Министерство образования и науки РФ  
Федеральное государственное бюджетное образовательное учреждение   
Высшего образования  
**“Московский технологический университет”  
МИРЭА**

Институт информационных технологий(ИТ)

Отчет по лабораторной работе

по дисциплине

**“Анализ сложности алгоритмов”**

Студент группы ИКБО-08-16 Балашова Т.С.

Преподаватель Мирабо Е.И.

Лабораторные работы зачтены « » 2017 г

Москва 2017

**СОДЕРЖАНИЕ:**

ВВЕДЕНИЕ…………………………………………………………………………………….3

1. Постановка задачи………………………………………………………………………..4

2. Теоретический раздел…………………………………………………………………….5

2.1. Сортировка простым обменом ………………………………………………..5

2.2. Быстрая Сортировка Хоара ……………………………………………………6

2.3. Сортировка вставками …………………………………………………………7

2.4. Сортировка с помощью двоичного дерева ………………………………….8

2.5 Линейный поиск..…………………………………………………………………9

2.6 Интерполяционный поиск ...………………………………………………….10

2.7 Двоичное дерево поиска……………………………………………………….11

3. Практический раздел ………………………………………………………………….12  
 3.1 Алгоритмы сортировки и поиска …………………………………………..12

ЗАКЛЮЧЕНИЕ…………………………………………………………………………… 16

ПРИЛОЖЕНИЕ……………………………………………………………………………..17

**ВВЕДЕНИЕ**

Анализ сложности алгоритмов – это область теоретической информатики, но затрагивает вопросы и со стороны практики. Из наиболее важных приложений данной теории можно назвать способы построения и анализа эффективных алгоритмов.  
 Количественная характеристика потребляемых ресурсов , необходимых программе или алгоритму для работы(успешного решения задачи) – это и есть сложность алгоритма.  
 Основные ресурсы: время и объем памяти. Время более важный ресурс , нежели объём.  
Очевидно , что для разных входных значений алгоритму может потребоваться разное время (количество ресурсов) для решения задачи.  
 Цель данной работы заключается в использовании популярных алгоритмов сортировки на языке программирования и определения наиболее эффективных среди предложенных.

**1.ПОСТАНОВКА ЗАДАЧИ**

Анализ вычислительной сложности алгоритма сортировки массива методом простого обмена, быстрой сортировки, сортировки вставками, сортировки по двоичному дереву. Реализация и исследование линейного, бинарного, интерполяционного поисков, поиска по двоичному дереву.

В программе должны быть реализованы два варианта формирования массива(выбирается в начале программы):

1)ввод с клавиатуры(для тестового прогона программы), n = 10;

2)с помощью генератора псевдослучайных чисел (для рабочего прогона программы)  
 n = 10000, n = 30000 , n = 50000 , n = 70000 и n = 90000;

3) также предусмотрен вариант с чтением из файла.

Провести теоретическую и практическую оценки вычислительной сложности алгоритмов.

Примечания:

1)Формирование исходного массива из псевдослучайных чисел оформить в виде функции.

2)Практическая оценка вычислительной сложности алгоритма производится путем вычисления количества выполненных операций сравнения С и перемещения М.

**2.ТЕОРИТИЧЕСКИЙ РАЗДЕЛ**

**2.1 Сортировка простым обменом**

Сортировка простыми обменами - алгоритм сортировки прямым обменом -основан на принципе сравнения и обмена пары соседних элементов до тех пор, пока не будут отсортированы все элементы. Совершаются проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к началу массива.

Если рассматривать массивы как вертикальные, а не горизонтальные построения, то элементы можно интерпретировать как пузырьки в банке с водой, причем вес каждого соответствует его ключу. В этом случае при каждом проходе один пузырек как бы поднимается до уровня, соответствующего его весу. Такой метод известен под именем «пузырьковая сортировка».

Анализ алгоритма

Число сравнений в алгоритме прямого обмена

**С = (n2 - n)/2,**

а минимальное, среднее и максимальное число перемещений элементов равно соответственно

**Mmin = 0,**

**Mср = 3(n2 - n)/2,**

**Mmax = 3(n2 - n)/4**

**2.2 Быстрая сортировка, сортировка Хоара**

Быстрая сортировка, сортировка Хоара ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) quicksort), часто называемая qsort (по имени в стандартной [библиотеке](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) языка [Си](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))) — широко известный [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), разработанный английским информатиком [Чарльзом Хоаром](https://ru.wikipedia.org/wiki/%D0%A5%D0%BE%D0%B0%D1%80,_%D0%A7%D0%B0%D1%80%D0%BB%D1%8C%D0%B7_%D0%AD%D0%BD%D1%82%D0%BE%D0%BD%D0%B8_%D0%A0%D0%B8%D1%87%D0%B0%D1%80%D0%B4) во время его работы в [МГУ](https://ru.wikipedia.org/wiki/%D0%9C%D0%93%D0%A3) в [1960 году](https://ru.wikipedia.org/wiki/1960_%D0%B3%D0%BE%D0%B4).

[Один из самых быстрых](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D0%B8_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B9) известных универсальных алгоритмов сортировки массивов: в среднем  обменов при упорядочении  элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «[Пузырьковая сортировка](https://ru.wikipedia.org/wiki/%D0%9F%D1%83%D0%B7%D1%8B%D1%80%D1%8C%D0%BA%D0%BE%D0%B2%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0)» и «Шейкерная сортировка»), известного, в том числе, своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы. Любопытный факт: улучшение самого неэффективного прямого метода сортировки дало в результате один из наиболее эффективных улучшенных методов.

Общая идея алгоритма состоит в следующем:

* Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность.
* Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: «меньшие опорного», «равные» и «большие».
* Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

**Лучший случай**

**O(n\*log2 n)**

**2.3 Сортировка вставками**

Сортировка вставками ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Insertion sort) — [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов. [Вычислительная сложность](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C) —  **O(n^2)**

На вход алгоритма подаётся последовательность N чисел: a1 , a2 , …,an . Сортируемые числа также называют ключами. Входная последовательность на практике представляется в виде массива с N элементами. На выходе алгоритм должен вернуть перестановку исходной последовательности a’1 , a’2, ……,a’n, чтобы выполнялось следующее соотношение a’1 <= a’2 <= a’3 <= … <= a’n

В начальный момент отсортированная последовательность пуста. На каждом шаге алгоритма выбирается один из элементов входных данных и помещается на нужную позицию в уже отсортированной последовательности до тех пор, пока набор входных данных не будет исчерпан. В любой момент времени в отсортированной последовательности элементы удовлетворяют требованиям к выходным данным алгоритма.

**2.4 Сортировка с помощью двоичного дерева**

Сортировка с помощью двоичного дерева (сортировка двоичным деревом, сортировка деревом, древесная сортировка, сортировка с помощью бинарного дерева, [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) tree sort) — универсальный [алгоритм сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8), заключающийся в построении [двоичного дерева поиска](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0) по ключам массива (списка), с последующей сборкой результирующего массива путём обхода узлов построенного дерева в необходимом порядке следования ключей. Данная сортировка является оптимальной при получении данных путём непосредственного чтения с [потока](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) (например из файла, сокета или консоли).

**Алгоритм.**

1. Построение двоичного дерева.

2. Сборка результирующего массива путём обхода узлов в необходимом порядке следования ключей.

**Эффективность.**

Процедура добавления объекта в бинарное дерево имеет среднюю алгоритмическую сложность порядка O(log(n)). Соответственно, для n объектов сложность будет составлять O(n log(n)), что относит сортировку с помощью двоичного дерева к группе «быстрых сортировок». Однако, сложность добавления объекта в разбалансированное дерево может достигать O(n), что может привести к общей сложности порядка O(n²).

При физическом развёртывании древовидной структуры в памяти требуется не менее чем 4n ячеек дополнительной памяти (каждый узел должен содержать ссылки на элемент исходного массива, на родительский элемент, на левый и правый лист), однако, существуют способы уменьшить требуемую дополнительную память.

**2.5 Линейный поиск**

Линейный, последовательный поиск — [алгоритм](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) нахождения заданного значения произвольной функции на некотором отрезке. Данный алгоритм является простейшим алгоритмом поиска и, в отличие, например, от [двоичного поиска](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA), не накладывает никаких ограничений на функцию и имеет простейшую реализацию. Поиск значения функции осуществляется простым сравнением очередного рассматриваемого значения (как правило, поиск происходит слева направо, то есть от меньших значений аргумента к большим) и, если значения совпадают (с той или иной точностью), то поиск считается завершённым.

Если отрезок имеет длину N, то найти решение с точностью до **O(n)** можно за время . Т.о. [асимптотическая сложность алгоритма](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B0) — N\e. В связи с малой эффективностью по сравнению с другими алгоритмами линейный поиск обычно используют, только если отрезок поиска содержит очень мало элементов, тем не менее, линейный поиск не требует дополнительной памяти или обработки/анализа функции, так что может работать в потоковом режиме при непосредственном получении данных из любого источника. Также линейный поиск часто используется в виде линейных алгоритмов поиска максимума/минимума.

**2.6 Интерполяционный поиск**

Интерполяционный поиск (интерполирующий поиск) основан на принципе поиска в телефонной книге или, например, в словаре. Вместо сравнения каждого элемента с искомым, как при [линейном поиске](https://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA), данный алгоритм производит предсказание местонахождения элемента: поиск происходит подобно [двоичному поиску](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA), но вместо деления области поиска на две части, интерполирующий поиск производит оценку новой области поиска по расстоянию между ключом и текущим значением элемента. Другими словами, бинарный поиск учитывает лишь знак разности между ключом и текущим значением, а интерполирующий ещё учитывает и модуль этой разности и по данному значению производит предсказание позиции следующего элемента для проверки. В среднем интерполирующий поиск производит **log(log(N))** операций, где N есть число элементов, среди которых производится поиск. Число необходимых операций зависит от равномерности распределения значений среди элементов. В плохом случае (например, когда значения элементов экспоненциально возрастают) интерполяционный поиск может потребовать до O(N) операций.

**2.7 Двоичное дерево поиска**

Двоичное дерево поиска ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) binary search tree, BST) — это [двоичное дерево](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), для которого выполняются следующие дополнительные условия (свойства дерева поиска):

* Оба поддерева — левое и правое — являются двоичными деревьями поиска.
* У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X.
* У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равно, нежели значение ключа данных самого узла X.

Очевидно, данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше.

Как правило, информация, представляющая каждый узел, является записью, а не единственным полем данных. Однако это касается реализации, а не природы двоичного дерева поиска.

Для целей реализации двоичное дерево поиска можно определить так:

* Двоичное дерево состоит из узлов (вершин) — записей вида (data, left, right), где data — некоторые данные, привязанные к узлу, left и right — ссылки на узлы, являющиеся детьми данного узла — левый и правый сыновья соответственно. Для оптимизации алгоритмов конкретные реализации предполагают также определения поля parent в каждом узле (кроме корневого) — ссылки на родительский элемент.
* Данные (data) обладают ключом (key), на котором определена операция сравнения «меньше». В конкретных реализациях это может быть пара (key, value) — (ключ и значение), или ссылка на такую пару, или простое определение операции сравнения на необходимой структуре данных или ссылке на неё.
* Для любого узла X выполняются свойства дерева поиска: key[left[X]] < key[X] ≤ key[right[X]], то есть ключи данных родительского узла больше ключей данных левого сына и нестрого меньше ключей данных правого.

**ПРАКТИЧЕСКИЙ РАЗДЕЛ**

**3.1 Алгоритмы сортировки и поиска**

По завершении работы над реализацией алгоритмов была произведена оценка их сложности. Были зафиксированы следующие параметры: время, кол-во операций сравнения(С) и кол-во операций перемещения(М) для каждого алгоритма.

Функция случайной генерации чисел создает массив в диапазоне от 0 до 100000.

В результате сравнения при разном наборе данных была получена следующая таблица:

Таблица 1. Оценка практической работы алгоритмов сортировки

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Сортировка  пузырьком | Быстрая  сортировка | Сортировка  вставками | Сортировка  по бинарному  дереву |
| N = 10000 | | | | |
| Время выполнения(сек) | 0,93700000 | 0,00000000 | 0,25000000 | 0,01500000 |
| Кол-во сравнений | 99350064 | 127288 | 25124418 | 154936 |
| Кол-во перемещений | 25122927 | 30746 | 25124418 | 10001 |
| N = 30000 | | | | |
| Время выполнения(сек) | 8,36700000 | 0,00000000 | 2,29700000 | 0,01600000 |
| Кол-во сравнений | 893610212 | 487804 | 224764904 | 516191 |
| Кол-во перемещений | 224751266 | 99018 | 224764904 | 30001 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Сортировка  пузырьком | Быстрая  сортировка | Сортировка  вставками | Сортировка  по бинарному  дереву |
| N = 50000 | | | | |
| Время выполнения(сек) | 24,01300000 | 0,01600000 | 6,53100000 | 0,01600000 |
| Кол-во сравнений | 1811966957 | 765179 | 625778848 | 869504 |
| Кол-во перемещений | 625740492 | 173224 | 625778848 | 500001 |
| N = 70000 | | | | |
| Время выполнения(сек | 47,83100000 | 0,01600000 | 12,85700000 | 0,03100000 |
| Кол-во сравнений | 575353127 | 1104535 | 1229922792 | 1230973 |
| Кол-во перемещений | 1229847697 | 246482 | 1229922792 | 700001 |
| N = 90000 | | | | |
| Время выполнения(сек | 76,33400000 | 0,01600000 | 20,95400000 | 0,03200000 |
| Кол-во сравнений | 5154943099 | 1605484 | 2021086772 | 1669914 |
| Кол-во перемещений | 2020963430 | 315679 | 2021086722 | 900001 |

Таблица 2. Оценка практической работы алгоритмов поиска

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Линейный  поиск | Интерполяционный  поиск | Бинарный | Поиск  по двоичному  дереву |
| N = 10000 | | | | |
| Время выполнения(сек) | 0,00000000 | 0,00000000 | 0,00000000 | 0,00000000 |
| Кол-во сравнений | 99998 | 1 | 12 | 8 |
| Кол-во перемещений | 0 | 0 | 0 | 0 |
| N = 30000 | | | | |
| Время выполнения(сек) | 0,00000000 | 0,00000000 | 0,00000000 | 0,00000000 |
| Кол-во сравнений | 29984 | 1 | 10 | 7 |
| Кол-во перемещений | 0 | 0 | 0 | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Линейный  поиск | Интерполяционный  поиск | Бинарный | Поиск  по двоичному  дереву |
| N = 50000 | | | | |
| Время выполнения(сек) | 0,00000000 | 0,00000000 | 0,00000000 | 0,00000000 |
| Кол-во сравнений | 49710 | 2 | 14 | 21 |
| Кол-во перемещений | 0 | 0 | 0 | 0 |
| N = 70000 | | | | |
| Время выполнения(сек | 0,00000000 | 0,00000000 | 0,00000000 | 0,00000000 |
| Кол-во сравнений | 69709 | 4 | 15 | 6 |
| Кол-во перемещений | 0 | 0 | 0 | 0 |
| N = 90000 | | | | |
| Время выполнения(сек | 0,00000000 | 0,00000000 | 0,01600000 | 0,00000000 |
| Кол-во сравнений | 89709 | 5 | 16 | 18 |
| Кол-во перемещений | 0 | 0 | 0 | 0 |

**Заключение**

В результате изучения методов сортировок, их применения и сравнения были выявлены наиболее эффективные алгоритмы сортировок и поиска.

При выполнении лабораторной работы наиболее эффективным алгоритмом сортировки оказалась “Быстрая” Сортировка Хоара, а наиболее эффективным алгоритмом поиска оказался интерполяционный поиск.

Эти алгоритмы показали наименьшее время, а также использовали наименьшее количество итераций и сравнений(в зависимости от размера массива).

**Приложение**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int C = 0;

int M = 0;

clock\_t time1, time2;

struct data

{

float result;

int time;

} \*dat = NULL;

struct data\* tmpdat = NULL;

struct Node

{

struct data value;

struct Node \*left;

struct Node \*right;

};

struct data\* key\_enter(int );

struct data\* random\_enter(int);

struct data\* copyArr (struct data\* , int );

void bubble\_sort(struct data\*, int);

void quick\_sort(struct data\*, int, int);

void insertion\_sort(struct data\*, int);

struct Node\* createNode(struct data );

struct Node\* createTree(struct Node \*, struct data);

void printTtree(struct Node \*root);

int Tree\_search(struct Node \*, int );

struct Node\* binarytree\_sort (struct data\* , int );

int Linear\_search (struct data\* , int , int );

int Interpolation\_search (struct data\* ,int , int , int );

int Binary\_search( struct data\* , int , int );

void printArr(struct data\*, int);

int main()

{

int choice;

int key;

int keyindex;

int i = 0;

int size;

char s[100];

FILE \*pf;

printf("To read from file enter 1\nTo enter from the keyboard enter 2\nTo work with a random array enter 3\n");

scanf("%d", &choice);

switch(choice)

{

case 1:

pf = fopen("C:\\\\laba1\_1.csv","r");

if (pf == NULL)

{

while (pf == NULL)

{

printf("Enter a file path:\n");

scanf("%s", s);

pf = fopen(s, "r");

}

}

dat=(struct data\*) malloc (sizeof(struct data));

while (fscanf (pf, "%f%\*c%d", &(dat[i].result), &(dat[i].time))!= EOF)

{

i++;

dat = (struct data\*) realloc (dat,(i+1)\*sizeof(struct data));

}

size = i-1;

size = size;

fclose(pf);

break;

case 2:

printf("\nEnter the number of values:");

scanf("%d", &size);

dat = key\_enter(size);

break;

case 3:

printf("\nEnter the number of values:");

scanf("%d", &size);

srand(time(NULL));

dat = random\_enter(size);

break;

default:

printf("Error\n");

return 1;

}

printf("\n\nTo search element(time value) in unsorted array enter 1\nEnter any other key to continue\n");

scanf("%d", &choice);

if (choice == 1)

{

printf("\nEnter element(time value) to find:");

scanf("\n%d", &key);

C = 0; M = 0;

time1 = clock();

keyindex = Linear\_search(dat, size, key);

time2 = clock();

double time = (double)(time2 - time1) /CLOCKS\_PER\_SEC;

printf("Time:%0.8lf sec\t \nComparisons:%d\nMove operations:%d\n", time, C, M);

}

printf("\n\n\tBubble sort");

tmpdat = copyArr(dat, size);

time1 = clock();

bubble\_sort(tmpdat, size);

time2 = clock();

double time = (double)(time2 - time1) /CLOCKS\_PER\_SEC;

printf("\nTime:%0.8lf sec\t \nComparisons:%d\nMove operations:%d", time, C, M);

M = 0;

C = 0;

printf("\n\n\tQuick sort");

tmpdat = copyArr(dat, size);

time1 = clock();

quick\_sort(tmpdat, 0, size-1);

time2 = clock();

time = (double)(time2 - time1) /CLOCKS\_PER\_SEC;

printf("\nTime:%0.8lf sec\t \nComparisons:%d\nMove operations:%d", time, C, M);

M = 0;

C = 0;

printf("\n\n\tBinary tree sort");

struct Node\* node;

tmpdat = copyArr(dat, size);

time1 = clock();

node = binarytree\_sort(tmpdat, size);

time2 = clock();

time = (double)(time2 - time1) /CLOCKS\_PER\_SEC;

printf("\nTime:%0.8lf sec\t \nComparisons:%d\nMove operations:%d", time, C, M);

M = 0;

C = 0;

printf("\n\n\tInsertion sort");

tmpdat = copyArr(dat, size);

time1 = clock();

insertion\_sort(tmpdat, size);

time2 = clock();

time = (double)(time2 - time1) /CLOCKS\_PER\_SEC;

printf("\nTime:%0.8lf sec\t \nComparisons:%d\nMove operations:%d", time, C, M);

printf("\n\nTo search element(time value) enter 1\nEnter any other key to continue\n");

scanf("%d", &choice);

if (choice == 1)

{

printf("\nEnter element(time value) to find:");

scanf("\n%d", &key);

printf("\n\n\tLinear search\n\n");

C = 0; M = 0;

time1 = clock();

keyindex = Linear\_search(tmpdat, size, key);

time2 = clock();

time = (double)(time2 - time1) /CLOCKS\_PER\_SEC;

printf("Time:%0.8lf sec\t \nComparisons:%d\nMove operations:%d\n", time, C, M);

printf("\n\n\tInterpolation search\n");

C = 0; M = 0;

time1 = clock();

keyindex = Interpolation\_search(tmpdat, key, 0, size-1);

time2 = clock();

time = (double)(time2 - time1) /CLOCKS\_PER\_SEC;

printf("Time:%0.8lf sec\t \nComparisons:%d\nMove operations:%d\n", time, C, M);

printf("\n\n\tBinary search\n");

C = 0; M = 0;

time1 = clock();

keyindex = Binary\_search( tmpdat, key, size);

time2 = clock();

time = (double)(time2 - time1) /CLOCKS\_PER\_SEC;

printf("Time:%0.8lf sec\t \nComparisons:%d\nMove operations:%d", time, C, M);

printf("\n\n\tTree search\n");

C = 0; M = 0;

time1 = clock();

Tree\_search( node, key);

time2 = clock();

time = (double)(time2 - time1) /CLOCKS\_PER\_SEC;

printf("Time:%0.8lf sec\t \nComparisons:%d\nMove operations:%d", time, C, M);

}

printf("\n\nTo print array enter 1\nEnter any other key to end the program\n");

scanf("%d", &choice);

if (choice == 1)

printArr(tmpdat, size);

else return 0;

return 0;

}

struct data\* key\_enter (int size)

{

int i;

struct data\* dat;

dat = (struct data\*) malloc (sizeof(struct data)\*size);

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

{

printf("\nEnter result %d:", i+1);

scanf("%f", &dat[i].result);

printf("\nEnter time %d:", i+1);

scanf("%d", &dat[i].time);

}

return dat;

}

struct data\* random\_enter(int size)

{

int i;

struct data\* dat;

dat = (struct data\*) malloc (sizeof(struct data)\*size);

srand(time(NULL));

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

{

dat[i].result = (rand()%101)\*0.01;

dat[i].time = rand()%100000;

}

return dat;

};

struct data\* copyArr (struct data\* dat, int size )

{

struct data\* datcopy;

datcopy = (struct data\*) malloc (sizeof(struct data)\* size);

int i = 0;

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

datcopy [i] = dat [i];

return datcopy;

}

void bubble\_sort(struct data\* dat, int size)

{

int i;

int sort = 0;

do

{

sort = 0;

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

{

if(dat[i].time < dat[i-1].time)

{

struct data tmp;

tmp = dat[i];

dat[i] = dat[i-1];

dat[i-1] = tmp;

M++;

sort = 1;

}

C++;

}

} while(sort != 0);

}

void quick\_sort(struct data\* dat, int left, int right)

{

int i = left;

int j = right;

int elem = dat[(i + j) / 2].time;

do

{

while (dat[i].time < elem)

{

i++;

C++;

}

while (dat[j].time > elem)

{

j--;

C++;

}

if(i <= j)

{

if (dat[i].time > dat[j].time)

{

struct data tmp;

tmp = dat[i];

dat[i] = dat[j];

dat[j] = tmp;

C++;

M++;

}

i++;

j--;

}

} while (i <= j);

if (i < right)

quick\_sort(dat, i, right);

if (left < j)

quick\_sort(dat, left, j);

}

void insertion\_sort(struct data\* dat, int size)

{

int i, j;

struct data tmp;

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

{

tmp = dat[i];

for (j = i-1; j >= 0;j--)

{

if (dat[j].time < tmp.time )

break;

dat[j+1] = dat[j];

dat[j] = tmp;

C++;

M++;

}

}

}

struct Node\* createNode(struct data val)

{

struct Node\* root = (struct Node\*) malloc(sizeof(struct Node));

root->left = root->right = NULL;

root->value = val;

return root;

}

struct Node\* createTree(struct Node \*root, struct data val)

{

C++;

if (root == NULL)

{

root = createNode(val);

M++;

}

if (val.time < root->value.time )

{

if (root->left == NULL)

root->left = createNode(val);

else

createTree(root->left, val);

}

if (val.time > root->value.time)

{

if (root->right == NULL)

root->right = createNode(val);

else

createTree(root->right, val);

}

return root;

}

struct Node\* binarytree\_sort (struct data\* dat, int size)

{

int i = 0;

struct Node \*node = NULL;

while (i < size)

{

node = createTree(node, dat[i]);

M++;

i++;

}

return node;

}

void printTree(struct Node \*root)

{

if (root != NULL)

{

printTree(root->left);

printf("%0.10f\t %d \t\n", root->value.result, root->value.time);

printTree(root->right);

}

}

int Tree\_search(struct Node \*root, int key)

{

C++;

if (root == NULL)

{

printf("No such element(%d) in the array.\n", key);

return -1;

}

if (key == root->value.time)

{

printf("Element(%d) is found\n", root->value.time);

return root->value.time;

}

if (key < root->value.time)

return Tree\_search(root->left, key);

if (key > root->value.time)

return Tree\_search(root->right, key);

}

int Linear\_search (struct data\* dat, int size, int key )

{

int i;

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

{

if (dat[i].time == key)

{

printf("Element's(%d) number is %d.\n", key , i+1);

break;

}

C++;

}

if (i == size)

printf("No such element(%d) in the array.\n", key);

return i;

}

int Interpolation\_search (struct data\* dat,int key, int left, int right)

{

int mid = 0;

while ((dat[left].time < key) && (key <dat[right].time))

{

mid = left + (key - dat[left].time) \* (right - left) / (dat[right].time - dat[left].time);

C++;

if (dat[mid].time < key )

left = mid + 1;

else if (dat[mid].time > key)

right = mid - 1;

else

{

printf("\nElement's(%d) number is %d.\n", key , mid+1);

return mid;

}

}

if (dat[left].time == key)

{

printf("\nElement's(%d) number is %d.\n", key , left+1);

return left;

}

else if (dat[right].time == key)

{

printf("\nElement's(%d) number is %d.\n", key , right+1);

return right;

}

else

printf("No such element(%d) in the array.\n", key);

}

int Binary\_search( struct data\* dat, int key, int size)

{

int left = 0;

int right = size - 1;

int mid;

while (left <= right)

{

mid = (left + right) / 2;

if (key < dat[mid].time)

{

right = mid - 1;

C++;

}

else if (key > dat[mid].time)

{

left = mid + 1;

C++;

}

else

{

printf("\nElement's(%d) number is %d.\n", key , mid+1);

return mid;

}

}

printf("No such element(%d) in the array.\n", key);

}

void printArr (struct data\* dat, int size)

{

int i;

printf(" Ascending sort :\t\t\t Descending sort: \n");

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

printf("%0.10f\t %12d \t |\t%0.10f\t %d\n",dat[i].result, dat[i].time, dat[size-i-1].result, dat[size-i-1].time);

}