МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Сортировка слабой кучей

Студент гр. 9383 _____ Корсунов А.А.

Преподаватель _____ Попова Е.В.

Санкт-Петербург

2020

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Корсунов А.А.

Группа 9383

Тема работы: Сортировка слабой кучей

Исходные данные:
Сгенерированный массив данных. Сортировка слабой кучей, сравнительное исследование с быстрой сортировкой.

Содержание пояснительной записки: «Содержание», «Введение». «Формальная постановка задачи», «Описание алгоритма», «Описание структур данных и функций», «Тестирование», «Исследование», «Заключение», «Список использованных источников».

Предполагаемый объем пояснительной записки:
Не менее страниц.

Дата выдачи задания: 04.11.2020

Дата сдачи реферата: 22.12.2020

Дата защиты реферата: 25.12.2020

Студент _____ Корсунов А.А.

Преподаватель _____ Попова Е.В.

АННОТАЦИЯ

В данной работе реализовано сравнительное исследование сортировки слабой кучей с быстрой сортировкой. Программа написана на языке программирования C++. Корректность сортировки проверяется с помощью стандартной сортировки языка C++. Метод сравнения сортировок — замер времени выполнения алгоритма. Представлена возможность просмотра промежуточных данных с целью убеждения в корректности алгоритмов сортировок.

SUMMARY

In this paper, a comparative study of weak heap sorting with fast sorting is implemented. The program is written in the C++programming language. The correct sorting is checked using the standard C++sorting language. The method of comparison of sorts is the measurement of the algorithm execution time. The possibility of viewing intermediate data in order to convince the correctness of sorting algorithms is presented.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Работа была посвящена такой структуре данных, как слабая куча. Главным алгоритмом является сортировкой слабой кучей.

Целью работы является исследование сортировки слабой кучей и сравнением с быстрой сортировкой.

Для реализации данной цели необходимо решить следующие задачи:

1. Собрать теоретические сведения по изучаемой структуре данных и реализации сортировки слабой кучей.
2. Написать программу на языке C++, реализующая сортировку слабой кучей.
3. Провести исследование для сортировки слабой кучей и быстрой сортировки.

1. ФОРМАЛЬНАЯ ПОСТАНОВКА ЗАДАЧИ

Вариант 32. Сортировка слабой кучей. Сравнительное исследование с другим алгоритмом сортировки. Сравнивать следует с алгоритмом сортировки, реализованным или частично реализованным в ЛР4 (а именно, с быстрой сортировкой).

Исследование должно содержать:

1. Анализ задачи, цели, технологию проведения и план экспериментального исследования.
2. Генерацию представительного множества реализаций входных данных (с заданными особенностями распределения (для среднего и для худшего случаев)).
3. Выполнение исследуемых алгоритмов на сгенерированных наборах данных. При этом в ходе вычислительного процесса фиксируется как характеристики (например, время) работы программы, так и количество произведенных базовых операций алгоритма.
4. Фиксацию результатов испытаний, их интерпретацию и сопоставление с теоретическими оценками.

2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ ПОЛОЖЕНИЯ

2.1. Слабая куча.

Куча — это специальная форма данных по типу дерева, удовлетворяющая условиям кучи - если B есть узел-потомок узла A, то ключ(A) ≥ ключ(B).

Слабая куча (структура данных) — куча по типу бинарного полного дерева, у которой только один из потомков удовлетворяет условию кучи (правый потомок), левый же потомок может быть как меньше, так и больше родителя. У корневого узла кучи нет левого потомка, потому как корневой узел должен быть максимальным, что исключает возможность левого потомка такого узла быть больше своего родителя.

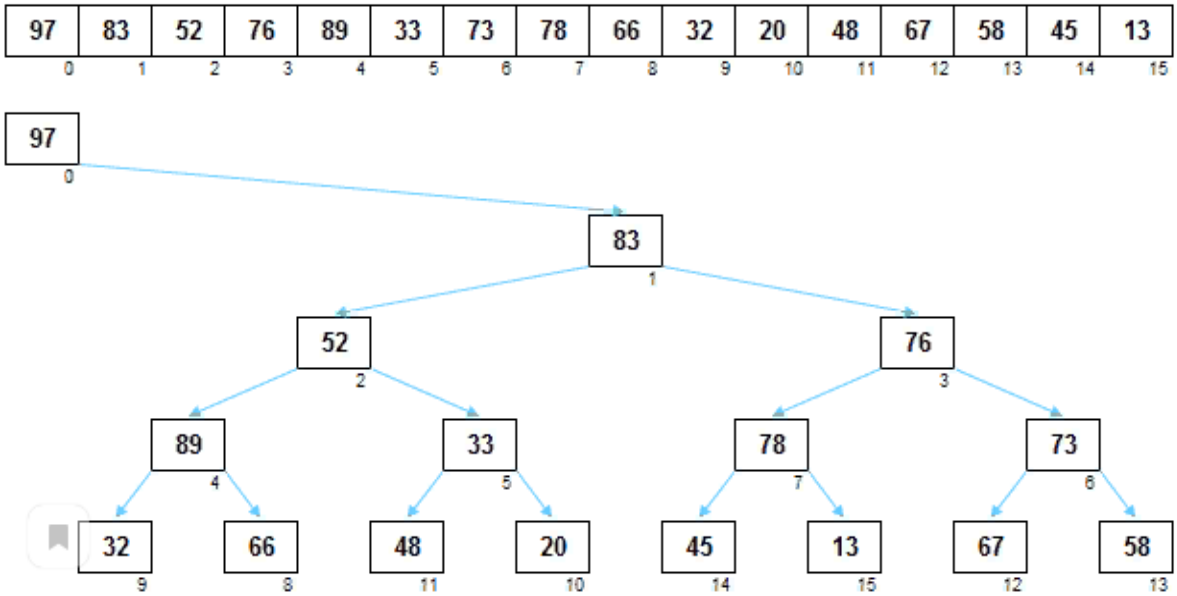

Рисунок 1 — Графическое представление слабой кучи для заданного массива

2.2. Бинарное дерево.

Бинарное дерево — это дерево, в котором для каждый узел-родитель содержит не более двух сыновей (левого и правого)(на рисунке 1 изображено бинарное дерево).

2.3. Полное дерево.

Полное N-дерево — это дерево, у которого полностью заполнены все уровни (у каждого узла N потомков), кроме последнего, а для предпоследнего уровня выполняется условие, что у узла, расположенного в дереве левее количество детей больше или равно количеству детей у узла, расположенного в дереве правее (на рисунке 1 изображено полное дерево).

3. АЛГОРИТМЫ

3.1. Алгоритм сортировки слабой кучей.

Алгоритм сортировки слабой кучей делится на два этапа:

1) Формируем из массива слабую кучу:

а) Перебираем элементы массива с конца до начала;

б) Для текущего элемента поднимаемся вверх по родительской ветке до ближайшего «правого» родителя;

в) Сравниваем текущий элемент и ближайшего правого родителя;

г) Если ближайший правый родитель меньше текущего элемента, то меняем местами (левый <> правый) поддеревья с потомками для узла, в котором находится текущий элемент, а также меняем значениями ближайший «правый» родитель и узел с текущим элементом.

2) Из корня кучи текущий максимальный элемент перемещается в конец неотсортированной части массива:

а) В корне кучи находится текущий максимальный элемент для неотсортированной части массива;

б) Меняем местами максимум из корня кучи и последний элемент в неотсортированной части массива. Последний элемент с максимумом перестает быть узлом слабой кучи;

в) Опускаемся из корня кучи по левым потомкам как можно ниже;

г) Поднимаемся по левым потомкам обратно к корню кучи, сравнивая каждый левый потомок с корнем;

д) Если элемент в корне меньше, чем очередной левый потомок, то меняем местами (левый <> правый) поддеревья с потомками для узла, в котором находится текущий левый потомок, а также меняем значениями корень кучи и узел с текущим левым потомком;

е) В корне слабой кучи снова находится максимальный элемент для оставшейся неотсортированной части массива. Возвращаемся в

пункт 2) и повторяем процесс, пока не будут отсортированы все элементы.

Пояснения к алгоритму:

*«Правый» родитель для правого потомка — его непосредственный родитель, для левого потомка — прародитель, который находится в правом поддереве;

*Согласно описанию структуры (страница 7) для определения правого и левого потомков используется битовый массив, *Битовый массив* — массив, состоящий из нулей и единиц, ноль показывает, что для родителя по i-му индексу обмена потомками не производилось, а единица — что обмен потомками был произведен, для того, чтобы получить левого потомка родителя используется формула — 2*<индекс родителя>+<элемент в массиве битов по индексу родителя> , а для правого потомка — 2*<индекс родителя>+1-<элемент в массиве битов по индексу родителя>, а для того, чтобы получить родителя текущего потомка достаточно использовать формулу [<индекс потомка>/2], ([] - целая часть);

Алгоритм сортировки слабой кучей имеет одинаковую сложность для лучшего, среднего и худшего случая — O(nlogn), однако чем больше в слабой куче одинаковых значений, тем быстрее работает сортировка.

3.2. Алгоритм быстрой сортировки.

1) Выбираем опорный элемент в неотсортированной части массива (в данном алгоритме опорным элементом всегда выбирается первый элемент в еще неотсортированной части массива);
2) Проходим по неотсортированной части массива, сравнивая текущий опорный элемент с очередным элементом из массива, перемещая элементы, меньшие опорного, слева от него, а элементы, большие опорного, справа от него;
3) Таким образом массив делится на две части (элементы, меньшие опорного и большие опорного);

4) Для обеих частей повторяем шаги 1-3 пока длина части не станет равна единице;

3.3. Сравнение алгоритмов.

Общие черты:

*У обоих алгоритмов в сортировке идет сравнение неотсортированных элементов с каким-то определенным элементом (в сортировке слабой кучей — с текущим максимальным элементом в неотсортированной части массива, в быстрой сортировке — с текущим опорным элементом);

Различия:

*Для сортировки слабой кучей массив всегда должен представлять из себя слабую кучу (то есть, изначально из массива нужно сделать слабую кучу, более того, в процессе сортировки неотсортированная часть массива перестает быть слабой кучей до того момента, пока текущий первый элемент (корневой узел) в неотсортированном массиве не будет являться максимальным), когда для быстрой сортировки массив не обязан представлять из себя какую-либо структуру данных;

*В ходе быстрой сортировки массив на каждом шаге условно делится на две части (элементы, меньшие текущего опорного элемента, и элементы, большие текущего опорного элемента), когда как в ходе сортировки слабой кучей неостортированная часть массива уменьшается на единицу на каждом шаге;

4. Сложности алгоритмов и ожидаемые результаты исследований

Быстрая сортировка: операция разделения массива на две части относительно опорного элемента занимает время O(log(2)n) (2 в основании). Поскольку все операции разделения, проделываемые на одной глубине рекурсии, обрабатывают разные части исходного массива, размер которого постоянен, суммарно на каждом уровне рекурсии потребуется также O(n) операций. Следовательно, общая сложность алгоритмов определяется лишь количеством разделений. Количество разделений в свою очередь, зависит от сочетания входных данных и способа определения опорного элемента.

Далее пойдут термины лучшие и худшие случаи, что требует пояснения; Лучший случай — сочетание входных данных, при которых сложность алгоритма по времени стремится к минимально возможной, худший, соответственно, к максимально возможной.

Лучший случай для быстрой сортировки: при каждой операции разделения массив делится на две одинаковые (плюс-минус один элемент) части, что дает общую сложность O(nlog(2)n);

Худший случай для быстрой сортировки: при каждой операции разделение дает два подмассива размерами 1 и n-1, то есть при каждом рекурсивном вызове больший массив будет на 1 короче, чем предыдущий. Такое может произойти, если в качестве опорного на каждом этапе будет выбран элемент либо наименьший, либо наибольший из всех обрабатываемых. Общая сложность такого разбиения составит O(n^2);

Сортировка слабой кучей: вначале алгоритма поданный на вход массив элементов преобразуется в слабую кучу (проход по элементом с конца до начала, меняя в отдельных случаях поддеревья местами (за счет битового массива) и сортируюя правых родителей их потомков от большего к меньшему), что занимает O(n) операций; «Удаление» из корня максимального элемента и проход по левым потомка от начала до конца и от конца до начала занимает O(logn) операций. Таким образом формируется сложность O(nlogn)

алгоритма по времени. В силу того, что сортировка всегда требует проход сначала по всем элементам в ходе формирования первоначальной слабой кучи, а после чего проход по левым потомка, то увеличить или уменьшить скорость сортировки можно только на этапе сравнения элементов друг с другом (чем их меньше, тем быстрее будет работать сортировка), отчего лучший случай будет выглядеть, как массив одинаковых элементов, а худший случай — массив отличных от друг друга элементов. Но даже так разница во времени работы будет минимальной и во всех вариантах будет заниматься O(nlogn) операций.

Ожидаемые результаты: так как сложность сортировки слабой кучей даже в худшем случае будет O(nlogn), когда как для быстрой сортировки сложность может достигать O(n^2), то на случайном наборе данных, в среднем, алгоритм сортировки слабой кучей будет работать быстрее, но в случае лучших вариантов для каждой сортировки, быстрая куча всегда будет работать быстрее.

5. ФУНКЦИИ ПРОГРАММЫ

5.1. Файл Weak_Heap.cpp

В данном файле реализован класс Weak_Heap:

Класс содержит следующие поля:

- **vector<int> bit (приватное)** — хранит массив битов
- **int len (публичное)** — хранит длину массива, который нужно отсортировать
- **vector<int> wheap (публичное)** — хранит массив, который нужно отсортировать

И следующие методы:

- **void merge_weak_heap(int, int) (приватное)** – сравнивает первый поданный аргумент (индексы элементов из массива) со вторым, если первый элемент больше, индекс левого потомка меньше длины массива и первый индекс не равен нулю, то меняет значение в битовом массиве (на ноль, если там была единица, на единицу, если там был ноль) и меняет значения по этим индексам местами;
- **void print_leadge_list(int, string, int) (публичное)** – выводит слабую кучу в виде уступчатого списка, метод является рекурсивным, аргументами выступают индекс текущего элемента, строка, которая увеличивается (к ней добавляется строка «--» и длину слабой кучи) ;
- **void read(vector<int>) (публичное)** - записывает элементы переданного вектора в поле wheap (предварительно чистит этот вектор);
- **void set_len(int) (публичное)** – записывает переданную длину массива в поле len;
- **void print_wheap() (публичное)** - выводит значения вектора wheap в консоль;
- **void print_bit() (публичное) -** выводит значения вектора bit в консоль;
- **void sort_weak() (публичное) -** сортировка слабой кучей путем вызовов двух методов (make_weak_heap() и weak_sow(len-1));

- **void make_weak_heap() (публичное)** — строит из массива первоначальную слабую кучу. В цикле идем от конца массива до начала, для каждого текущего элемента и «правого» родителя вызываем метод merge_weak_heap(int, int);
- **void weak_sow(int, int) (публичное)** — сортирует слабую кучу. Метод рекурсивный, аргументами выступают флаг, который определяет, нужно ли выводить текущую слабую кучу в виде уступчатогосписка и индекс последнего еще неотсортированного элемента, в цикле идем по левым потомком, пока не встретим последнего, идем назад, каждый раз вызывая метод merge_weak_heap(int, int) для текущего и первого элемента. В конце вызываем еще раз этот метод для первого и второго элементов, если осталось всего 2 элемента или вызываем метод weak_sow(int) передавая индекс следующего последнего элемента;

5.2. Файл Quick_Sort.cpp

В данном файле реализован класс Quick_Sort:

Класс содержит следующие поля:

- **int len (приватное)** — хранит длину массива;
- **vector<int> qheap (публичное)** — хранит массив, который нужно отсортировать;

И следующие методы:

- **int get_len() (публичное)** — возвращает длину массива;
- **void set_len(int) (публичное)** — записывает переданную длину массива в поле len;
- **void read(vector<int>) (публичное)** — записывает переданный массив в поле qheap;
- **void qsort(int, int, int) (публичное)** — сортирует переданный массив. Данный метод рекурсивный. Он принимает на вход индекс начального элемента, индекс конечного элемента и флаг (который отвечает за вывод массива в виде уступчатого списка). После чего записываются опорный

элемент, его индекс, индексы конца (вспомогательные индексы для прохода по текущей стороне и вывода вектора в консоль, если флаг установлен в единицу. В цикле производится обход по текущей стороне массива, где все элементы, меньшие опорного элемента, записываются слева от него, а элементы, большие опорного элемента, записываются в конце вектора.

- **void print(int, int, int, int)** – выводит либо весь вектор целиком (если в параметр icount был передан ноль) либо выводит изменения текущей стороны вектора;
- **void print_sorted()** - выводит отсортированный массив в консоль;

5.3. Файл main.cpp:

- **main()** - в главной функции создаются элементы классов, описанных выше и вызывается функция panel(), в которую передаются указатели на созданные элементы;
- **panel(Weak_Heap* arr1, Quick_Sort* arr2)** – функция для выбора задачи. В начале выводится справка с предлагаемыми действиями.

6. ТЕСТИРОВАНИЕ И ОПИСАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

6.1. Описание интерфейса пользователя.

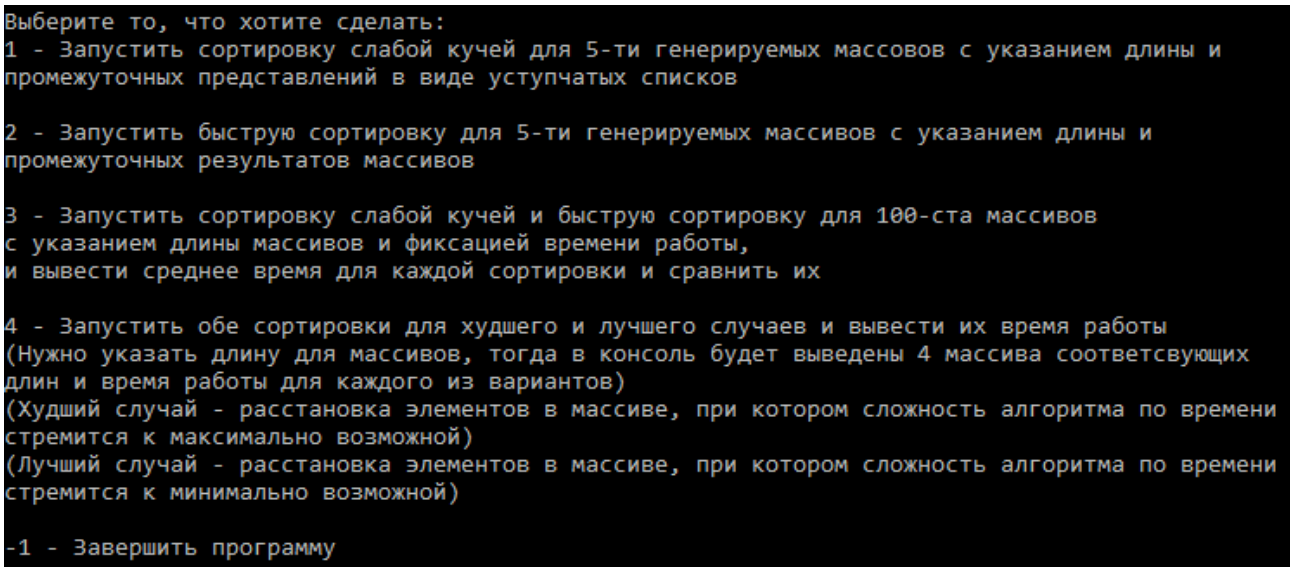

Рисунок 2 — Интерфейс пользователя

На рисунке 2 изображен интерфейс пользователя для взаимодействия с программой. Взаимодействие происходит через консоль, вначале и после каждого взаимодействия выводится небольшая справка с выбором возможных вариантов:

- Если ввести «-1», то программа завершиться;
- Если ввести «2», то в циклах создаются 5 массивов (значения от 0 до 100), в консоль выводятся промежуточные данные в ходе сортировки слабой кучей;
- Если ввести «3», то в циклах создаются 5 массивов (значения от 0 до 100), в консоль выводятся промежуточные данные в ходе быстрой сортировки;
- Если ввести «4», то в циклах создаются 100 массивов (диапазон значений от 0 до 10000), пользователь вводит длину массива, после чего происходит сортировка слабой кучей и быстрой сортировкой с промежуточным временем работы для каждой из них. В конце вычисляется среднее время работы каждой сортировки, сравнивается и выводится результат сравнения;

- Если ввести «5», пользователь вводит длину массивов для худших и лучший случаев сортировки (подробнее указано на рисунке 2), после чего выводится их время работы;

6.2. Наглядное представление промежуточных результатов сортировки

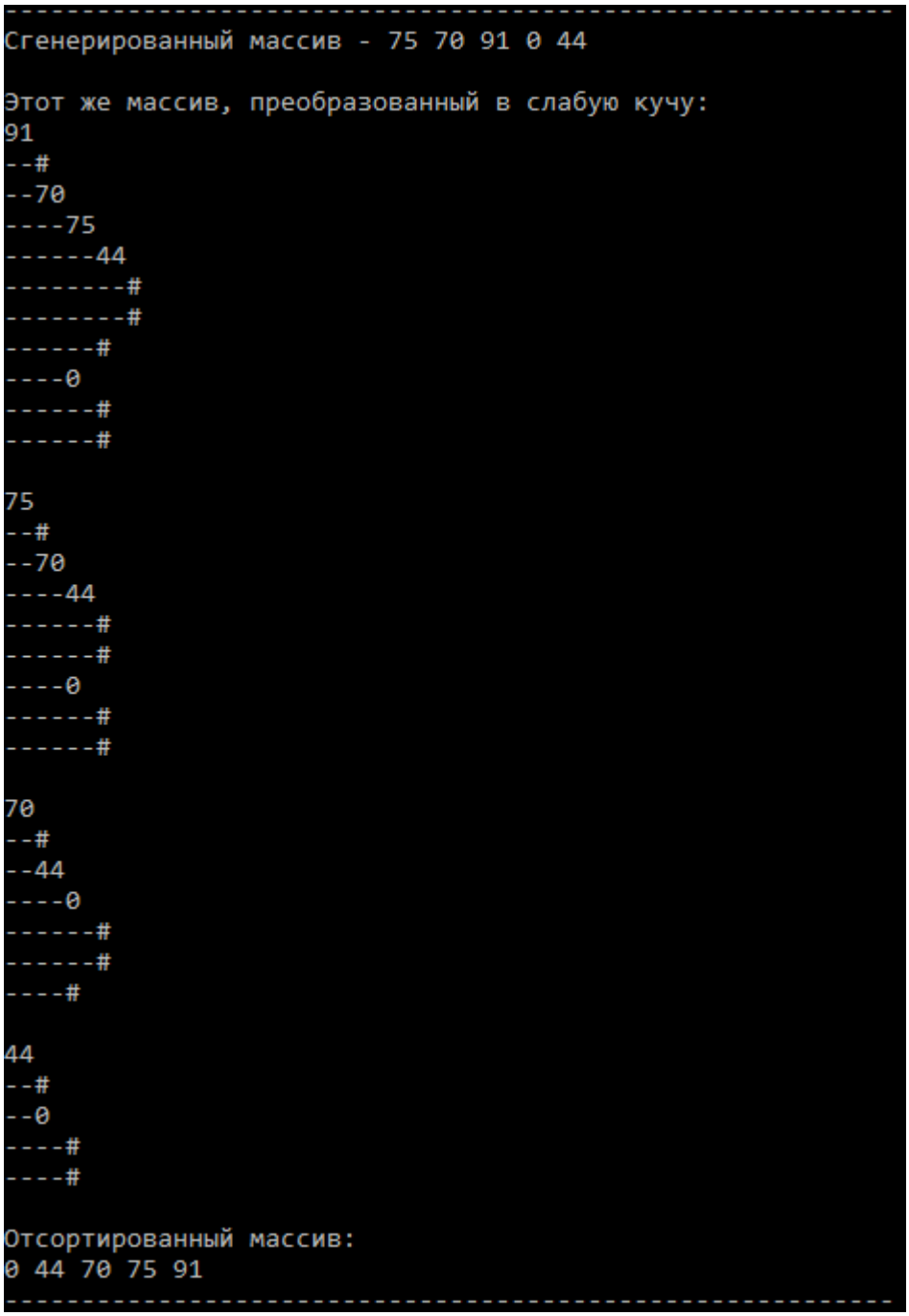

Рисунок 3 — Представление промежуточных результатов сортировки - слабой кучи в виде уступчатого списка

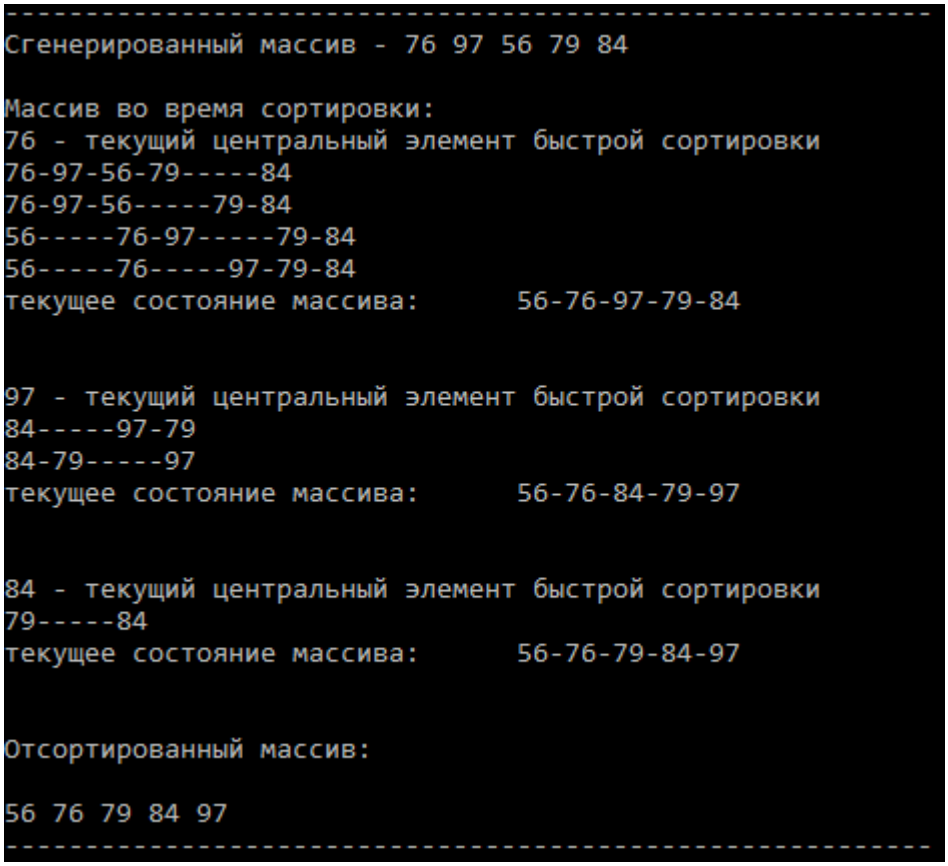

Рисунок 4 — Представление промежуточных результатов быстрой сортировки

6.3. Сравнение времени работы каждой сортировки.

```
N               Сортировка слабой кучей         Быстая сортировка
0               0.000968354                     0.00542541
1               0.000666848                     0.00604552
2               0.000650889                     0.00561691
3               0.000649179                     0.00651687
4               0.000685657                     0.00639148
5               0.000665708                     0.00631511
6               0.000659438                     0.0059435
7               0.000652599                     0.00544365
8               0.000657158                     0.00609681
9               0.000650319                     0.00493866
10              0.000660009                     0.00490846
```

```
        90              0.000673117                     0.00657159
        91              0.000739803                     0.00587054
        92              0.000662288                     0.00639148
        93              0.000755191                     0.00732792
        94              0.000742082                     0.00661262
        95              0.000677677                     0.00668843
        96              0.000751772                     0.00708056
        97              0.000754051                     0.0067876
        98              0.000675397                     0.00599479
        99              0.000740942                     0.00605236

Среднее время сортировки слабой кучей: 0.000724317
Среднее время быстрой сортировки: 0.00658161

Сортировка слабой кучей в среднем быстрее

------------------------------------------------------------
```

Рисунок 5 — Сравнение среднего времени работы (длина каждого массива — 1000, элементы генерируются от нуля до 10000)

6.4. Фиксация наилучших и наихудших вариантов для каждой сортировки

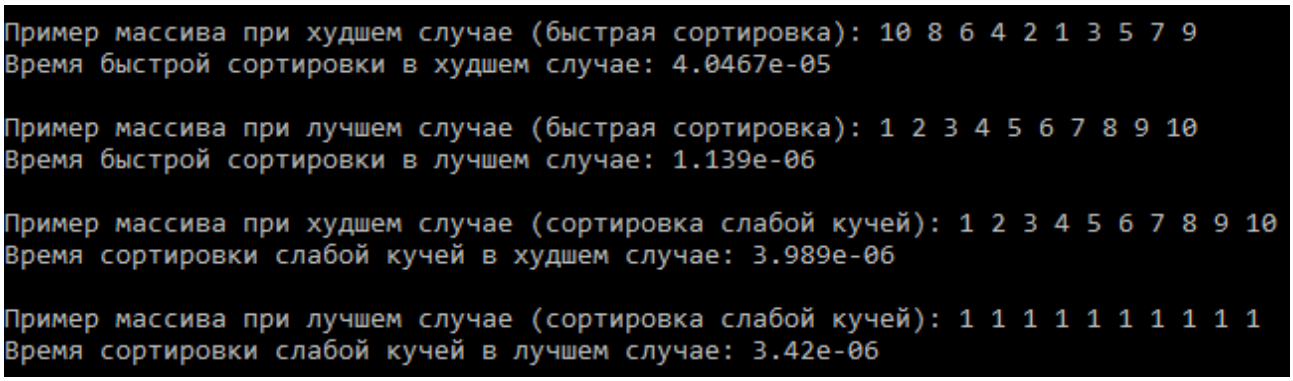

Рисунок 6 — Фиксация времени работы сортировок в худших и лучших случаев (под худшими и лучшими случаями подразумеваются соответствующие расстановки элементов, при которых сложность алгоритмов по времени стремится к соответственно максимальной и минимальной)

7. ИССЛЕДОВАНИЕ

7.1. Цель исследования

Задачей исследования является изучение эффективности (время работы алгоритма) сортировки слабой кучей. Сравнение скорости сортировки слабой кучей с быстрой сортировкой. Сопоставление с теоретическими данными.

7.2. План и технология исследования

- Сравнить сортировки слабой кучей на генерируемых массивах для разных N;
- Сравнить скорости сортировки слабой кучей для худшего и лучшего случаев;

7.3. Генерация данных с заданными особенностями распределения

В силу того, что сортировка слабой кучей имеет одну и ту же сложность в любом случае (O(nlogn)), данные будут генерироваться случайным образом. Время работы представлено в секундах.

7.4. Выполнение исследуемых алгоритмов и накопление статистики

- **Сравнение сортировки слабой кучей на генерируемых массивах для разных длин:**

  Количество элементов — длина каждого массива;

  Число массивов — число создаваемых массивов для сортировок;

  Генерация чисел — выбор случайного числа в заданном диапозоне;

  Результаты испытаний продемонстрированы ниже;

  1) 1-ое испытание: 1000 элементов, 100 массивов, генерация чисел от 0 до 1000;
  2) 2-ое испытание: 20000 элементов, 100 массивов, генерация чисел от 0 до 1000;

3) 3-тье испытание: 100 элементов, 1000 массивов, генерация чисел от 0 до 1000;
4) 4-ое испытание: 100 элементов, 100 массивов, генерация чисел от 0 до 10000;
5) 5-ое испытание: 1000 элементов, 100 массивов, генерация чисел от 0 до 10000;

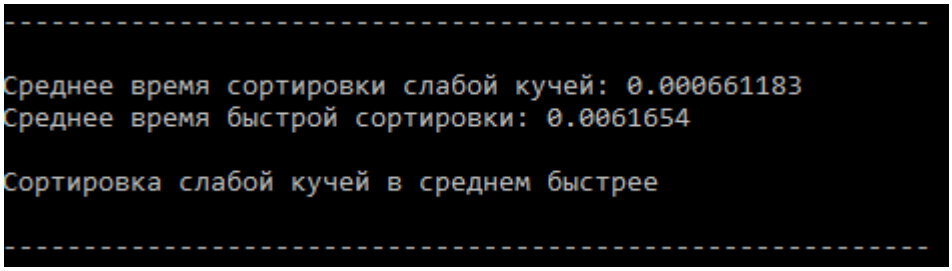

Рисунок 7 — Первое испытание

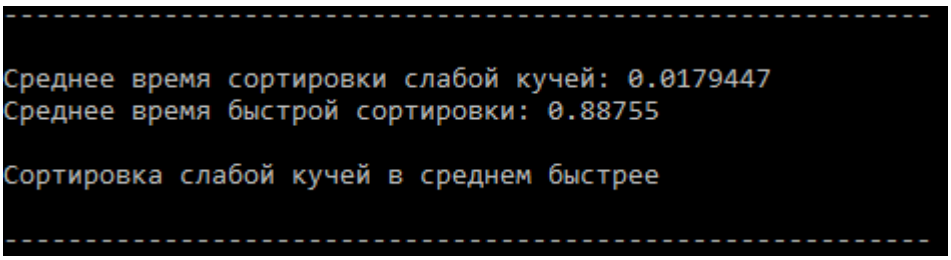

Рисунок 8 — Второе испытание

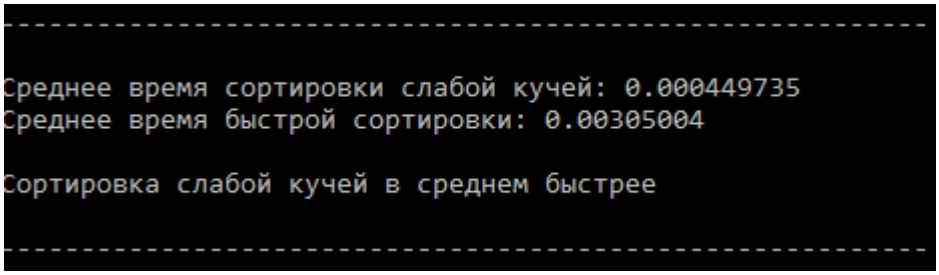

Рисунок 9 — Третье испытание

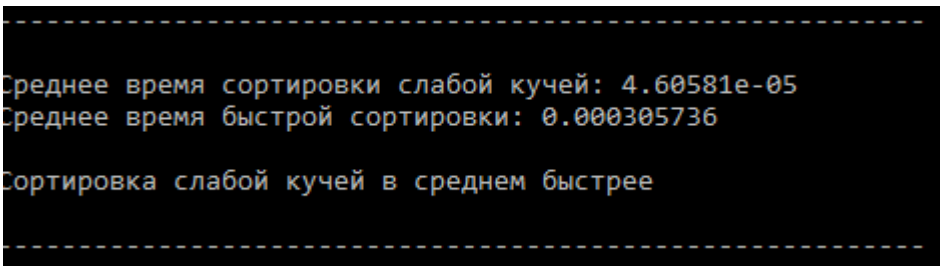

Рисунок 10 — четвертое испытание

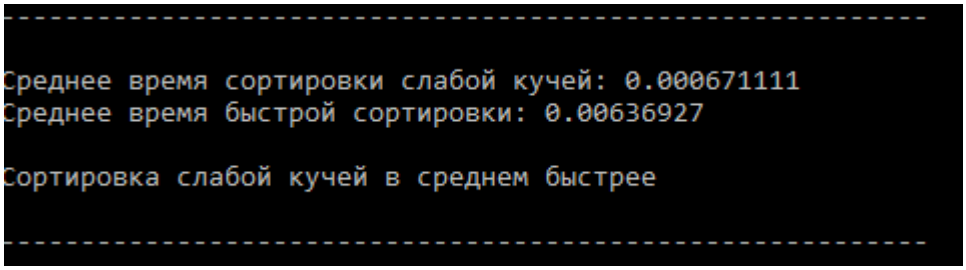

Рисунок 11 — пятое испытание

- **Сравнение скорости сортировки слабой кучей для худшего и лучшего случаев:**
- Лучший случай для сортировки слабой кучей — все элементы одинаковы
- Худший случай для сортировки слабой кучей — все элементы разные
- Лучший случай для быстрой сортировки — отсортированных массив (массив делится на примерно равные доли после каждого прохода)
- Худший случай для быстрой сортировки — массив делится на части 1 и n-1 (n-длина массива)

  Результаты испытаний расположены ниже;

  1) 1-ое испытание: длина массивов 10;
  2) 2-ое испытание: длина массивов 15;
  3) 3-тье испытание: длина массивов 5;
  4) 4-ое испытание: длина массивов 20;
  5) 5-ое испытание: длина массивов 3;

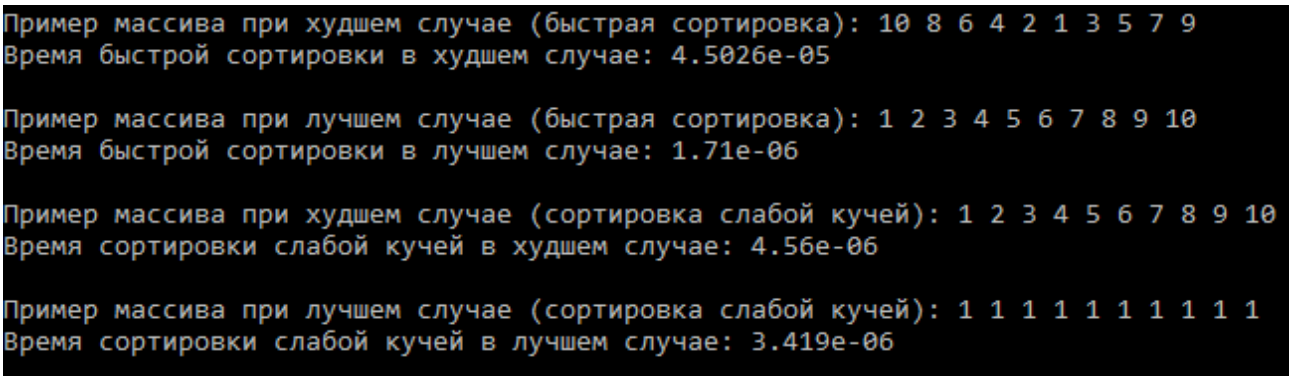

Рисунок 12 — Первое испытание

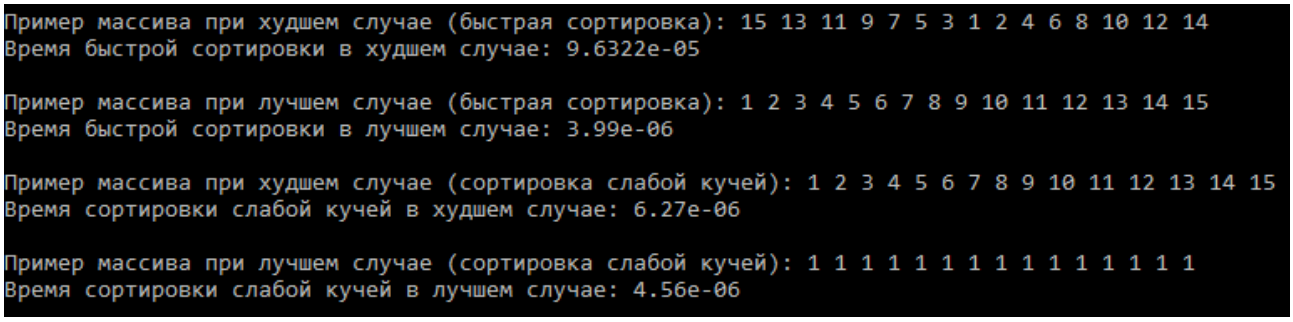

Рисунок 13 — Второе испытание

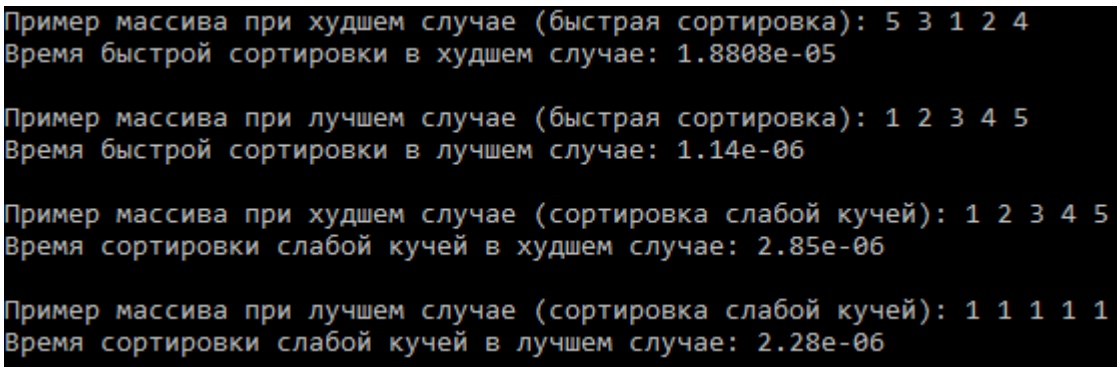

Рисунок 14 — Третье испытание

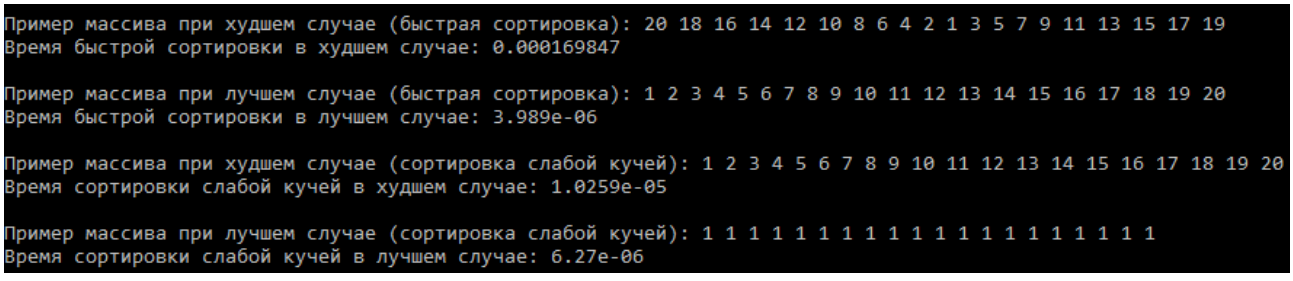

Рисунок 15 — Четвертое испытание

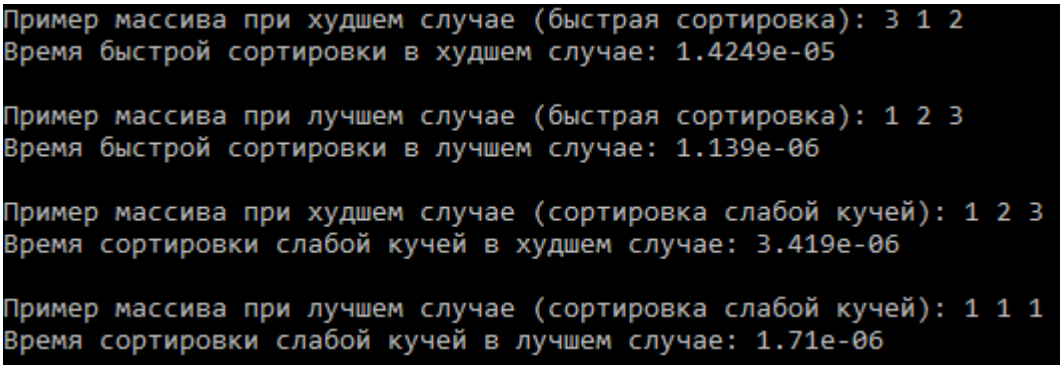

Рисунок 16 — Пятое испытание

7.5. Интерпретация результатов испытаний и сопоставление с теоретическими оценками:

- **Сравнение сортировки слабой кучей на генерируемых массивах для разных длин:**

(Сопоставимо теоретическим данным) Нетрудно заметить, что в среднем сортировка слабой куче на случайно сгенерированных данных работает быстрее, причем, чем больше выборка (больший диапазон), тем лучше это заметно. Объясняется это тем, что у сортировки слабой кучей сложность в любом из случаев O(nlogn) (за счет того, что при самой сортировки удаление максимального элемента с промежуточными обменами поддеревьями между потомками требует O(nlogn) операций), а быстрая сортировка в худшем случае может иметь сложность O(n^2) (за счет того, что центральным элементом в сгенерированном массиве может выступать максимальный или минимальный из еще неотсортированной части);

- **Сравнение скорости сортировки слабой кучей для худшего и лучшего случаев:**

(Сопоставимо теоретическим данным) Ожидаемо, в лучшем случае быстрая сортировка работает быстрее (за счет того, что в лучшем случае массив делится на две примерно равные части), что естественно больше быстрой сортировки в худшем случае, при которой массив делится на части 1 и n-1. Для сортировки слабой кучей лучший и худший вариант примерно равны, лучший

вариант немного быстрее за счет того, что приходится делать меньше обменов между родителями и потомками;

8. ЗАКЛЮЧЕНИЕ

Поставленная задача, а именно исследование сортировки слабой кучей и ее сравнение с быстрой сортировкой, была достигнута. Для достижения этой задачи были решены следующие задачи:

- Собраны теоретические сведения по структуре данных — слабая куча и реализации сортировки слабой кучей;
- Написана программа на языке C++, реализующая сортировку слабой кучей, а также было проведено исследование сортировки слабой кучей в сравнении с быстрой сортировкой;
- Получившиеся результаты исследование:

а) В среднем сортировка слабой кучей работает быстрее, чем быстрая сортировка;

б) При лучшем варианте быстрая сортировка работает быстрее, чем сортировка слабой кучей при лучшем варианте;

в) Сортировка слабой кучей в лучшем и худшем варианте примерно равны;

9. СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Habr.com URL: https://habr.com/ru/company/edison/blog/499786/ (Дата обращения 22.12.2020)
2. GitHub.com URL: https://github.com/notepad104/algorithms/blob/master/WeakHeap.cpp (Дата обращения 20.12.2020)
3. Wikipedia.org URL: https://ru.wikipedia.org/wiki/Пирамидальная_сортировка (Дата обращения 22.12.2020)
4. sortings.github.io URL: https://sortings.github.io/sort_types/quick.html (Дата обращения 22.12.2020)

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp:

```
#include <iostream>
#include <windows.h>
#include <vector>
#include "Weak_Heap.h"
#include "Quick_Sort.h"
#include <random>
#include <ctime>
#include <chrono>
#include <time.h>
#include <algorithm>

using namespace std;

void panel(Weak_Heap* arr1, Quick_Sort* arr2)
{
   int key = 0;
   while (key != -1)
   {
      cout << "\nВыберите то, что хотите сделать:\n";
        cout << "1 - Запустить сортировку слабой кучей для 5-ти генерируемых массовов с указанием длины и\n";
      cout << "промежуточных представлений в виде уступчатых списков\n\n";
          cout << "2 - Запустить быструю сортировку для 5-ти генерируемых массивов с указанием длины и\n";
      cout << "промежуточных результатов массивов\n\n";
```

```
cout << "3 - Запустить сортировку слабой кучей и быструю сортировку для 100-ста массивов\n";
cout << "с указанием длины массивов и фиксацией времени работы,\n";
cout << "и вывести среднее время для каждой сортировки и сравнить их\n\n";
cout << "4 - Запустить обе сортировки для худшего и лучшего случаев и вывести их время работы\n";
cout << "(Нужно указать длину для массивов, тогда в консоль будет выведены 4 массива соответсвующих\n";
cout << "длин и время работы для каждого из вариантов)\n";
cout << "(Худший случай - расстановка элементов в массиве, при котором сложность алгоритма по времени\n";
cout << "стремится к максимально возможной)\n";
cout << "(Лучший случай - расстановка элементов в массиве, при котором сложность алгоритма по времени\n";
cout << "стремится к минимально возможной)\n\n";
cout << "-1 - Завершить программу\n\n";
cin >> key;
if (key == -1)
{
    return;
}
if (key == 1)
{
    int len;
    cout << "Введите длину для массивов\n\n";
    cin >> len;
    for (int i = 0; i < 5; i++)
    {
        vector<int> arr;
```

```
        for (int j = 0; j < len; j++)
        {
            arr.push_back(rand()%101);
        }
        arr1->set_len(len);
        arr1->read(arr);
        arr1->make_weak_heap();
        cout << "-----------------------------------------------------\n";
        cout << "Сгенерированный массив - ";
        for (int i = 0; i < len; i++)
        {
            cout << arr[i] << " ";
        }
        cout << "\n\n";
        cout << "Этот же массив, преобразованный в слабую кучу:\n";
        arr1->weak_sow(arr1->len-1, 1);
        cout << "\nОтсортированный массив:";
        arr1->print_wheap();
        cout << "-----------------------------------------------------\n";
    }
}
if (key == 2)
{
    int len;
    cout << "Введите длину для массивов\n\n";
    cin >> len;
    for (int i = 0; i < 5; i++)
    {
        vector<int> arr;
        for (int j = 0; j < len; j++)
```

```
        {
            arr.push_back(rand()%101);
        }
        arr2->set_len(len);
        arr2->read(arr);
        cout << "-----------------------------------------------------\n";
        cout << "Сгенерированный массив - ";
        for (int i = 0; i < len; i++)
        {
            cout << arr[i] << " ";
        }
        cout << "\n\n";
        cout << "Массив во время сортировки:\n";
        arr2->qsort(0, arr2->get_len()-1, 1);
        cout << "Отсортированный массив:\n";
        arr2->print_sorted();
        cout << "-----------------------------------------------------\n";
    }
}
if (key == 3)
{
    double temp;
    double Avg_Weak_Sort = 0;
    double Avg_Quick_Sort = 0;
    int len;
    cout << "Введите длину для массивов\n\n";
    cin >> len;
    cout << "-----------------------------------------------------\n";
    cout << "\n\tN\t\tСортировка слабой кучей\t\tБыстая сортировка";
    for (int i = 0; i < 100; i++)
```

```
        {
            vector<int> arr;
            for (int j = 0; j < len; j++)
            {
                arr.push_back(rand()%10001);
            }
            arr1->set_len(len);
            arr1->read(arr);
            arr2->set_len(len);
            arr2->read(arr);
            auto start = std::chrono::steady_clock::now();
            arr1->sort_weak();
            auto end = std::chrono::steady_clock::now();
            temp = chrono::duration<double>(end-start).count();
            Avg_Weak_Sort += temp;
            start = std::chrono::steady_clock::now();
            arr2->qsort(0, arr2->get_len()-1, 0);
            end = std::chrono::steady_clock::now();
            Avg_Quick_Sort += chrono::duration<double>(end-start).count();
            cout << "\n\t" << i << "\t\t" << temp << "\t\t\t";
            cout << chrono::duration<double>(end-start).count();
        }
                cout << "\n\nСреднее время сортировки слабой кучей: " <<
Avg_Weak_Sort/100 << "\n";
          cout << "Среднее время быстрой сортировки: " << Avg_Quick_Sort/100
<< "\n";
        if (Avg_Weak_Sort < Avg_Quick_Sort)
        {
            cout << "\nСортировка слабой кучей в среднем быстрее\n";
        }
```

```
    else if (Avg_Quick_Sort > Avg_Weak_Sort)
    {
        cout << "\nБыстрая сортировка в среднем быстрее\n";
    }
    else
    {
        cout << "\nСортировки равны\n";
    }
    cout << "\n-------------------------------------------------------\n";
    cout << "\n";
}
if (key == 4)
{
    int len;
    cout << "Введите длину для массивов\n\n";
    cin >> len;
    vector<int> arr;
    int i = len;
    while (i > 0)
    {
        arr.push_back(i);
        i -= 2;
    }
    if (len % 2 == 1)
    {
        i++;
        i++;
    }
    i++;
    while (i < len)
```

```
        {
            arr.push_back(i);
            i += 2;
        }
        arr2->set_len(len);
        arr2->read(arr);
        cout << "\nПример массива при худшем случае (быстрая сортировка): ";
        for (int j = 0; j < len; j++)
        {
            cout << arr[j] << " ";
        }
        auto start = std::chrono::steady_clock::now();
        arr2->qsort(0, arr2->get_len()-1, 0);
        auto end = std::chrono::steady_clock::now();
                cout << "\nВремя быстрой сортировки в худшем случае: " <<
chrono::duration<double>(end-start).count() << "\n";
        arr.clear();
        for (int j = 1; j < len+1; j++)
        {
            arr.push_back(j);
        }
        arr2->read(arr);
        cout << "\nПример массива при лучшем случае (быстрая сортировка): ";
        for(int j = 0; j < len; j++)
        {
            cout << arr[j] << " ";
        }
        start = std::chrono::steady_clock::now();
        arr2->qsort(0, arr2->get_len()-1, 0);
        end = std::chrono::steady_clock::now();
```

```
            cout << "\nВремя быстрой сортировки в лучшем случае: " << chrono::duration<double>(end-start).count() << "\n";
        arr.clear();
        for (int j = 1; j < len+1; j++)
        {
           arr.push_back(j);
        }
        arr1->set_len(len);
        arr1->read(arr);
             cout << "\nПример массива при худшем случае (сортировка слабой кучей): ";
        for(int j = 0; j < len; j++)
        {
           cout << arr[j] << " ";
        }
        start = std::chrono::steady_clock::now();
        arr1->sort_weak();
        end = std::chrono::steady_clock::now();
             cout << "\nВремя сортировки слабой кучей в худшем случае: " << chrono::duration<double>(end-start).count() << "\n";
        arr.clear();
        for (int j = 0; j < len; j++)
        {
           arr.push_back(1);
        }
        arr1->read(arr);
             cout << "\nПример массива при лучшем случае (сортировка слабой кучей): ";
        for(int j = 0; j < len; j++)
        {
```

```
            cout << arr[j] << " ";
        }
        start = std::chrono::steady_clock::now();
        arr1->sort_weak();
        end = std::chrono::steady_clock::now();
            cout << "\nВремя сортировки слабой кучей в лучшем случае: " << chrono::duration<double>(end-start).count() << "\n";
      }
   }
}

int main()
{
        SetConsoleCP(1251); // функции для работы с кириллицей (IDE CODE::BLOCKS плохо работает с кириллицей, поэтому приходится подключать соотвествущие функции
   SetConsoleOutputCP(1251);
   srand(time(NULL));
   Weak_Heap arr1;
   Quick_Sort arr2;
   panel(&arr1, &arr2);
   return 0;
}
```

Файл Weak_Heap.h:

```
#pragma once
#include <vector>
#include <iostream>
#include <string>

using namespace std;
```

```
class Weak_Heap
{
private:
   vector<int> bit; //свойство под массив битов

   void merge_weak_heap(int, int);

public:
   vector<int> wheap; //свойство под массив
   int len; //свойство под длину массива

   void print_leadge_list(int, string, int);
   void read(vector<int>);    //считывание массива слабой кучи с консоли
   void set_len(int);  //установка длины массива слабой кучи
   void print_wheap(); //вывод содержимого вектора wheap в консоль
   void print_bit();   //вывод сожержимого вектора bit в консоль
   void sort_weak();
   void make_weak_heap();
   void weak_sow(int, int);
};
```

Файл Weak_Heap.cpp:

```
#include "Weak_Heap.h"

void Weak_Heap::print_leadge_list(int index, string deg, int len)
{
   cout << deg << wheap[index] << "\n";
   deg = deg + "--";
   if (index == 0)
   {
```

```
        cout << "--#\n";
    }
    if (2*index+bit[index] < len && index != 0) //для левого
    {
        print_leadge_list(2*index+bit[index], deg, len);
    }
    if (2*index+1-bit[index] < len)
    {
        print_leadge_list(2*index+1-bit[index], deg, len); // для правого
    }
    if (2*index+bit[index] >= len && index != 0)
    {
        cout << deg << "#\n";
    }
    if (2*index+1-bit[index] >= len)
    {
        cout << deg << "#\n";
    }
}

void Weak_Heap::sort_weak()
{
    make_weak_heap();
    weak_sow(len-1, 0);
}

void Weak_Heap::weak_sow(int last_index, int flag)
{
    int i = 0;
    if (flag == 1)
```

```
{
   print_leadge_list(0, "", last_index+1);
   cout << "\n";
}
swap(wheap[0], wheap[last_index]);
while (2*i+bit[i] < last_index)
{
   if (i == 0)
   {
      i = 2*i+1;
   }
   else
   {
      i = 2*i+bit[i];
   }
}
while (i > 0)
{
   merge_weak_heap(i, 0);
   i = i/2;
}
if (last_index-1 != 1)
{
   if (flag == 1)
   {
      weak_sow(last_index-1, 1);
   }
   else
   {
      weak_sow(last_index-1, 0);
```

```
        }
    }
    else
    {
        if (flag == 1)
        {
            print_leadge_list(0, "", last_index);
        }
        merge_weak_heap(0, 1);
    }
}

void Weak_Heap::merge_weak_heap(int index1, int index2)
{
    if (wheap[index1] > wheap[index2])
    {
        if (2*index1 < len && index1 != 0)
        {
            if (bit[index1] == 0)
            {
                bit[index1] = 1;
            }
            else
            {
                bit[index1] = 0;
            }
        }
        swap(wheap[index1], wheap[index2]);
    }
}
```

```
void Weak_Heap::make_weak_heap()
{
   int temp;
   for (int i = len-1; i > 0; i--)
   {
      if (i % 2 == 1)
      {
         merge_weak_heap(i, (i-1)/2);
      }
      else
      {
         temp = i/2;
         while (temp % 2 == 0)
         {
            temp = temp/2;
         }
         merge_weak_heap(i, (temp-1)/2);
      }
   }
}

void Weak_Heap::read(vector<int> arr)
{
   wheap.clear();
   bit.clear();
   for (int i = 0; i < len; i++)
   {
      wheap.push_back(arr[i]);
      bit.push_back(0);
```

```
    }
}

void Weak_Heap::print_wheap()    //метод, выводящий содержимое вектора на экран
{
    cout << "\n";
    for (int i = 0; i < len; i++)
    {
        cout << wheap[i] << " ";
    }
    cout << "\n";
}

void Weak_Heap::print_bit()
{
    cout << "\n";
    for (int i = 0; i < len; i++)
    {
        cout << bit[i] << " ";
    }
    cout << "\n";
}

void Weak_Heap::set_len(int len)
{
    this->len = len;
}
```

Файл Quick_Sort.h:

```
#pragma once
#include <vector>
#include <iostream>

using namespace std;

class Quick_Sort
{
private:
    int len;
public:
    vector<int> qheap;

    int get_len();
    void set_len(int);
    void read(vector<int>);
    void qsort(int, int, int);
    void print(int, int, int, int);
    void print_sorted();
};
```

Файл Quick_Sort.cpp:

```
#include "Quick_Sort.h"

int Quick_Sort::get_len()
{
    return len;
}

void Quick_Sort::read(vector<int> arr)
```

```
{
    qheap.clear();
    for (int i = 0; i < len; i++)
    {
        qheap.push_back(arr[i]);
    }
}

void Quick_Sort::set_len(int len)
{
    this->len = len;
}

void Quick_Sort::print_sorted()
{
    cout << "\n";
    for (int i = 0; i < len; i++) //вывод отсортированного вектора
    {
        cout << qheap[i] << " ";
    }
    cout << "\n";
}

void Quick_Sort::qsort(int start, int finish, int flag)
{
     int mid = qheap[start]; //опорный элемент (элемент, относительно которого производится текущая быстрая сортировка
    int index_mid = start; //индекс опорного элемента
     int icount = finish; //переменная, с помощью которой производится обход по текущей стороне вектора
```

```
        int check = finish; //переменная, с помощью которой производится корректный вывод отсортированных эелементов
    if (flag == 1)
    {
        cout << mid <<" - текущий центральный элемент быстрой сортировки\n";
    }
    do
    {
         if (mid > qheap[check]) //если опорный элемент больше расматриваемого в цикле while, то он перемещается слева от опорного элемента
        {
            qheap.insert(qheap.begin()+index_mid, qheap[check]);
            index_mid++;
            qheap.erase(qheap.begin()+check+1);
            check++;
        }
        if (flag == 1)
        {
            print(start, finish, check, index_mid);
        }
        icount--;
        check--;
    } while (icount >= start+1);
    if (flag == 1)
    {
        cout << "текущее состояние массива:\t";
        print(0, qheap.size()-1, 0, 0); //вывод вектора в конце текущей функции
        cout << "\n\n";
    }
```

```
    if (start < index_mid-1) //вызов сортировки для левой стороны
    {
        qsort(start, index_mid-1, flag);
    }
    if (index_mid+1 < finish) //вызов сортировеи для правой стороны
    {
        qsort(index_mid+1, finish, flag);
    }
}

void Quick_Sort::print(int start, int finish, int icount, int index_mid)
//вспомогательная функция для вывода вектора на консоль
{
    if (icount == 0) //вывод вектора в конце функции qsort
    {
        for (int i = 0; i <= finish; i++)
        {
            if (i == finish)
            {
                cout << qheap[i];
            }
            else
            {
                cout << qheap[i] << "-";
            }
        }
        cout << "\n";
        return;
    }
    for (int i = start; i <= finish; i++) //вывод вектора в ходе сортировки
```

```
    {
        if (i == index_mid && i != start)
        {
            cout<< "----";
        }
        else if (i == icount)
        {
            cout<< "----";
        }
        if (i == finish)
        {
            cout << qheap[finish];
        }
        else
        {
            cout << qheap[i] << "-";
        }
    }
    cout << "\n";
}
```