low < high then

p := partition(a, low, high)

quicksort(a, low, p - 1)

quicksort(a, p + 1, high)

function partition(array a, low, high):

pivot := a[(low + high) / 2]

i := low - 1

j := high + 1

while true

do

i := i + 1

while a[i] < pivot

do

j := j - 1

while a[j] > pivot

if i >= j then

return j

swap a[i] with a[j]

**Свойства:**

* Неустойчивый
* Естественный, так как при частичной упорядоченности разделение массива более равномерное.
* Рекурсивный, глубина рекурсии O(log n).
* В худшем случае (массив упорядочен или его элементы равны) вычислительная сложность O(n2), в среднем и наилучшем случае O(n log n)
* Затраты памяти O(log n) дополнительно.

**Преимущества:**

* Один из самых быстрых алгоритмов сортировки, которые легко реализуются на практике
* Не вызывает большого количества кэш-промахов
* Легко организовать параллельную реализацию

**Недостатки:**

* Может возникнуть ошибка переполнения стека.
* При неудачных входных данных эффективность падает до O(n2).

**2.2** Выбором

**2.2.1** Сортировка выбором

Сортировка выбором – один из простейших алгоритмов сортировки, основная идея которого в выборе на каждой итерации в неотсортированной части массива минимального элемента и вставке его в конец отсортированной части.

**Алгоритм:**

1. Инициализируем индекс i, чтобы он указывал на первый элемент массива;
2. Находим минимальный элемент подмассива от i до n и запоминаем его номер в переменную;
3. Если номер i-го элемента и номер найденного элемента не совпадают, то производим обмен значениями
4. Увеличиваем i на 1 и переходим на шаг 2, пока i меньше размера массива n.

**Псевдокод:**

function selectionSort(array a):

for i = 0 to n - 2

min = i

for j = i + 1 to n - 1

if a[j] < a[min]

min = j

swap(a[i], a[min])

**Свойства:**

* Неустойчивый, но можно реализовать и устойчивый вариант при использовании структур данных, поддерживающих эффективные вставки и удаления.
* Нестественный
* В любом случае вычислительная сложность O(n2)
* Затраты памяти O(log n) дополнительно.

**Преимущества:**

* Простая реализация
* Эффективен при малых размерах массива в случаях, когда дополнительная память ограничена
* Вычислительная сложность относительно операций сравнения всегда O(n)
* Недостатки:
* Квадратичное время выполнения в любом случае. На почти отсортированных массивах работает так же долго, как и на случайных

**2.2.2** Пирамидальная сортировка

Пирамидальная сортировка (сортировка кучей) — алгоритм, основанный на использовании такой структуры данных как двоичная куча (или пирамида, сортирующее дерево).

Куча — это древовидная структура данных с некоторыми ограничениями. Двоичное дерево является убывающей кучей, если:

* Оно является полным, то есть все узлы имеют по два потомка, кроме листьев, которые расположены на одинаковой глубине.
* Все узлы в дереве больше своих потомков.

Если все узлы меньше своих потомков, то такая куча называется возрастающей.

Часто кучу представляют в памяти как массив А, в котором узлу с индексом i соответствуют потомки с индексами 2i+1 и 2i+2.

**Алгоритм:**

1. На базе исходного массива А строится куча. Максимальный элемент таким образом оказывается на вершине кучи.
   1. Начать построение пирамиды с a[k]...a[n], k = [size/2], так как эта часть массива не имеет потомков и удовлетворяет свойству пирамиды.
   2. Добавить k-1 й элемент к готовой части. Чтобы сохранялась пирамидальность выполняем следующие действия («просеивание»):
      1. Выбираем наибольшего из сыновей добавляемого элемента.
      2. Если этот элемент больше текущего, то меняем их местами и идем к шагу 1.2.1, взяв за текущий новое положение добавляемого элемента в массиве. Иначе просеивание завершено.
   3. Уменьшить k на 1 и перейти к шагу 1.2, пока не будут просеяны все вершины.
2. Берется верхний элемент кучи и меняется с последним. Таким образом максимальный элемент становится на свое место.
3. Строится куча из массива А[0]...А[n-1], для чего достаточно просеять новый первый элемент.
4. Повторяется шаг 2, пока обрабатываемая часть массива не уменьшится до одного элемента.

**Псевдокод:**

function Heapsort(array A):

BuildHeap(A)

for i = n to 1

swap(A[1], A[i])

n = n - 1

Heapify(A, 1)

function BuildHeap(array A):

n = elements\_in(A)

for i = floor(n/2) to 1

Heapify(A,i,n)

function Heapify(array A, int i, int n):

left = 2i

right = 2i+1

if (left <= n) and (A[left] > A[i])

max = left

else

max = i

if (right<=n) and (A[right] > A[max])

max = right

if (max != i)

swap(A[i], A[max])

Heapify(A, max)

**Свойства:**

* Неустойчивый
* Неестественный
* В любом случае вычислительная сложность O(n log n)
* Затраты памяти O(1) дополнительно.

**Преимущества:**

* Эффективна на больших массивах.
* Сравнима с быстрой сортировкой, хотя и немного медленнее, однако время выполнения не деградирует на неудачных данных, что является большим преимуществом в крупных системах с серьезными требованиями по безопасности.

**Недостатки:**

* Сложность реализации.
* Из-за сложности эффективна только на больших n. На небольших n (до нескольких тысяч) быстрее сортировка Шелла.
* На почти отсортированных массивах работает так же долго, как и на случайных
* Часто вызывает кэш-промахи.
* Неэффективна на структурах данных, не предоставляющих быстрый прямой доступ.

**2.3** Вставками

**2.3.1** Сортировка вставками

Сортировка вставками – один из простейших алгоритмов сортировки. На каждом шаге алгоритма выбирается один из элементов входных данных и помещается на нужную позицию в уже отсортированной части массива.

**Алгоритм:**

1. Инициализируем индекс i, чтобы он указывал на первый элемент массива A;
2. Берем элемент A[i] и вставляем на нужное место в готовую часть массива.
   1. Вставляемый элемент последовательно сравнивается с элементом, стоящим перед ним.
   2. Если вставляемый элемент меньше, то он меняется местами с предыдущим и процесс повторяется, иначе он остается на текущем месте и переход на шаг 3
3. Увеличиваем i на 1 и переходим на шаг 2, пока не будет обработан весь массив.

**Псевдокод:**

function insertionSort(a):

for i = 1 to n - 1

j = i - 1

while j ⩾ 0 and a[j] > a[j + 1]

swap(a[j], a[j + 1])

j—

**Свойства:**

* Устойчивый
* Естественный
* В среднем и худшем случае (отсортированный в обратном порядке массив) вычислительная сложность O(n2), в наилучшем случае (упорядоченный массив) O(n).
* Затраты памяти O(1) дополнительно.

**Преимущества:**

* Эффективна на небольших массивах.
* Эффективнее многих других квадратичных алгоритмов
* Простота реализации
* Может работать получая элементы из потока.

**Недостатки:**

* Неэффективен на больших массивах
* Требует много операций записи при сдвиге элементов массива

**2.3.2** Сортировка Шелла

Сортировка Шелла – усовершенствование сортировки вставками, при котором

сравниваются и сортируются между собой значения на некотором расстоянии {\displaystyle d} d, уменьшающемся на каждом шаге.

**Алгоритм:**

1. Задается начальное расстояние d1.
2. Подмассивы, состоящие из элементов, стоящих на расстоянии di сортируются обычной сортировкой вставками.
3. Получаем новое уменьшенное расстояние di.
4. Переходим на шаг 2, пока расстояние больше или равно единице.

Существует множество последовательностей расстояний для данного алгоритма, от выбора которых зависит временная сложность алгоритма. Сам

Шелл предлагал использовать hi=hi-1/2 при начальном значении h1=n/2, дающую скорость O(n2).

**Псевдокод:**

function shellSort(array a):

while interval < a.length /3 do:

interval = interval \* 3 + 1

while interval > 0 do:

for outer = interval; outer < a.length; outer ++ do:

valueToInsert = a[outer]

inner = outer;

while inner > interval -1 && a[inner - interval] >= valueToInsert do:

a[inner] = a[inner - interval]

inner = inner - interval

end while

a[inner] = valueToInsert

interval = (interval -1) /3;

**Свойства:**

* Неустойчивый
* Естественный
* В худшем случае (отсортированный в обратном порядке массив) вычислительная сложность O(n2), в наилучшем случае O(n log² n).
* В среднем случае временная сложность зависит от последовательности для циклических итераций.
* Последовательность Шелла [n/2], [n/4], [n/8], ..., дает скорость O(n2).
* Последовательность Седжвика дает O().
* Затраты памяти O(1) дополнительно.

**Преимущества:**

* Эффективна на больших массивах.
* Сравнима с быстрой сортировкой, хотя и медленнее, но не требует для выполнения стек.

**Недостатки:**

* Вызывает больше кэш-промахов, чем быстрая сортировка.

**2.3.3** Сортировка деревом

Сортировка деревом — алгоритм, идея которого в построении двоичного дерева поиска, с дальнейшим получением отсортированного массива с помощью прямого обхода дерева.

Двоичное дерево поиска – это двоичное дерево, для которого выполняются следующие условия:

* Оба поддерева являются двоичными деревьями поиска;
* У всех потомков левого поддерева любого узла значения ключей меньше, чем значение в узле.
* У всех потомков правого поддерева любого узла значения ключей не меньше, чем значение в узле.

**Алгоритм:**

1. Построение двоичного дерева.

2. Прямой обход дерева с записью результатов в выходной массив.

**Псевдокод:**

struct binaryTree

binaryTree left

object node

binaryTree right

function insert(binaryTree searchTree, object item)

if searchTree.node = null

searchTree.node =item

else

if item <= searchTree.node

insert(searchTree.left, item)

else

insert(searchTree.right, item)

function inOrder(binaryTree searchTree)

if searchTree.node=null

return

else

inOrder(searchTree.left)

addToResult(searchTree.node)

inOrder(searchTree.right)

function TreeSort(array a)

binaryTree searchTree

for i = 0 to n - 1

insert(searchTree, a[i])

inOrder(searchTree)

**Свойства:**

* Устойчивый
* Неестественный
* В худшем случае (отсортированный в прямом или обратном порядке массив) дерево вырождается в список и вычислительная сложность становится O(n2), в среднем и наилучшем случае O(n log n).
* При явном построении дерева в памяти требуется не менее 4n ячеек дополнительно (каждый узел должен содержать ссылки на элемент исходного массива, на родительский элемент, на левый и правый лист), однако, можно реализовать дерево неявно.

**Преимущества:**

* Относится к быстрым сортировкам
* Может работать, получая элементы из потока.

**Недостатки:**

* При рекурсивной реализации может случиться переполнение стека, особенно в случае вырождения дерева при отсортированном входном массиве

**2.4** Сортировка слиянием

Сортировка слиянием – популярный и эффективный алгоритм сортировки, предложенный Джоном фон Нейманом. В его основе, как и у быстрой сортировки, лежит принцип «разделяй и властвуй». Исходный массив разделяется практически равные части, каждая из которых рекурсивно и независимо сортируется, затем части объединяются.

**Алгоритм:**

1. Массив рекурсивно разбивается на две равные или практически равные части, далее каждая из половин делится, пока размер очередного подмассива не станет равным единице;
2. Два единичных массива объединяются в общий. Из каждого выбирается меньший (левый) элемент и записывается в свободную левую ячейку результирующего массива.
3. Из двух результирующих массивов собирается третий общий по такому же принципу. Если один из массивов заканчивается раньше другого, то оставшиеся элементы второго последовательно дописываются в результирующий массив;
4. Аналогично продолжать объединять получившиеся массивы, пока не получится отсортированный исходный массив.

**Псевдокод:**

function mergesort(array a):

if length(a) ≤ 1

return a

else

middle = length(m) / 2

for each x in a up to middle

add x to left

for each x in a after middle

add x to right

left = mergesort(left)

right = mergesort(right)

result = merge(left, right)

return result

**Свойства:**

* Устойчивый
* Неестественный. Существует естественный вариант, но он требует дополнительной памяти.
* В любом случае временная сложность O(n log n).
* Затраты памяти O(n) дополнительно

**Преимущества:**

* Относится к быстрым сортировкам
* Работает на структурах данных последовательного доступа.
* Хорошо сочетается с механизмом кэширования
* Несложно организовать параллельную реализацию

**Недостатки:**

* Требует дополнительной памяти равной размеру сортируемого массива.

**3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА ДЛЯ ИССЛЕДОВАНИЯ АЛГОРИТМОВ СОРТИРОВКИ**

* 1. Требования к программному средству

Для достижения целей курсового проектирования перед программным средством поставлены следующие задачи:

* Сортировка числовых массивов типа char. Данный тип был выбран для того, чтобы можно было с наименьшими затратами памяти хранить снимки сортируемого массива размером до 1000 элементов на каждом шаге алгоритма для последующей визуализации.
* Выбор одного из 10 вышеприведенных алгоритмов сортировки с возможностью повторной сортировки исходного массива новым алгоритмом.
* Получение ввода в текстовом поле и извлечение из него при наличии последовательности числовых данных.
* Генерации массива случайных чисел указанного пользователем размера.
* Вывод отсортированного массива в текстовое поле, времени выполнения сортировки, а также визуализации шагов алгоритма для массивов размером меньше 1000 элементов. Данный размер был выбран, чтобы даже для алгоритмов с наибольшим числом состояний (например сортировка пузырьком) затраты памяти на хранение снимков массива были приемлемы.

**3.2** Архитектура классов сортировки

Для реализации основных принципов ООП построена иерархия классов, приведенная на рисунке 3.1. При разработке архитектуры был применен паттерн «Стратегия» для удобной организации вызова различных алгоритмов сортировки единым методом с возможностью в дальнейшем легко дополнить программу новыми алгоритмами.

Основные классы:

* Sorter – главный класс, управляющий сортировкой. Его основной задачей является вызов функции сортировки у хранящегося внутри него объекта абстрактного класса SortingAlgorithm. Sorter поддерживает инициализацию алгоритмом через параметры конструктора и смену алгоритма через метод ChangeAlgorithm();
* SortingAlgorithm – абстрактный класс, от которого наследуются все классы алгоритмов сортировки. Они обязаны переопределить его чистую виртуальную функцию Sort(), с помощью которой запускается процесс сортировки конкретным алгоритмом. Некоторые наследники также содержат приватные методы, необходимые им для внутренних нужд (обычно рекурсивные).
* Tracker – класс, отвечающий за отслеживание выполнения алгоритма сортировки. В его задачи входит сохранение в вектор AlgorithmTracker текущего состояния массива с помощью функции SaveCurrentState(), для последующей визуализации процесса сортировки. Метод ClearStates() отвечает за очистку вектора, а StatesCount() за возврат его текущего размера. Также класс отслеживает время выполнения алгоритма с помощью функции clock() модуля ctime стандартной библиотеки. Поле startTime хранит время начала запуска текущего отсчета, elapsedTime – время выполнения алгоритма на данный момент. Для управления отсчетом времени используются методы StartTimer(), PauseTimer() и ClearTimer(). GetTime() возвращает время выполнения вызвавшему классу.

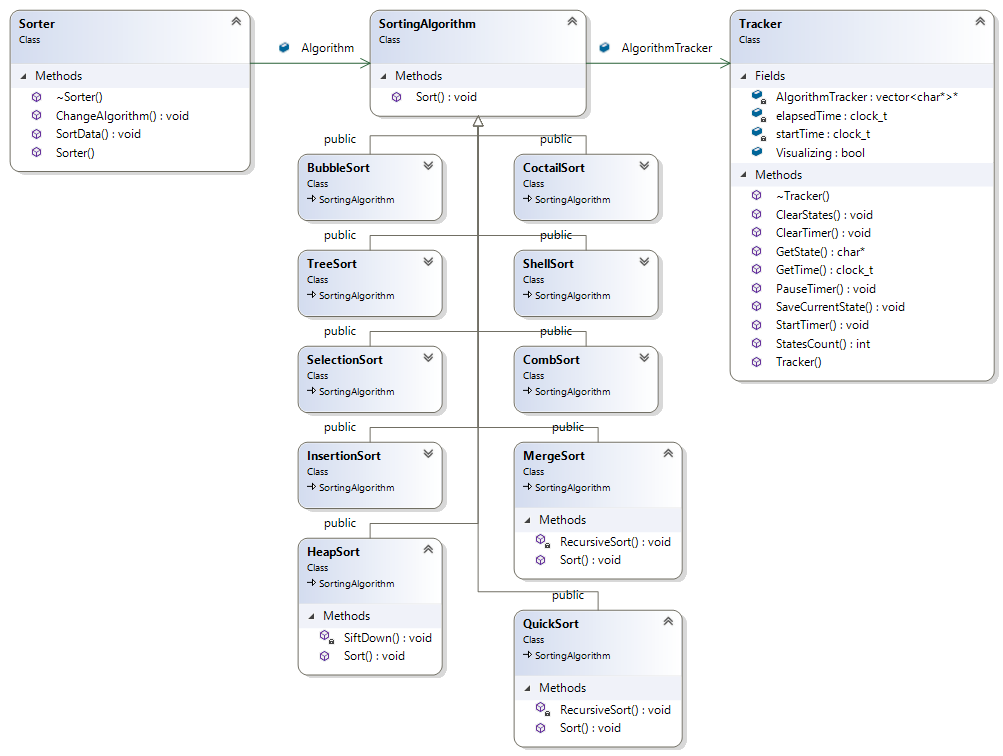


Рис. 3.1 – Архитектура классов сортировки программы

**3.3** Пользовательский интерфейс программы

Пользовательский интерфейс разработан с помощью библиотеки WinForms .NET и описан в модуле MainForm. Основные элементы интерфейса:

* MainForm – главный компонент приложения, содержащий все остальные элементы
* MySorter – указатель на объект класса Sorter, вызывающий алгоритмы сортировки
* MyDrawer – указатель на объект класса Drawer, единственная задача которого вывести на переданном ему в конструкторе объекте Chart^ StatesGraph все элементы полученного массива с помощью метода DrawArray().
* Различные компоненты интерфейса для взаимодействия с пользователем, такие как Button, TrackBar, TextBox, Label.
* Компоненты FlowLayoutPanel для организации расположения компонентов на главной форме.
* Конструктор и метод InitializeComponent() для инициализации формы.
* Обработчики различных действий пользователя, таких как нажатия кнопки, выбора элемента в comboBox или перемещения трэкпада.

Реализация пользовательского интерфейса с использованием данных компонентов приведена на рисунке 3.2

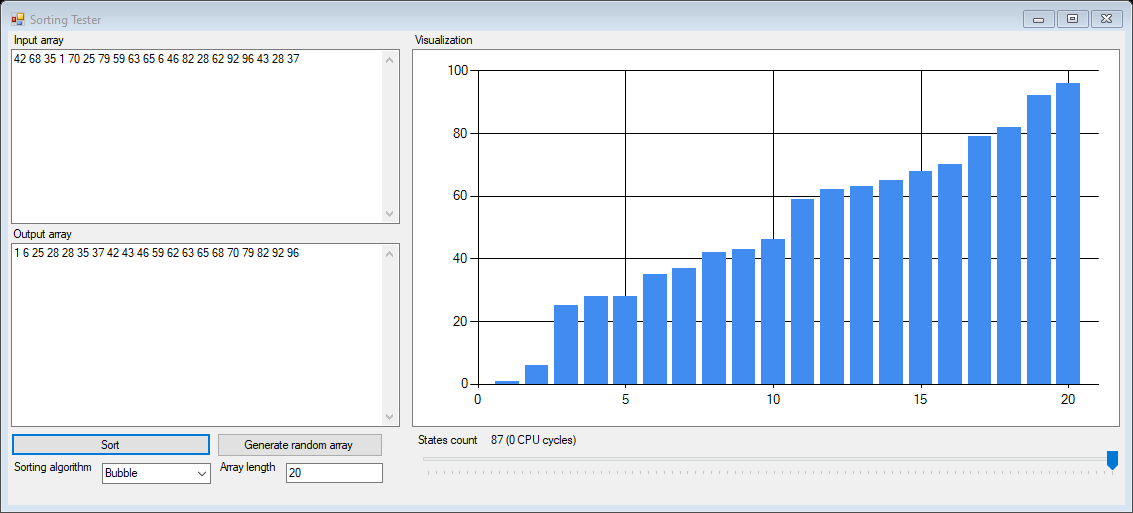


Рис. 3.2 – Интерфейс программы

В данном модуле используются методы классов и коллекции библиотеки .NET, например для парсинга строк, а также размещение данных в управляемой куче с использованием ссылок на управляемые объекты (задаются знаком «^» вместо «\*»), однако их использование сведено к необходимому минимуму.

**4 ТЕСТИРОВАНИЕ И АНАЛИЗ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ**

Для сравнения временных затрат выполнения на случайных массивах для каждого алгоритма было сгенерировано 3 массива, для каждого было выполнено 3 попытки сортировки. Время выполнения измерялось в единицах, возвращаемых функцией *clock() стандартного модуля ctime.* Средний результат для каждого метода при фиксированном размере массива приведен в таблице 4.1.

Таблица 4.1 – Результаты тестирования №1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Алгоритм | Время выполнения для каждого размера массива | | | | | |
| 4000 | 8000 | 16000 | 32000 | 64000 | 128000 |
| Сортировка пузырьком | 79 | 330 | 1408 | 5477 | 21648 | 94577 |
| Сортировка перемешиванием | 50 | 200 | 795 | 3222 | 12864 | 63042 |
| Сортировка расчёской | <1 | 1 | 2 | 4 | 11 | 21 |
| Быстрая сортировка | <1 | 1 | 2 | 3 | 6 | 16 |
| Сортировка выбором | 18 | 70 | 274 | 1090 | 4366 | 17449 |
| Пирамидальная сортировка | 1 | 2 | 3 | 7 | 14 | 29 |
| Сортировка вставками | 11 | 43 | 170 | 679 | 2706 | 15107 |
| Сортировка Шелла | <1 | 1 | 2 | 5 | 10 | 21 |
| Сортировка деревом | 3 | 8 | 25 | 101 | 415 | 2040 |
| Сортировка слиянием | 2 | 5 | 9 | 24 | 49 | 95 |

Для наглядности на рисунке 4.1 приведены графики для каждого алгоритма. Также пунктиром обозначены функции .

Рисунок 4.1 – Графики временных затрат для неупорядоченных массивов

Аналогично предыдущему был проведен тест на упорядоченных массивах, генерируемых сортировкой одним из методов случайного массива. Результаты теста приведены в таблице 4.2. Графики временных затрат отображены на рисунке 4.2.

Таблица 4.2 – Результаты тестирования №2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Алгоритм | Время выполнения для каждого размера массива | | | | | |
| 4000 | 8000 | 16000 | 32000 | 64000 | 128000 |
| Сортировка пузырьком | <1 | <1 | <1 | <1 | <1 | <1 |
| Сортировка перемешиванием | <1 | <1 | <1 | <1 | <1 | <1 |
| Сортировка расчёской | <1 | <1 | 1 | 3 | 7 | 13 |
| Быстрая сортировка | <1 | <1 | 1 | 2 | 5 | 8 |
| Сортировка выбором | 17 | 68 | 275 | 1089 | 4369 | 17454 |
| Пирамидальная сортировка | 1 | 1 | 2 | 6 | 13 | 24 |
| Сортировка вставками | <1 | <1 | <1 | <1 | 1 | 1 |
| Сортировка Шелла | <1 | <1 | 1 | 3 | 5 | 11 |
| Сортировка деревом | 80 | 355 | 1461 | StOver | StOver | StOver |
| Сортировка слиянием | 2 | 4 | 8 | 22 | 44 | 90 |

Рисунок 4.2 – Графики временных затрат для упорядоченных массивов

Для данной реализации сортировки деревом массивы размером более 16000 элементов вызывали переполнение стека в связи с большой глубиной рекурсии. Это связано с тем, что дерево, построенное по отсортированному массиву, вырождается в список. Следовательно, для больших частично упорядоченных массивов следует использовать другие методы сортировки или итеративную реализацию построения дерева.

**5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ**

Перед началом тестирования алгоритмов сортировки необходимо ввести числовой массив. Это можно сделать вручную, введя последовательность чисел от -128 до 127, разделенную пробелами. Также можно сгенерировать массив случайных чисел, указав в поле «Array length» размер массива и нажав «Generate random array». Далее в выпадающем меню «Sorting algorithm» необходимо выбрать желаемый алгоритм сортировки и нажать «Sort». Если исходный массив меньше 1000 элементов, то справа будет доступна визуализация процесса сортировки. Передвигая слайдер можно отследить вид сортируемого массива на различных шагах алгоритма. Для любого размера массива справа будет отображён окончательный вид отсортированного массива и под ним указано время выполнения в единицах, возвращаемых функцией clock() модуля ctime. После сортировки можно ввести новый массив или еще раз отсортировать предыдущий, выбрав другой алгоритм сортировки.

**ЗАКЛЮЧЕНИЕ**

В данной курсовой работе рассмотрены наиболее известные алгоритмы сортировки, их реализации, анализ, а также преимущества и недостатки. Для проверки основных свойств данных методов было реализовано программное средство для сортировки числовых массивов с визуализацией процесса сортировки и оценкой скорости выполнения. С помощью данной программы проводилось тестирование алгоритмов для различных размеров входных массивов и степеней упорядоченности. Проведенное тестирование программы показывает корректность её работы, а также демонстрирует на практике основные свойства каждого алгоритма.

По мнению автора программы, основные задачи курсового проектирования выполнены и спроектированную программу можно использовать в учебных целях для наглядного изучения и сравнения различных методов сортировки.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Кнут Д. Э. Искусство программирования. Том 3. Сортировка и поиск / Кнут Д. Э. – Москва. : Вильямс, 2007. – 832 с.

[2] Седжвик Р. Фундаментальные алгоритмы на C. Анализ/Структуры данных/Сортировка/Поиск / Седжвик Р. – СПб.: ДиаСофтЮП, 2003. – 672 с.

[3] Вирт Н. Алгоритмы и структуры данных/ Вирт Н. – Москва.: Мир, 1989. – 360 с.

[4] Описание алгоритмов сортировки и сравнение их производительности [Электронный ресурс] – Режим доступа: https://habr.com/ru/post/335920/

[5] Алгоритм сортировки [Электронный ресурс] – Режим доступа:

http://ru.wikipedia.org/wiki/Алгоритм\_сортировки

**ПРИЛОЖЕНИЕ А**

**(обязательное)**

**Текст главного программного модуля**

MainForm.h

#pragma once

#include "SortingAlgorithms.h"

#include "Drawer.h"

#include <String>

#include <sstream>

namespace SortingTest {

using namespace System;

using namespace System::ComponentModel;

using namespace System::Collections;

using namespace System::Windows::Forms;

using namespace System::Data;

using namespace System::Drawing;

using namespace DataVisualization::Charting;

/// <summary>

/// Summary for MainForm

/// </summary>

public ref class MainForm : public System::Windows::Forms::Form

{

private: System::Windows::Forms::ComboBox^ AlgorithmComboBox;

private: System::Windows::Forms::Button^ GenerateArrayButton;

private: System::Windows::Forms::TextBox^ ArrayLengthBox;

private: System::Windows::Forms::Label^ CountLabel;

private: System::Windows::Forms::TextBox^ OutputTextBox;

private: System::Windows::Forms::Label^ ArrayLengthLabel;

private: System::Windows::Forms::Label^ SortingAlgorithmLabel;

private: System::Windows::Forms::FlowLayoutPanel^ flowLayoutPanel1;

private: System::Windows::Forms::Label^ StatesCountLabel;

private: System::Windows::Forms::Label^ InputArrayLabel;

private: System::Windows::Forms::Label^ OutpuLabel;

private: System::Windows::Forms::FlowLayoutPanel^ flowLayoutPanel2;

private: System::Windows::Forms::FlowLayoutPanel^ flowLayoutPanel3;

private: System::Windows::Forms::Label^ VisualizationLabel;

private: System::Windows::Forms::TextBox^ InputTextBox;

private: System::Windows::Forms::Button^ SortButton;

private: System::Windows::Forms::DataVisualization::Charting::Chart^ StatesGraph;

private: System::Windows::Forms::TrackBar^ StatesTrackBar;

public:

Sorter\* MySorter;

int ArraySize;

char\* ResultArray;

Drawer^ MyDrawer; // Object for visualizing changing states of array

MainForm(void)

{

InitializeComponent();

MySorter = new Sorter(new BubbleSort()); //Initialize Sorter with BubbleSort algorithm by default

MyDrawer = gcnew Drawer(StatesGraph); //Initialize Sorter with StatesGraph component to draw

}

protected:

~MainForm()

{

if (components)

{

delete components;

}

}

private:

/// <summary>

/// Required designer variable.

/// </summary>

System::ComponentModel::Container^ components;

#pragma region Windows Form Designer generated code

/// <summary>

/// Required method for Designer support - do not modify

/// the contents of this method with the code editor.

/// </summary>

void InitializeComponent(void)

{

System::Windows::Forms::DataVisualization::Charting::ChartArea^ chartArea1 = (gcnew System::Windows::Forms::DataVisualization::Charting::ChartArea());

System::Windows::Forms::DataVisualization::Charting::Series^ series1 = (gcnew System::Windows::Forms::DataVisualization::Charting::Series());

this->InputTextBox = (gcnew System::Windows::Forms::TextBox());

this->SortButton = (gcnew System::Windows::Forms::Button());

this->StatesGraph = (gcnew System::Windows::Forms::DataVisualization::Charting::Chart());

this->StatesTrackBar = (gcnew System::Windows::Forms::TrackBar());

this->AlgorithmComboBox = (gcnew System::Windows::Forms::ComboBox());

this->GenerateArrayButton = (gcnew System::Windows::Forms::Button());

this->ArrayLengthBox = (gcnew System::Windows::Forms::TextBox());

this->CountLabel = (gcnew System::Windows::Forms::Label());

this->OutputTextBox = (gcnew System::Windows::Forms::TextBox());

this->ArrayLengthLabel = (gcnew System::Windows::Forms::Label());

this->SortingAlgorithmLabel = (gcnew System::Windows::Forms::Label());

this->flowLayoutPanel1 = (gcnew System::Windows::Forms::FlowLayoutPanel());

this->InputArrayLabel = (gcnew System::Windows::Forms::Label());

this->OutpuLabel = (gcnew System::Windows::Forms::Label());

this->StatesCountLabel = (gcnew System::Windows::Forms::Label());

this->flowLayoutPanel2 = (gcnew System::Windows::Forms::FlowLayoutPanel());

this->flowLayoutPanel3 = (gcnew System::Windows::Forms::FlowLayoutPanel());

this->VisualizationLabel = (gcnew System::Windows::Forms::Label());

(cli::safe\_cast<System::ComponentModel::ISupportInitialize^>(this->StatesGraph))->BeginInit();

(cli::safe\_cast<System::ComponentModel::ISupportInitialize^>(this->StatesTrackBar))->BeginInit();

this->flowLayoutPanel1->SuspendLayout();

this->flowLayoutPanel2->SuspendLayout();

this->flowLayoutPanel3->SuspendLayout();

this->SuspendLayout();

//

// InputTextBox

//

this->InputTextBox->Location = System::Drawing::Point(3, 16);

this->InputTextBox->MaxLength = 500000;

this->InputTextBox->Multiline = true;

this->InputTextBox->Name = L"InputTextBox";

this->InputTextBox->ScrollBars = System::Windows::Forms::ScrollBars::Vertical;

this->InputTextBox->Size = System::Drawing::Size(389, 175);

this->InputTextBox->TabIndex = 0;

//

// SortButton

//

this->SortButton->Location = System::Drawing::Point(3, 400);

this->SortButton->Name = L"SortButton";

this->SortButton->Size = System::Drawing::Size(200, 23);

this->SortButton->TabIndex = 1;

this->SortButton->Text = L"Sort";

this->SortButton->UseVisualStyleBackColor = true;

this->SortButton->Click += gcnew System::EventHandler(this, &MainForm::SortButton\_Click);

//

// StatesGraph

//

this->StatesGraph->BorderlineColor = System::Drawing::Color::Gray;

this->StatesGraph->BorderlineDashStyle = System::Windows::Forms::DataVisualization::Charting::ChartDashStyle::Solid;

chartArea1->Name = L"ChartArea1";

this->StatesGraph->ChartAreas->Add(chartArea1);

this->StatesGraph->Location = System::Drawing::Point(3, 16);

this->StatesGraph->Name = L"StatesGraph";

series1->ChartArea = L"ChartArea1";

series1->Name = L"Series1";

this->StatesGraph->Series->Add(series1);

this->StatesGraph->Size = System::Drawing::Size(708, 378);

this->StatesGraph->TabIndex = 2;

this->StatesGraph->Text = L"chart1";

//

// StatesTrackBar

//

this->StatesTrackBar->Location = System::Drawing::Point(3, 16);

this->StatesTrackBar->Maximum = 0;

this->StatesTrackBar->Name = L"StatesTrackBar";

this->StatesTrackBar->Size = System::Drawing::Size(711, 45);

this->StatesTrackBar->TabIndex = 5;

this->StatesTrackBar->Scroll += gcnew System::EventHandler(this, &MainForm::StatesTrackBar\_Scroll);

//

// AlgorithmComboBox

//

this->AlgorithmComboBox->AllowDrop = true;

this->AlgorithmComboBox->FormattingEnabled = true;

this->AlgorithmComboBox->Items->AddRange(gcnew cli::array< System::Object^ >(10) {

L"Bubble", L"Coctail", L"Comb", L"Heap",

L"Insertion", L"Merge", L"Quick", L"Selection", L"Shell", L"Tree"

});

this->AlgorithmComboBox->Location = System::Drawing::Point(94, 430);

this->AlgorithmComboBox->Name = L"AlgorithmComboBox";

this->AlgorithmComboBox->Size = System::Drawing::Size(109, 21);

this->AlgorithmComboBox->TabIndex = 6;

this->AlgorithmComboBox->Text = L"Bubble";

this->AlgorithmComboBox->SelectedIndexChanged += gcnew System::EventHandler(this, &MainForm::AlgorithmComboBox\_SelectedIndexChanged);

//

// GenerateArrayButton

//

this->GenerateArrayButton->Location = System::Drawing::Point(209, 400);

this->GenerateArrayButton->Name = L"GenerateArrayButton";

this->GenerateArrayButton->Size = System::Drawing::Size(166, 24);

this->GenerateArrayButton->TabIndex = 7;

this->GenerateArrayButton->Text = L"Generate random array";

this->GenerateArrayButton->UseVisualStyleBackColor = true;

this->GenerateArrayButton->Click += gcnew System::EventHandler(this, &MainForm::GenerateArrayButton\_Click);

//

// ArrayLengthBox

//

this->ArrayLengthBox->Location = System::Drawing::Point(278, 430);

this->ArrayLengthBox->Name = L"ArrayLengthBox";

this->ArrayLengthBox->Size = System::Drawing::Size(97, 20);

this->ArrayLengthBox->TabIndex = 8;

this->ArrayLengthBox->Text = L"20";

//

// CountLabel

//

this->CountLabel->AutoSize = true;

this->CountLabel->Location = System::Drawing::Point(76, 0);

this->CountLabel->Name = L"CountLabel";

this->CountLabel->Size = System::Drawing::Size(13, 13);

this->CountLabel->TabIndex = 9;

this->CountLabel->Text = L"0";

this->CountLabel->TextAlign = System::Drawing::ContentAlignment::MiddleCenter;

//

// OutputTextBox

//

this->OutputTextBox->BackColor = System::Drawing::SystemColors::Window;

this->OutputTextBox->Dock = System::Windows::Forms::DockStyle::Bottom;

this->OutputTextBox->Location = System::Drawing::Point(3, 210);

this->OutputTextBox->Multiline = true;

this->OutputTextBox->Name = L"OutputTextBox";

this->OutputTextBox->ReadOnly = true;

this->OutputTextBox->ScrollBars = System::Windows::Forms::ScrollBars::Vertical;

this->OutputTextBox->Size = System::Drawing::Size(389, 184);

this->OutputTextBox->TabIndex = 10;

//

// ArrayLengthLabel

//

this->ArrayLengthLabel->Anchor = static\_cast<System::Windows::Forms::AnchorStyles>((((System::Windows::Forms::AnchorStyles::Top | System::Windows::Forms::AnchorStyles::Bottom)

| System::Windows::Forms::AnchorStyles::Left)

| System::Windows::Forms::AnchorStyles::Right));

this->ArrayLengthLabel->AutoSize = true;

this->ArrayLengthLabel->Location = System::Drawing::Point(209, 427);

this->ArrayLengthLabel->Name = L"ArrayLengthLabel";

this->ArrayLengthLabel->Size = System::Drawing::Size(63, 27);

this->ArrayLengthLabel->TabIndex = 11;

this->ArrayLengthLabel->Text = L"Array length";

//

// SortingAlgorithmLabel

//

this->SortingAlgorithmLabel->AutoSize = true;

this->SortingAlgorithmLabel->Location = System::Drawing::Point(3, 427);

this->SortingAlgorithmLabel->Name = L"SortingAlgorithmLabel";

this->SortingAlgorithmLabel->Size = System::Drawing::Size(85, 13);

this->SortingAlgorithmLabel->TabIndex = 12;

this->SortingAlgorithmLabel->Text = L"Sorting algorithm";

//

// flowLayoutPanel1

//

this->flowLayoutPanel1->AutoSizeMode = System::Windows::Forms::AutoSizeMode::GrowAndShrink;

this->flowLayoutPanel1->Controls->Add(this->InputArrayLabel);

this->flowLayoutPanel1->Controls->Add(this->InputTextBox);

this->flowLayoutPanel1->Controls->Add(this->OutpuLabel);

this->flowLayoutPanel1->Controls->Add(this->OutputTextBox);

this->flowLayoutPanel1->Controls->Add(this->SortButton);

this->flowLayoutPanel1->Controls->Add(this->GenerateArrayButton);

this->flowLayoutPanel1->Controls->Add(this->SortingAlgorithmLabel);

this->flowLayoutPanel1->Controls->Add(this->AlgorithmComboBox);

this->flowLayoutPanel1->Controls->Add(this->ArrayLengthLabel);

this->flowLayoutPanel1->Controls->Add(this->ArrayLengthBox);

this->flowLayoutPanel1->Location = System::Drawing::Point(0, 2);

this->flowLayoutPanel1->Name = L"flowLayoutPanel1";

this->flowLayoutPanel1->Size = System::Drawing::Size(392, 477);

this->flowLayoutPanel1->TabIndex = 14;

//

// InputArrayLabel

//

this->InputArrayLabel->AutoSize = true;

this->InputArrayLabel->Location = System::Drawing::Point(3, 0);

this->InputArrayLabel->Name = L"InputArrayLabel";

this->InputArrayLabel->Size = System::Drawing::Size(57, 13);

this->InputArrayLabel->TabIndex = 16;

this->InputArrayLabel->Text = L"Input array";

//

// OutpuLabel

//

this->OutpuLabel->AutoSize = true;

this->OutpuLabel->Location = System::Drawing::Point(3, 194);

this->OutpuLabel->Name = L"OutpuLabel";

this->OutpuLabel->Size = System::Drawing::Size(65, 13);

this->OutpuLabel->TabIndex = 17;

this->OutpuLabel->Text = L"Output array";

//

// StatesCountLabel

//

this->StatesCountLabel->AutoSize = true;

this->StatesCountLabel->Location = System::Drawing::Point(3, 0);

this->StatesCountLabel->Name = L"StatesCountLabel";

this->StatesCountLabel->Size = System::Drawing::Size(67, 13);

this->StatesCountLabel->TabIndex = 15;

this->StatesCountLabel->Text = L"States count";

//

// flowLayoutPanel2

//

this->flowLayoutPanel2->Controls->Add(this->StatesCountLabel);

this->flowLayoutPanel2->Controls->Add(this->CountLabel);

this->flowLayoutPanel2->Controls->Add(this->StatesTrackBar);

this->flowLayoutPanel2->Location = System::Drawing::Point(3, 400);

this->flowLayoutPanel2->Name = L"flowLayoutPanel2";

this->flowLayoutPanel2->Size = System::Drawing::Size(714, 61);

this->flowLayoutPanel2->TabIndex = 16;

//

// flowLayoutPanel3

//

this->flowLayoutPanel3->Controls->Add(this->VisualizationLabel);

this->flowLayoutPanel3->Controls->Add(this->StatesGraph);

this->flowLayoutPanel3->Controls->Add(this->flowLayoutPanel2);

this->flowLayoutPanel3->Location = System::Drawing::Point(401, 2);

this->flowLayoutPanel3->Name = L"flowLayoutPanel3";

this->flowLayoutPanel3->Size = System::Drawing::Size(716, 477);

this->flowLayoutPanel3->TabIndex = 17;

//

// VisualizationLabel

//

this->VisualizationLabel->AutoSize = true;

this->VisualizationLabel->Location = System::Drawing::Point(3, 0);

this->VisualizationLabel->Name = L"VisualizationLabel";

this->VisualizationLabel->Size = System::Drawing::Size(65, 13);

this->VisualizationLabel->TabIndex = 18;

this->VisualizationLabel->Text = L"Visualization";

//

// MainForm

//

this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);

this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;

this->ClientSize = System::Drawing::Size(1117, 474);

this->Controls->Add(this->flowLayoutPanel3);

this->Controls->Add(this->flowLayoutPanel1);

this->Name = L"MainForm";

this->Text = L"Sorting Tester";

(cli::safe\_cast<System::ComponentModel::ISupportInitialize^>(this->StatesGraph))->EndInit();

(cli::safe\_cast<System::ComponentModel::ISupportInitialize^>(this->StatesTrackBar))->EndInit();

this->flowLayoutPanel1->ResumeLayout(false);

this->flowLayoutPanel1->PerformLayout();

this->flowLayoutPanel2->ResumeLayout(false);

this->flowLayoutPanel2->PerformLayout();

this->flowLayoutPanel3->ResumeLayout(false);

this->flowLayoutPanel3->PerformLayout();

this->ResumeLayout(false);

}

#pragma endregion

private: System::Void SortButton\_Click(System::Object^ sender, System::EventArgs^ e) {

//Getting data from inputbox

OutputTextBox->Clear();

if (InputTextBox->Text->Length == 0)

{

OutputTextBox->Text = "Nothing to sort";

return;

}

array<String^>^ nums = InputTextBox->Text->Split(gcnew array<wchar\_t> {' ', ','});

//Parsing string data to ints

System::Collections::Generic::List<char>^ ints = gcnew System::Collections::Generic::List<char>();

for each (String ^ MySorter in nums)

{

int ParseResult;

if (System::Int32::TryParse(MySorter, ParseResult))

{

ints->Add(ParseResult);

}

}

if (ints->Count == 0)

{

OutputTextBox->Text = "Haven`t found any numbers";

return;

}

//Preparing an array to sort

ArraySize = ints->Count;

if (ResultArray)

delete[] ResultArray;

ResultArray = new char[ArraySize];

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

{

ResultArray[i] = (char)ints[i];

}

//Sort and draw results if enabled

MySorter->Algorithm->AlgorithmTracker.Visualizing = (ArraySize > 1000) ? false : true;

MySorter->SortData(ResultArray, ArraySize);

MyDrawer->DrawArray(ResultArray, ArraySize);

//Result output

std::string str = "";

std::stringstream ss;

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

{

ss << (short)ResultArray[i] << " ";

}

str = ss.str();

OutputTextBox->Text = gcnew String(str.c\_str());

StatesTrackBar->Maximum = MySorter->Algorithm->AlgorithmTracker.StatesCount() - 1;

StatesTrackBar->Value = StatesTrackBar->Maximum;

CountLabel->Text = (ArraySize > 1000) ? "Visualization of states is disabled for arrays of size>1000" : StatesTrackBar->Value.ToString();

CountLabel->Text += " (" + MySorter->Algorithm->AlgorithmTracker.GetTime().ToString() + " CPU cycles)";

}

private: System::Void StatesTrackBar\_Scroll(System::Object^ sender, System::EventArgs^ e) {

MyDrawer->DrawArray(MySorter->Algorithm->AlgorithmTracker.GetState(StatesTrackBar->Value), ArraySize);

CountLabel->Text = StatesTrackBar->Value.ToString();

}

private: System::Void AlgorithmComboBox\_SelectedIndexChanged(System::Object^ sender, System::EventArgs^ e) {

switch (AlgorithmComboBox->SelectedIndex)

{

case 0:MySorter->ChangeAlgorithm(new BubbleSort()); break;

case 1:MySorter->ChangeAlgorithm(new CoctailSort()); break;

case 2:MySorter->ChangeAlgorithm(new CombSort()); break;

case 3:MySorter->ChangeAlgorithm(new HeapSort()); break;

case 4:MySorter->ChangeAlgorithm(new InsertionSort()); break;

case 5:MySorter->ChangeAlgorithm(new MergeSort()); break;

case 6:MySorter->ChangeAlgorithm(new QuickSort()); break;

case 7:MySorter->ChangeAlgorithm(new SelectionSort()); break;

case 8:MySorter->ChangeAlgorithm(new ShellSort()); break;

case 9:MySorter->ChangeAlgorithm(new TreeSort()); break;

default:

MySorter->ChangeAlgorithm(new BubbleSort()); break;

}

}

private: System::Void GenerateArrayButton\_Click(System::Object^ sender, System::EventArgs^ e) {

ArraySize = System::Int32::Parse(ArrayLengthBox->Text); //Getting size of an array from input

InputTextBox->Clear();

//Generating random numbers

System::Collections::Generic::List<char>^ rands = gcnew System::Collections::Generic::List<char>(ArraySize);

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

{

rands->Add(rand() % 100 + 1);

}

// Printing generated numbers to inputbox

std::string str = "";

std::stringstream ss;

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

{

ss << (short)rands[i] << " ";

}

str = ss.str();

InputTextBox->Text = gcnew String(str.c\_str());

}

};

}

MainForm.cpp

#include "MainForm.h"

using namespace SortingTest;

using namespace System;

using namespace System::Windows::Forms;

[STAThreadAttribute]

int Main(array<String^>^ args) {

Application::EnableVisualStyles();

Application::SetCompatibleTextRenderingDefault(false);

MainForm form;

Application::Run(% form);

return 0;

}

**ПРИЛОЖЕНИЕ Б**

**(обязательное)**

**Текст программного модуля с реализацией алгоритмов сортировки**

SortingAlgorithms.h

|  |
| --- |
|  |
|  |  |

#pragma once

#include "Tracker.h"

class SortingAlgorithm

{

public:

void virtual Sort(char\* data, int size) = 0;

Tracker AlgorithmTracker;

};

class BubbleSort :public SortingAlgorithm

{

public:

void Sort(char\* data, int size);

};

class CoctailSort :public SortingAlgorithm

{

public:

void Sort(char\* data, int size);

};

class CombSort :public SortingAlgorithm

{

public:

void Sort(char\* data, int size);

};

class SelectionSort :public SortingAlgorithm

{

public:

void Sort(char\* data, int size);

};

class InsertionSort :public SortingAlgorithm

{

public:

void Sort(char\* data, int size);

};

class ShellSort :public SortingAlgorithm

{

public:

void Sort(char\* data, int size);

};

class QuickSort :public SortingAlgorithm

{

public:

void Sort(char\* data, int size);

private:

void RecursiveSort(char\* data, int size, int begin, int end);

};

class TreeSort :public SortingAlgorithm

{

public:

void Sort(char\* data, int size);

};

class HeapSort :public SortingAlgorithm

{

public:

void Sort(char\* data, int size);

private:

void SiftDown(char\* data, int size, int begin, int end); //Recursive internal procedure for sifting small elements down to build heap

};

class MergeSort :public SortingAlgorithm

{

public:

void Sort(char\* data, int size);

private:

void RecursiveSort(char\* data, int size, int begin, int end); //Recursive internal procedure function for sorting

};

class Sorter

{

public:

Sorter(SortingAlgorithm\* alg);

~Sorter();

void SortData(char\* data, int size);

void ChangeAlgorithm(SortingAlgorithm\* alg);

SortingAlgorithm\* Algorithm;

};

SortingAlgorithms.cpp

#include "SortingAlgorithms.h"

//Internal function for swapping 2 elements of and array

void swap(char\* a, char\* b) {

char tmp = \*a;

\*a = \*b;

\*b = tmp;

}

// Internal type for TreeSort algorithm

class BinaryTree

{

public:

BinaryTree();

~BinaryTree();

void AddNode(char key);

void Linearize(char\* result);

private:

struct Node

{

Node \* left, \* right;

char key;

};

Node\* Root;

void RecursiveLinearize(Node\* node, int\* pos, char\* result); // LNR traversal with output to char[] array

void RecursiveAdd(char key, Node\* node); // Internal function to add Node

void RecursiveDelete(Node\* node); // Internal function to delete tree during LRN traversal

};

BinaryTree::BinaryTree()

{

Root = 0;

}

BinaryTree::~BinaryTree()

{

RecursiveDelete(Root);

}

void BinaryTree::AddNode(char key)

{

if (!Root)

{

Root = new Node();

Root->key = key;

Root->left = 0;

Root->right = 0;

}

else

{

RecursiveAdd(key, Root);

}

}

void BinaryTree::Linearize(char\* result)

{

int position = 0; // Position in result array to insert key

RecursiveLinearize(Root, &position, result);

}

void BinaryTree::RecursiveLinearize(Node\* node, int\* pos, char\* result)

{

if (node)

{

RecursiveLinearize(node->left, pos, result);

result[\*pos] = node->key;

(\*pos)++;

RecursiveLinearize(node->right, pos, result);

}

}

void BinaryTree::RecursiveDelete(Node\* node)

{

if (node)

{

RecursiveDelete(node->left);

RecursiveDelete(node->right);

delete node;

}

}

void BinaryTree::RecursiveAdd(char key, Node\* node)

{

Node\* tmp;

bool left; //bool to check for left or right son of parent

if (key < node->key)

{

tmp =node->left;

left = true;

}

else

{

tmp = node->right;

left = false;

}

if (!tmp) //if current node doesnt exit

{

tmp = new Node();

tmp->key = key;

tmp->left = 0;

tmp->right = 0;

if (left)

{

node->left = tmp;

}

else

{

node->right = tmp;

}

}

else

{

RecursiveAdd(key,tmp);

}

}

Sorter::Sorter(SortingAlgorithm\* alg) :Algorithm(alg)

{

}

Sorter::~Sorter()

{

delete Algorithm;

}

void Sorter::SortData(char\* data, int size)

{

Algorithm->AlgorithmTracker.ClearStates();

Algorithm->AlgorithmTracker.ClearTimer();

Algorithm->AlgorithmTracker.StartTimer();

Algorithm->Sort(data, size);

Algorithm->AlgorithmTracker.PauseTimer();

}

void Sorter::ChangeAlgorithm(SortingAlgorithm\* alg)

{

if (Algorithm)

{

delete Algorithm;

}

Algorithm = alg;

}

void BubbleSort::Sort(char\* data, int size)

{

AlgorithmTracker.SaveCurrentState(data, size);

bool SwapHappened;

do {

SwapHappened = false;

for (int i = 0; i < size - 1; i++)

{

if (data[i + 1] < data[i])

{

swap(&data[i], &data[i+1]);

SwapHappened = true;

AlgorithmTracker.SaveCurrentState(data, size);

}

}

} while (SwapHappened);

}

void SelectionSort::Sort(char\* data, int size)

{

AlgorithmTracker.SaveCurrentState(data, size);

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

{

int indexOfMin = i;

for (int j = i; j < size; j++)

{

if (data[j]<data[indexOfMin])

{

indexOfMin = j;

}

}

swap(&data[indexOfMin], &data[i]);

AlgorithmTracker.SaveCurrentState(data, size);

}

}

void InsertionSort::Sort(char\* data, int size)

{

AlgorithmTracker.SaveCurrentState(data, size);

for (int i = 1; i < size; i++)

{

char current = data[i];

int j = i - 1;

while ((j>=0)&&(current<data[j]))

{

data[j + 1] = data[j];

j--;

}

data[j + 1] = current;

AlgorithmTracker.SaveCurrentState(data, size);

}

}

void ShellSort::Sort(char\* data, int size)

{

AlgorithmTracker.SaveCurrentState(data, size);

vector<int> SedgewickSeq;

int SeqElement=0;

for (int i = 0; SeqElement < size; i++)

{

SeqElement = (i % 2 == 0) ? (9\*pow(2,i)-9\*pow(2,i/2)+1) : (8 \* pow(2, i) - 6 \* pow(2, (i+1)/ 2) + 1) ;

SedgewickSeq.push\_back(SeqElement);

}

SedgewickSeq.pop\_back();

while (SedgewickSeq.size()>0)

{

int step = SedgewickSeq.back();

SedgewickSeq.pop\_back();

for (int i = step; i < size; i++)

{

char current = data[i];

int j = i - step;

while ((j >= 0) && (current < data[j]))

{

data[j + step] = data[j];

j-= step;

}

data[j + step] = current;

AlgorithmTracker.SaveCurrentState(data, size);

}

}

}

void QuickSort::Sort(char\* data, int size)

{

AlgorithmTracker.SaveCurrentState(data, size);

RecursiveSort(data, size, 0, size-1);

}

void QuickSort::RecursiveSort(char\* data, int size, int begin, int end)

{

int i = begin, j = end;

int center = data[(begin + end) / 2];

do {

while (data[i] < center)

i++;

while (data[j] > center)

j--;

if (i <= j)

{

if (i < j)

{

swap(&data[i], &data[j]);

AlgorithmTracker.SaveCurrentState(data, size);

}

i++;

j--;

}

} while (i <= j);

if (i < end)

RecursiveSort(data, size, i, end);

if (begin < j)

RecursiveSort(data, size, begin, j);

}

void CoctailSort::Sort(char\* data, int size)

{

AlgorithmTracker.SaveCurrentState(data, size);

bool SwapHappened;

int begin = 0;

int end = size;

do {

SwapHappened = false;

for (int i = begin; i < end - 1; i++)

{

if (data[i + 1] < data[i])

{

swap(&data[i], &data[i + 1]);

SwapHappened = true;

AlgorithmTracker.SaveCurrentState(data, size);

}

}

end--;

if (!SwapHappened)

{

break;

}

SwapHappened = false;

for (int i = end-1; i >= 0; i--)

{

if (data[i + 1] < data[i])

{

swap(&data[i], &data[i + 1]);

SwapHappened = true;

AlgorithmTracker.SaveCurrentState(data, size);

}

}

begin++;

} while (SwapHappened);

}

void CombSort::Sort(char\* data, int size)

{

AlgorithmTracker.SaveCurrentState(data, size);

double decreaseFactor = 1.247;

int step = (int) (size / decreaseFactor);

while (step>1)

{

for (int i = 0; i < size - step; i++)

{

if (data[i +step] < data[i])

{

swap(&data[i], &data[i + step]);

AlgorithmTracker.SaveCurrentState(data, size);

}

}

step = (int) (step / decreaseFactor);

}

bool SwapHappened;

do {

SwapHappened = false;

for (int i = 0; i < size - 1; i++)

{

if (data[i + 1] < data[i])

{

swap(&data[i], &data[i + 1]);

AlgorithmTracker.SaveCurrentState(data, size);

SwapHappened = true;

}

}

} while (SwapHappened);

}

void TreeSort::Sort(char\* data, int size)

{

AlgorithmTracker.SaveCurrentState(data, size);

BinaryTree tree;

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

{

tree.AddNode(data[i]);

}

tree.Linearize(data);

AlgorithmTracker.SaveCurrentState(data, size);

}

void HeapSort::Sort(char\* data, int size)

{

AlgorithmTracker.SaveCurrentState(data, size);

for (int i = size/2; i >= 0; i--)

SiftDown(data,size, i, size-1);

int end = size-1;

while (end > 0) {

swap(&data[0], &data[end]);

AlgorithmTracker.SaveCurrentState(data, size);

SiftDown(data,size, 0, end-1);

end--;

}

}

void HeapSort::SiftDown(char\* data, int size, int begin, int end)

{

int current = begin;

while (current<=end/2)

{

if (current \* 2 + 2 <= end)

{

int maxi = current \* 2 + 1;

if (data[current \* 2 + 1] < data[2 \* current + 2])

{

maxi = 2 \* current + 2;

}

if ((data[current] < data[maxi]))

{

swap(&data[current], &data[maxi]);

current = maxi;

}

else { break; }

AlgorithmTracker.SaveCurrentState(data, size);

}

else if(current \* 2 + 1 <= end)

{

if ((data[current] < data[current \* 2 + 1]))

{

swap(&data[current], &data[current \* 2 + 1]);

current = current \* 2 + 1;

AlgorithmTracker.SaveCurrentState(data, size);

}

else break;

}

else {

break;

}

}

}

void MergeSort::Sort(char\* data, int size)

{

AlgorithmTracker.SaveCurrentState(data, size);

RecursiveSort(data, size, 0, size - 1);

}

void MergeSort::RecursiveSort(char\* data, int size, int begin, int end)

{

if (begin+1>end)

{

return;

}

int center = (begin + end) / 2;

RecursiveSort(data, size,begin, center);

RecursiveSort(data, size, center+1, end);

char\* result = new char[end - begin+1];

int current1 = begin;

int current2 = center + 1;

for (int i = 0; i < (end-begin+1); i++)

{

if (current1<=center && current2 <= end)

{

if (data[current1] < data[current2])

{

result[i] = data[current1];

current1++;

}

else

{

result[i] = data[current2];

current2++;

}

}

else

{

if (current1 > center)

{

result[i] = data[current2];

current2++;

}

else

{

result[i] = data[current1];

current1++;

}

}

}

for (int i = 0; i < (end - begin+1); i++)

{

data[begin+i] = result[i];

AlgorithmTracker.SaveCurrentState(data, size);

}

delete[] result;

}

Tracker.h

#pragma once

#include <ctime>

#include <vector>

using std::vector;

using std::clock\_t;

class Tracker

{

private:

clock\_t startTime;

clock\_t elapsedTime;

vector<char\*>\* AlgorithmTracker;

public:

bool Visualizing;

void StartTimer();

void PauseTimer();

void ClearTimer();

clock\_t GetTime();

void SaveCurrentState(char\* data, int size);

char\* GetState(int i);

void ClearStates();

int StatesCount();

Tracker();

~Tracker();

};

Tracker.cpp

#include "Tracker.h"

void Tracker::StartTimer()

{

startTime = clock();

}

void Tracker::PauseTimer()

{

clock\_t CurrentClock = clock();

elapsedTime += CurrentClock - startTime;

}

void Tracker::ClearTimer()

{

startTime = 0;

elapsedTime = 0;

}

clock\_t Tracker::GetTime()

{

return elapsedTime;

}

void Tracker::SaveCurrentState(char\* data, int size)

{

if (this->Visualizing==false)

{

return;

}

PauseTimer();

//Copying current array

char\* NewData = new char[size];

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

{

NewData[i] = data[i];

}

AlgorithmTracker->push\_back(NewData);

StartTimer();

}

char\* Tracker::GetState(int i)

{

return AlgorithmTracker->at(i);

}

void Tracker::ClearStates()

{

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

{

delete[] AlgorithmTracker->at(i);

}

AlgorithmTracker->clear();

}

int Tracker::StatesCount()

{

return AlgorithmTracker->size();

}

Tracker::Tracker()

{

AlgorithmTracker = new vector<char\*>;

Visualizing = true;

startTime = 0;

elapsedTime = 0;

}

Tracker::~Tracker()

{

ClearStates();

}

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Обозначение | | | | Наименование | | | | Дополнительные сведения | | | |
|  | | | | Текстовые документы | | | |  | | | |
|  | | | |  | | | |  | | | |
| БГУИР КП 1–40 01 01 ПЗ | | | | Пояснительная записка | | | | 49 с. | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  | | | |  | | | |  | | | |
|  |  |  |  |  | БГУИР КП 1-40 01 01  Д1 | | | | | | |
|  |  |  |  |  |
|  |  |  |  |  |
| Изм. | Л. | № докум. | Подп. | Дата | Анализ и сравнение алгоритмов сортировки Ведомость курсового  проекта |  | | | | Лист | Листов |
| Разраб. | | Киселев Н.И. |  | 23.12.19 |  |  | |  | 49 | 49 |
| Пров. | | Шостак Е.В. |  | 23.12.19 | Кафедра ПОИТ | | | | | |
|  | |  |  |  |