МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное бюджетное
образовательное учреждение высшего образования

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

С.В. Самуйлов

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Лабораторный практикум

Пенза
Издательство ПГУ
2017

УДК 004.67

Р е ц е н з е н т
кандидат технических наук, доцент кафедры
«Менеджмент, информатика и общегуманитарные науки»
Пензенского филиала Финуниверситета
О.В. Прокофьев

Самуйлов С.В.
Структуры и алгоритмы обработки данных: лабораторный практикум. / С.В. Самуйлов – Пенза: Изд-во Пенз. гос. ун-та, 2017. – 42 с.

Рассматриваются наиболее известные алгоритмы поиска и сортировки. В заданиях к лабораторным работам основное внимание уделяется оценке алгоритмов по быстродействию и требуемой памяти. Даны задания к курсовой работе.

Лабораторный практикум подготовлен на кафедре «Математическое обеспечение и применение ЭВМ» и рекомендуется для бакалавров направления подготовки 09.03.04 «Программная инженерия», изучающих дисциплину «Структуры и алгоритмы обработки данных».

УДК 004.67

1. Лабораторные работы

Лабораторная работа № 1

Последовательный поиск

Цель лабораторной работы – сравнение и анализ алгоритмов последовательного поиска в неупорядоченной и упорядоченной последовательности данных.

Теоретическая часть

Поиском называется процедура выделения из множества записей некоторого его подмножества, записи которого удовлетворяют заданному условию. Множество записей может храниться в массиве, связанном списке, дереве или графе.

Здесь и далее под записью понимается как отдельное поле, содержащее одно значение, так и структура любой допустимой степени сложности.

Будем предполагать, что имеется множество из N записей и необходимо определить положение соответствующей записи. Предположим, что каждая запись содержит специальное поле, которое называется **ключом.**

В алгоритмах поиска обычно задан **аргумент** поиска K, и задача состоит в нахождении записи, имеющей K своим ключом. Существуют две возможности окончания поиска: либо поиск оказался **удачным**, т.е. была найдена запись с ключом K, либо он оказался **неудачным**, т.е. ключ K не был найден ни в одной записи.

Хотя целью поиска является информация, которая содержится в записи, в дальнейшем будем рассматривать алгоритмы, в которых обрабатываются только ключи.

Эффективность различных алгоритмов поиска оценивается количеством сравниваемых пар ключей и количеством их перестановок или сдвигов, необходимым для выполнения поиска. Обычно оценивается **среднее** и **максимальное** количество сравнений и перестановок или сдвигов.

Рассмотрим несколько алгоритмов последовательного (линейного) поиска. При этом основное внимание будет уделено сравнительному анализу изучаемых алгоритмов, их сложностным и временным характеристикам.

Неоптимальный последовательный поиск

Если нет никакой дополнительной информации о способе хранения данных, т.е. вероятнее всего данные расположены в произвольном порядке, то очевидный подход – простой последовательный просмотр массива [1]. Сформулируем алгоритм более точно.

Алгоритм последовательного поиска

Пусть имеется неупорядоченный массив записей с ключами K_1, K_2, ...,K_n. Необходимо найти запись с заданным ключом K [2].

1. $i = 1$.
2. Если $K = K_i$, алгоритм заканчивается удачно.
3. Иначе $i = i + 1$.
4. Если $i <= N$, то вернуться к шагу 2.
5. Иначе алгоритм заканчивается неудачно.

Условия окончания поиска следующие.
1. Элемент найден, т.е. обнаружен ключ $K_i = K$.
2. Все записи просмотрены, и совпадение не обнаружено.

Ясно, что **максимальное число сравнений** при удачном и неудачном поиске равно N.

Среднее число сравнений при удачном поиске равно $(N + 1)/2$.

Для этого метода поиска как среднее, так и максимальное время поиска пропорционально n, т.е. **сложность рассмотренного алгоритма** равна $O(n)$.

Оптимальный последовательный поиск

Приведенный выше алгоритм при всей своей очевидности не является, однако оптимальным. В алгоритме последовательного поиска в цикле имеются две проверки: на совпадение ключей и на выход значения индекса за пределы массива.

Если гарантировать, что ключ всегда будет найден, то без второго условия можно обойтись. Для этого достаточно в конце массива ($N+1$-й элемент) поместить дополнительный элемент, присвоив ему значение искомого ключа K. Назовем такой вспомогательный элемент «**барьером**», так как он предотвращает выход за пределы массива [1].

Алгоритм оптимального последовательного поиска

Пусть имеется неупорядоченный массив записей с ключами $K_1, K_2, ..., K_n$. Необходимо найти запись с заданным ключом K.

1. $K_{n+1} = K$
2. $i = 1$.
3. Если $K = K_i$, на шаг 5.
4. Иначе $i = i + 1$. На шаг 3.
5. Если $i = N+1$, то поиск неудачен (ключ найден в барьере), иначе – поиск удачен (ключ найден в i-ом элементе массива).

Таким образом, ключ всегда будет найден в записи с индексом i.

При $i = N+1$ (индекс достиг конца массива) ключ не найден (т.е. найден только в барьере). Поиск неудачен. В противном случае – поиск удачен.

Такой подход позволяет исключить в среднем $(N + 1)/2$ сравнений.

Алгоритм, реализующий описанный выше метод поиска, называется **алгоритмом оптимального последовательного поиска**.

Сложность рассмотренного алгоритма также равна $O(n)$.

Последовательный поиск в упорядоченном массиве

Если даже массив упорядочен, то и в этом случае иногда целесообразно использовать последовательный поиск. Нет смысла программировать более сложные алгоритмы поиска или использовать для хранения данных более сложную структуру, если количество записей невелико, либо поиск должен быть выполнен однократно.

Если известно, что ключи упорядочены (например, расположены в возрастающем порядке), то алгоритм последовательного поиска целесообразно несколько изменить.

Пусть имеется множество записей с ключами $K_1, K_2, ...,K_n$, причем $K_1 \le K_2 \le..\le K_n$. Необходимо найти запись с заданным ключом K. В целях увеличения скорости работы в конец массива помещается дополнительный элемент – «**барьер»**, значение которого должно быть больше значения искомого ключа K (например, $K+1$).

В отличие от поиска в неупорядоченном массиве просмотр элементов упорядоченного массива заканчивается в тот момент, когда первый раз выполнится условие $K \le K_i$. Очевидно, что все элементы $K_{i+1}, K_{i+2},..., K_n$ будут больше K [3].

Если $K = K_i$ – поиск удачен. В противном случае, искомого элемента нет в массиве.

Алгоритм оптимального последовательного поиска в упорядоченном массиве

Пусть имеется упорядоченный массив записей с ключами $K_1, K_2, ...,K_n$. Необходимо найти запись с заданным ключом K.

1. $K_{n+1} = K+1$
2. $i = 1$.
3. Если $K <= K_i$, на шаг 5.
4. Иначе $i = i + 1$, на шаг 3.
5. Если $K = K_i$, то поиск удачен, иначе – поиск неудачен.

В случае удачного поиска алгоритм поиска в упорядоченном массиве не лучше алгоритма поиска в неупорядоченном массиве, однако, отсутствие нужного ключа этот алгоритм позволяет обнаружить в два раза быстрее.

Таким образом, среднее число сравнений и при удачном, и при неудачном поиске равно $(N+1)/2$, следовательно, с**ложность рассмотренного алгоритма** также равна $O(n)$.

Задание на лабораторную работу

1. Задан неупорядоченный массив. Написать программу оценки временных характеристик работы алгоритмов оптимального и неоптимального последовательного поиска.

2. Задан упорядоченный массив. Написать программу оценки временных характеристик работы алгоритма оптимального последовательного поиска в неупорядоченном массиве и алгоритма оптимального последовательного поиска в упорядоченном массиве.

Массив из 500000÷1000000 элементов задавать с помощью датчика случайных чисел. В случае высокого быстродействия используемого компьютера для снятия временных характеристик поиск следует осуществлять многократно – 100÷10000 раз либо увеличить размерность исходного массива.. Подобрать размер массива и, если требуется, количество зацикливаний таким образом, чтобы минимальное время поиска было не менее 100 тиков.

На форме предусмотреть поле для ввода искомого ключа. Выводить для каждого метода поиска результат поиска (индекс найденного элемента/поиск неудачен), а также время поиска ключа.

Контрольные вопросы

1. Дайте определение понятия «поиск».
2. Каково среднее количество сравнений при поиске в неупорядоченном массиве: если поиск неудачен, если поиск удачен?
3. Каково среднее количество сравнений при поиске в упорядоченном массиве: если поиск неудачен, если поиск удачен?
4. В чем отличие поиска ключа в неупорядоченном и упорядоченном массивах?

Лабораторная работа № 2
Бинарный поиск

Цель лабораторной работы – сравнение и анализ алгоритмов бинарного (двоичного) поиска в упорядоченном массиве.

Теоретическая часть

Алгоритм **бинарного поиска** является одним из наиболее эффективных алгоритмов поиска в упорядоченном массиве. Упрощенно этот алгоритм состоит в следующем. Аргумент поиска сравнивается со средним элементом массива. Если они равны, то поиск успешно заканчивается. В противном случае поиск аналогичным образом должен быть осуществлен в верхней или нижней половине массива [1, 3].

Алгоритм неоптимального бинарного поиска

Пусть имеется массив, значения которого упорядочены по возрастанию, т.е. $K_1 \le K_2 \le ... \le K_n$. Необходимо найти элемент массива, равный заданному аргументу поиска K.

1. $L = 1$. Левая граница массива.
2. $R = n$. Правая граница массива.
3. Если $R < L$, то алгоритм заканчивается неудачно.
4. $i = \lfloor (L + R)/2 \rfloor$.
5. Если $K = K_i$, алгоритм заканчивается успешно.
6. Иначе если $K < K_i$, то $R = i - 1$. Перейти на шаг 3.
7. Иначе $(K > K_i)$ $L = i + 1$. Перейти на шаг 3.

Алгоритм оптимального бинарного поиска

Приведенный выше алгоритм в случае **успешного поиска** позволяет найти элемент в среднем за $log_2(N+1)-1$ сравнений. При **неудачном поиске** требуется $log_2(N+1)$ сравнений, т.е. **в среднем** удачный поиск отличается от неудачного только на одно сравнение.

Основываясь на этом факте можно упростить алгоритм бинарного поиска, убрав из цикла (шаг 5) проверку на равенство $K = K_i$. Так как в худшем случае цикл выполняется $log_2(N+1)$ раз, то именно столько раз не будет выполняться данная проверка, что является нашим **выигрышем.**

Однако в этом случае добавляется в среднем одна итерация в цикле, так как всегда приходится искать до тех пор, пока левая и правая границы не пересекутся. Кроме того, необходимо выполнить одно сравнение после цикла, проверяющее наличие искомого элемента на пересечении левой и правой границы массива (наш **проигрыш**).

Тогда алгоритм оптимального бинарного поиска будет состоять из следующих шагов [4].

1. $L = 1$. Левая граница массива.
2. $R = n$. Правая граница массива.
3. Если $R <= L$, то перейти на шаг 7.
4. $i = \lfloor (L + R)/2 \rfloor$.
5. Если $K <= K_i$, то $R = i$. Перейти на шаг 3.
6. Иначе $(K > K_i)$ $L = i + 1$. Перейти на шаг 3.
7. Если $K_R = K$, то поиск удачен. Иначе элемента нет в массиве.

Алгоритм оптимального бинарного поиска имеет те же характеристики, что и алгоритм неоптимального бинарного поиска. Следовательно, с**ложность рассмотренных алгоритмов** бинарного поиска – $O(log_2(N))$.

Задание на лабораторную работу

Написать программу оценки временных характеристик работы алгоритмов неоптимального бинарного поиска, оптимального бинарного поиска и последовательного поиска в упорядоченном массиве.

Массив из 300000÷1000000 элементов задавать с помощью датчика случайных чисел. В случае высокого быстродействия используемого компьютера для снятия временных характеристик поиск следует осуществлять многократно – 10000÷300000 раз. Подобрать размер массива и, если требуется, количество зацикливаний таким образом, чтобы минимальное время поиска было не менее 100 тиков.

На форме предусмотреть поле для ввода искомого ключа. Выводить для каждого метода поиска результат поиска (индекс найденного элемента/поиск неудачен), а также время поиска ключа.

Контрольные вопросы

1. Каково среднее число сравнений для алгоритма бинарного поиска, если поиск удачен?
2. Каково число сравнений для алгоритма бинарного поиска, если поиск неудачен?
3. В чем отличие алгоритма бинарного поиска от алгоритма оптимального бинарного поиска?

Лабораторная работа № 3
Хеширование

Цель лабораторной работы – сравнение и анализ функций хеширования, методов хранения и поиска информации на основе хеширования.

Теоретическая часть

В рассматриваемых ранее методах поиска время поиска является функцией от n, где n – количество элементов. Лучший из рассмотренных методов (бинарный поиск) имеет сложность $O(\log_2 n)$.

Эффективными методами поиска являются те методы, которые минимизируют число «ненужных» сравнений. В идеале хотелось бы иметь такую организацию данных, при которой вообще не было бы «ненужных» сравнений, т.е. ключ должен быть найдет за одно сравнение.

Но если каждый ключ должен быть извлечен за одно обращение, то положение записи (ее адрес) должно зависеть только от значения ключа этой записи. Следовательно, необходим метод преобразования ключа в некоторый адрес в заданном диапазоне.

Функция, преобразующая ключ в некоторый адрес, называется **хеш-функцией.** Если H – некоторая хеш-функция, а K – ключ, то $H(K)$ является значением хеш-функции или **хеш-адресом.**

Любая хорошая функция хеширования должна как можно равномернее распределять ключи по всему диапазону значений адресов. Однако всегда существует вероятность того, что найдутся ключи $K_1 \neq K_2$ такие, что $H(K_1)=H(K_2)$. Такая ситуация называется **коллизией при хешировании.**

Таким образом, при использовании метода хеширования необходимо решить две задачи: **выбрать функцию хеширования** и **метод разрешения коллизий** [3].

Каждый ключ может быть цифровым (номер зачетной книжки), алфавитным (ФИО студента) или алфавитно-цифровым (номер группы). Однако всегда имеется возможность преобразовать ключи в целые числа, поэтому будем предполагать, что множество ключей состоит из целых величин.

Функции хеширования

Выбор хеш–функции – непростая задача. Такая функция должна удовлетворять двум требованиям: равномерное распределение записей по всей таблице, что минимизирует количество коллизий, и быстрота вычисления.

Рассмотрим наиболее известные и эффективные с точки зрения озвученных выше требований функции преобразования ключа в адрес.

Метод деления

Наиболее распространенная функция хеширования основывается на **методе деления** и определяется в виде

$$H(K) = (K \bmod m) + 1,$$

где K – ключ, *mod* – операция, вычисляющая остаток от деления, m – делитель.

Равномерность распределения получаемых адресов во многом зависит от выбранного делителя m. Следует избегать четных делителей, так как при этом четные и нечетные ключи отображаются соответственно в четные и нечетные адреса. Если множество ключей состоит в основном из четных или в основном из нечетных ключей, будут возникать многочисленные коллизии.

Как показывают исследования, если m является большим простым числом, то количество коллизий невелико. Также неплохие результаты дает выбор в качестве делителя m нечетного числа, имеющего 20 и более делителей.

Метод середины квадрата

При **хешировании по методу середины квадрата** исходный ключ умножается сам на себя (возводится в квадрат). В качестве адреса выбирается столько цифр из середины результата, какова требуемая длина адреса.

Рассмотрим вышеизложенное на примере. Пусть дан ключ 234583. При возведении его в квадрат получаем 7539582389. Если требуется 100 адресов, то адрес будет равен 58, если необходимо 1000 адресов – 582, если 10000 – 9582.

Иногда необходимо получить некратное 10 количество адресов, например 736. В этом случае необходимо взять три средние цифры и умножить на коэффициент 0.736. Например, 582*0.736 = 428.

Эксперименты с реальными данными показали, что метод середины квадрата дает неплохой результат при условии, что ключи не содержат много левых или правых нулей подряд.

Метод свертывания

В **методе свертывания** ключ разбивается на части, каждая из которых имеет длину, равную длине требуемого адреса. Адрес формируется как сумма этих частей. При этом **перенос в старший разряд игнорируется.** Экспериментальным путем было показано, что свертывание справа налево дает меньшее число коллизий, чем свертывание слева направо.

Пусть дан ключ 3415768898. Для двух-, трех- и четырехцифрового адреса получим следующие значения:

98+88+76+15+34 = 11;
898+768+415+3 = 084;
8898+1576+34 = 0508

Метод свертывания используется, как правило, для больших ключей.

Метод умножения

Пусть количество адресов равно m. Зафиксируем константу A в интервале $0<A<1$ и положим

$$H(K)=\lfloor m \cdot D(K \cdot A) \rfloor$$

где $D(K \cdot A)$ – дробная часть произведения $K \cdot A$.

Достоинство метода в том, что качество хеш-функции мало зависит от выбора m.

Метод умножения работает при любом выборе константы A, но некоторые ее значения могут быть лучше других. В литературе предлагается следующее значение константы A

$$A \approx \left(\sqrt{5}-1\right)/2 \approx 0.6180339887...$$

Метод преобразования систем счисления

Этот метод получения хеш-адреса заключается в том, что ключ, представленный в системе счисления q, рассматривается как число в системе счисления s, где $s>q$, причем s и q – взаимно простые.

Число из системы счисления с основанием s переводится в систему счисления с основанием q, а адрес формируется путем выбора правых цифр нового числа (или любым другим рассмотренным выше методом).

Пусть задан ключ $(530476)_{10}$. Рассматривая его как $(530476)_{11}$, переведем в десятичную систему счисления с помощью следующих вычислений:

$$(530476)_{11} = 5*11^5 + 3*11^4 + 4*11^2 + 7*11^1 + 6*11^0 = (849745)_{10}.$$

При выборе хеш-функции важна **эффективность ее вычисления**, так как поиск некоторого объекта за одну попытку не будет эффективнее, если на эту попытку затрачивается больше времени, чем на несколько попыток при альтернативном методе.

Универсальное хеширование.

Если недоброжелатель будет специально подбирать данные для хеширования, то (зная функцию h) он. может устроить так, что все m ключей будут соответствовать одной позиции в таблице, в результате чего время поиска будет равно $O(m)$. Любая фиксированная хеш-функция может быть дискредитирована таким образом.

Единственный выход из положения – выбирать хеш-функцию случайным образом, не зависящим от того, какие именно данные вы хешируете. Такой подход называется **универсальным хешированием**. Что бы ни предпринимал ваш недоброжелатель, если он не имеет информации о выбранной хеш-функции, среднее время поиска останется хорошим.

Основная идея универсального хеширования – выбирать хеш-функцию во время исполнения программы случайным образом из некоторого множества. Недостатком данного подхода может считаться то, что при повторном вызове с теми же входными данными алгоритм будет работать уже по-другому, давать другой результат.

Такой подход гарантирует, что нельзя придумать входных данных, на которых алгоритм всегда бы работал неэффективно.

Методы разрешения коллизий

Существует два основных метода разрешения коллизий: **метод открытой адресации** и **метод цепочек**.

Метод открытой адресации

Пусть задан ключ K и массив S, состоящий из n элементов. Пусть $d=H(K)$ – получаемый при использовании некоторой хеш-функции H индекс массива S, причем $1 \le d \le n$. Необходимо разместить ключ K в массиве S.

Если элемент массива S_d свободен, то ключ K помещается в эту позицию и включение элемента завершается. Если же элемент S_d уже занят, в массиве просматриваются другие элементы. Эти действия выполняются до тех пор, пока не будет найдено свободное место для размещения ключа K.

Простейший способ поиска свободной позиции состоит в последовательном просмотре элементов массива с индексами

$$d, d+1, ..., n-1, n, 1, 2, ..., d-1.$$

Если в массиве есть хотя бы один свободный элемент, он будет найден. В противном случае, после просмотра всех позиций массива можно сделать заключение о том, что массив переполнен и добавление нового ключа невозможно.

Алгоритм поиска ключа состоит в вычислении его хеш-адреса и просмотре той же последовательности элементов. Либо, при удачном поиске, будет найден искомый ключ, либо, при поиске неудачном, будет найден пустой элемент или будут просмотрены все элементы массива.

Рассмотрим применение метода открытой адресации на примере. Пусть задан массив *S* из 11 элементов, хеш-функция *H(K) = (K mod 11) + 1* и последовательность ключей 88, 21, 96, 86, 11, 22, 5, 29, 19.

Результаты заполнения массива методом открытой адресации показаны на рисунке 1. Как видно из рисунка, коллизия возникает при занесении ключей 11, 22 и 19.

1	2	3	4	5	6	7	8	9	10	11
88	11	22	19		5		29	96	86	21

Рис. 1. Метод открытой адресации

Если из массива удаляется ключ K_i, то в соответствующий элемент массива необходимо занести специальный признак удаленного ключа (например, отрицательное число, если все ключи положительные, или наибольшее положительное число и т.п.), т.е. значение, которое не равно ни одному возможному значению ключа.

При поиске помеченные как удаленные записи рассматриваются как занятые, а при занесении, как свободные.

Разрешение коллизий с помощью метода открытой адресации имеет ряд недостатков.

1. Метод неэффективен в случае большого количества удалений ключей из массива. При этом может возникнуть ситуация, когда массив будет содержать только записи, помеченные как удаленные, а поиск отсутствующей записи потребует просмотра всего массива.

2. Другим недостатком метода является **эффект скучивания**, который усиливается при почти заполненном массиве. Эффект скучивания заключается в следующем. Для рассмотренного выше примера при пустом массиве вероятность того, что новый элемент попадет в первую позицию, равна 1/11. Пусть теперь первая позиция занята. При втором добавлении вероятность того, что будут занята позиция два, в два раза больше, чем вероятность попадания в остальные позиции и т.д.

Следовательно, имеет место тенденция возникновения все более длинных последовательностей занятых подряд позиций, что увеличивает и время поиска, и время добавления элементов.

3. Кроме того, так как в качестве структуры хранения используется массив, т.е. структура данных с фиксированным количеством элементов, то всегда существует вероятность переполнения.

4. Если массив заполнен почти полностью, то при поиске отсутствующего ключа в худшем случае будет просматриваться практически весь массив, что очень далеко от поставленной задачи – поиск за одно обращение.

Метод цепочек

Другой способ разрешения коллизий состоит в том, чтобы поддерживать *M* связанных списков, по одному на каждый возможный хеш-адрес. Таким образом, каждый список будет содержать все ключи с одинаковым хеш-адресом. Кроме того, необходимо иметь массив из *M* указателей на голову каждого из *M* списков.

Рассмотрим описанный выше метод на примере той же последовательности из 9 элементов (рис.2).

Если списки поддерживать упорядоченными, это уменьшит как время добавления элемента, так и время неудачного поиска. В этом случае все пустые указатели можно заменить указателями на вспомогательную запись с ключом, превышающим значение любого допустимого ключа.

Метод цепочек лишен большинства недостатков, присущих методу открытой адресации. Так использование списков практически не ограничивает количества добавляемых элементов.

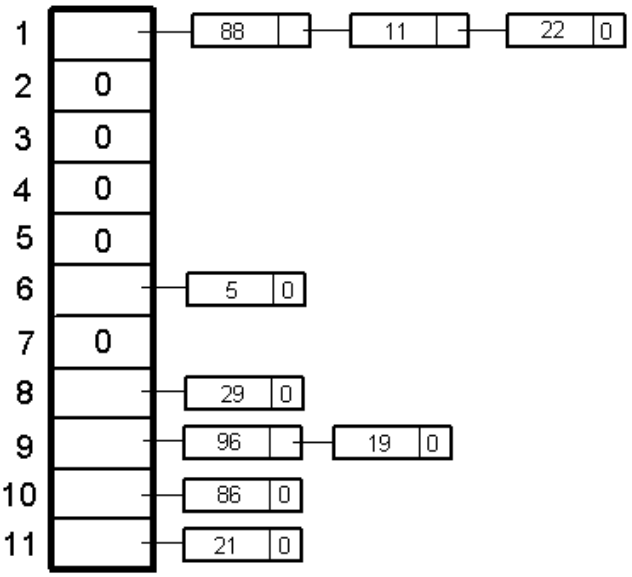

Рис. 2. Метод цепочек

Удаление элемента состоит в исключении узла из связного списка, что никак не влияет на эффективность как алгоритма поиска, так и алгоритма включения.

Основным недостатком метода цепочек является то, что для указателей требуется дополнительная память. Кроме того, если списки станут слишком длинными, то теряет смысл вся идея хеширования.

Задание на лабораторную работу

1. Написать функции получения хеш-адреса для рассмотренных выше методов деления, середины квадратов, свертывания, умножения и универсального хеширования.

2. Сгенерировать массив *M* из 1000 случайных чисел, значения которых находятся в диапазоне от 0 до 65000.

3. Используя метод цепочек, определить наиболее эффективную функцию хеширования. Для этого, последовательно применяя каждую из указанных в п.1 хеш-функций, заполнять цепочки ключами из массива *M*, подсчитывая количество коллизий. На вход каждой из хеш-функций должна подаваться **одна и та же последовательность ключей.**

4. Увеличить счетчик лучшей функции на 1.

5. Выполнить шаги 2, 3 и 4 *n* раз (*n≥100*) (количество итераций задается на форме).

6. Вывести счетчики для каждой функции хеширования.

7. Написать программу оценки временных характеристик поиска ключей для метода открытой адресации и метода цепочек. Для этого:

7.1. Используя наилучшую по эффективности функцию хеширования, заполнить 10000 одних и тех же случайных ключей соответствующие структуры хранения согласно методу открытой адресации и методу цепочек. Значения ключей должны находится в интервале от 0 до 10000;

7.2. Для 10000 одних и тех же случайных ключей определить время их поиска методом открытой адресации и методом цепочек. Значения ключей должны находится в интервале от 0 до 20000.

7.3. Для метода открытой адресации и метода цепочек вывести время поиска, среднее число сравнений и количество найденных ключей.

Контрольные вопросы

1. Когда возникает коллизия при хешировании?
2. Какие хеш-функции используются для формирования хеш-адресов?
3. В чем отличие метода открытой адресации от метода цепочек? Преимущества и недостатки каждого метода.
4. Перечислите ограничения на выбор делителя *m* в методе деления.

Лабораторная работа № 4
Цифровой поиск

Цель лабораторной работы – изучение методов организации хранения и поиска данных с помощью дерева цифрового поиска.

Теоретическая часть

Существует класс задач, которые оперируют с данными, представляющими собой множество слов некоторого языка. К таким задачам можно отнести задачи проверки орфографии, задачи перевода текста и т.п.

Одна из основных операций, которая часто используется в подобных задачах – поиск слова в множестве слов. Эффективность организации такого поиска во многом является определяющим для всей системы в целом.

Пусть имеется множество слов *M*. Необходимо определить, принадлежит ли слово *s* множеству *M*.

Поставленную выше задачу можно решить с помощью одного из уже рассмотренных нами алгоритмов поиска. Одним из наиболее эффективных, например, является алгоритм бинарного поиска. Однако, во-первых, множество *M* должно быть упорядочено, что является существенным ограничением. Во-вторых, время поиска будет зависеть от размера множества *M*, а не от искомого слова *s*.

Для рассмотренного выше класса задач используют еще один алгоритм поиска – **алгоритм цифрового поиска** [2].

Пусть множество слов *M* неупорядочено, т.е. для поиска приемлем только последовательный поиск. В этом случае время поиска значительно сократится, если разбить множество *M* на подмножества $M_А$, $M_Б$, ..., $M_Я$, в каждом из которых хранятся слова на соответствующую букву.

Тогда структура хранения будет выглядеть как двухуровневое дерево следующего вида (рис. 3).

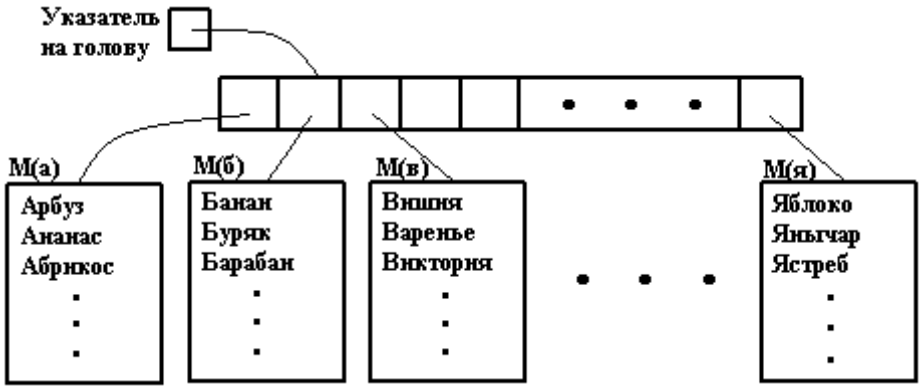

Рис.3. Двухуровневое дерево

На первом уровне этого дерева массив указателей, каждый из которых может быть либо пустым (указатели ъ, ь, й), либо содержать адрес соответствующего подмножества слов, находящегося на втором уровне дерева.

Рассмотренный выше подход можно развить дальше, организовав трехуровневое дерево, в котором первый уровень – указатели для первой буквы слова, второй – указатели для второй буквы слова, третий – множества слов, начинающихся на *АА, АБ, ..., ЯЯ*.

Ясно, что при этом количество элементов в подмножествах $M_{АА}$, $M_{АБ}$, ...,$M_{ЯЯ}$ значительно уменьшится, а, следовательно, уменьшится и время последовательного поиска.

Продолжая процесс увеличения уровней дерева, можно прийти к структуре дерева, в котором каждая буква слова будет располагаться на своем уровне. Такое дерево называется **деревом цифрового поиска**, или **бором.**

При такой организации хранения и поиска данных время поиска слова будет зависеть только от количества букв в этом слове.

Создание дерева цифрового поиска

Рассмотрим процесс формирования дерева цифрового поиска на примере. Пусть задано множество слов {воск, волк, ток, вар, толк, том, вилка, тик}. Построим дерево цифрового поиска (рис. 4).

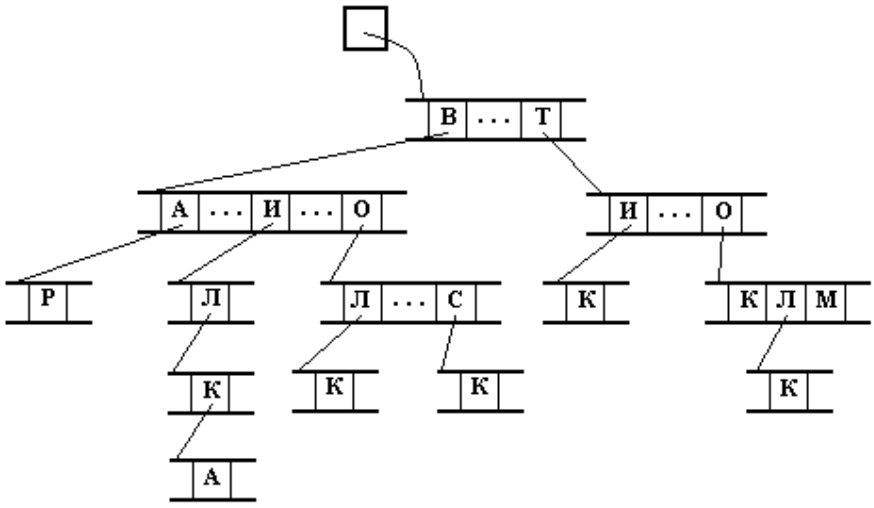

Рис. 4. Пример дерева цифрового поиска

Поиск в дереве цифрового поиска

Алгоритм поиска по бору достаточно очевиден. Первая буква искомого слово ищется на первом уровне дерева. Если соответствующий букве указатель пуст – поиск неудачен. В противном случае вторая буква слова таким же образом ищется в элементе дерева на втором уровне по найденному указателю и т.д.

Поиск заканчивается либо когда в очередном узле встретится пустой указатель (поиск неудачен), либо когда будет найдена последняя буква искомого слова.

При всей очевидности алгоритма поиска в нем есть несколько не всегда очевидных моментов. Так, если использовать рассмотренный выше алгоритм формирования бора, то в дереве на рисунке 4 будет найдено не только слово «волк», но и слово «вол», а также «во», «в» и другие подслова введенных слов, хотя мы их не вводили.

Для решения этой проблемы необходимо, чтобы каждое заносимое в бор слово имело символ завершения (символ, не встречающийся в словах бора). Таким символом может быть, например, символ «*».

Вторая проблема заключается в том, что при такой организации хранения данных очень расточительно используется память. Требуемую память можно сократить за счет увеличения времени выполнения, представляя каждую вершину бора в виде связанного списка (рисунок 5).

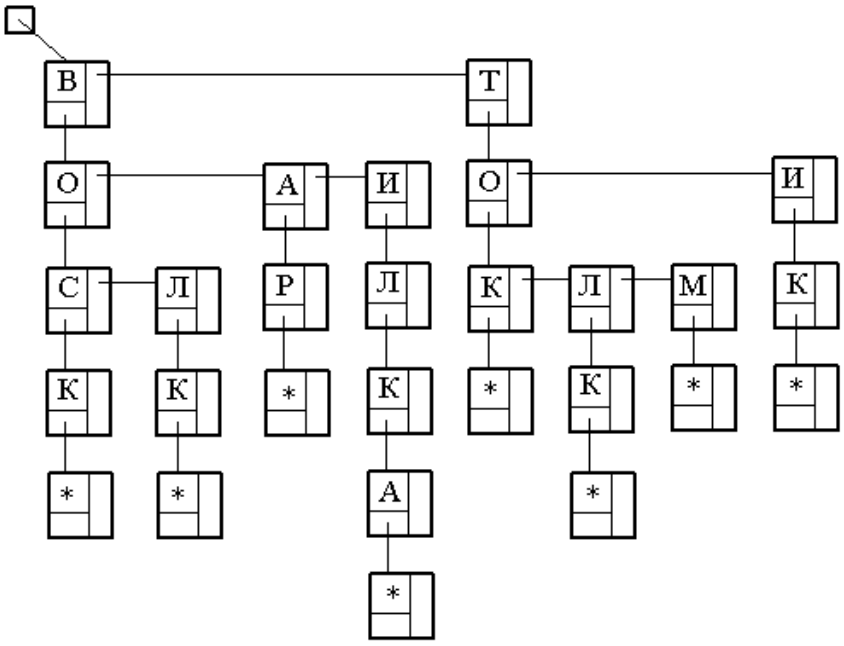

Рис. 5. Использование связного списка

Другой подход к использованию деревьев цифрового поиска предполагает, что только несколько уровней бора используются для поиска первых букв ключа, а для поиска оставшихся букв используется какая-нибудь другая структура типа линейного списка или бинарного дерева.

Другими словами, на практике можно использовать цифровой поиск как промежуточный вариант между рассмотренными нами на рисунках 3 и 4 структурами.

Задание на лабораторную работу

Написать программу, реализующую алгоритм цифрового поиска. Обязательно должны быть реализованы функции добавления нового слова и поиска слова в дереве. На форме должен присутствовать список введенных в дерево слов. Предусмотреть процедуру освобождения занимаемой деревом памяти.

Контрольные вопросы

1. Какова структура узла дерева цифрового поиска?
2. Какова структура дерева цифрового поиска?
3. Привести пример формирования дерева цифрового поиска.
4. Привести пример поиска в дереве цифрового поиска.

Лабораторная работа № 5
Алгоритмы внутренней сортировки

Цель лабораторной работы – сравнение и анализ основных методов упорядочения данных, расположенных в оперативной памяти.

Теоретическая часть

Сортировка – это процесс расстановки элементов в некотором порядке. Для человека, как правило, порядок данных весьма важен, так как позволяет выявлять важные свойства этих данных. Кроме того, упорядоченность данных во многих случаях позволяет использовать вычислительную технику более эффективно [5].

Сортировка двух записей состоит из сравнения их ключевых полей и определения, которое из них меньше. После этого записи переставляются так, что запись с меньшим ключом ставится перед записью с большим ключом.

При сравнении двух ключей можно использовать и символьное, и числовое, и буквенно-цифровое упорядочение.

Ниже при рассмотрении алгоритмов сортировки для простоты будем предполагать, что записи состоят только из одного поля, что не ограничивает область применения этих алгоритмов.

Методы сортировки делятся на внутренние и внешние [6].

Внутренние методы предполагают, что сортируемые данные целиком располагаются в оперативной памяти.

Внешние методы используются для сортировки файлов данных, которые слишком велики, чтобы полностью поместиться в оперативной памяти.

В данной лабораторной работе будут рассматриваться только алгоритмы внутренней сортировки.

Рассматриваемые ниже алгоритмы сортировки отличаются эффективностью своей работы. При определении эффективности того или иного алгоритма будут анализироваться количество необходимых сравнений ключей и количество перестановок элементов.

Эти значения являются функцией от n – количества сортируемых элементов.

Рассмотрим сначала три простых с точки зрения программирования метода сортировки: сортировку с помощью прямого обмена, сортировку с помощью прямого выбора и сортировку с помощью прямого включения. Все эти алгоритмы имеют сложность $O(n^2)$ [7].

Далее будет рассмотрен более эффективный алгоритм сортировки (сортировка методом Шелла), сложность которого оценивается как $O(n \cdot log(n))$, и алгоритм линейной сортировки сложности $O(3 \cdot n)$. Однако более эффективные алгоритмы сортировки, как правило, являются и более сложными для программирования, либо, как алгоритм линейной сортировки, имеют существенные ограничения на их использование.

Пусть необходимо отсортировать по возрастанию массив S, состоящий из n числовых элементов.

Сортировка с помощью прямого обмена

Сортировка с помощью прямого обмена (сортировка стандартным обменом, сортировка методом пузырька) перемещает один элемент массива *S* в соответствующую позицию при каждом просмотре.

При первом просмотре каждый элемент S_i сравнивается с элементом S_{i+1} $(1 \le i \le n-1)$ и при необходимости, если $S_i > S_{i+1}$, меняется с ним местами. В результате наибольший элемент помещается в последнюю позицию.

При втором просмотре выполняются те же действия, но уже не для *n*, а для $(n-1)$ первых элементов массива. В результате следующий по величине элемент перемещается в предпоследнюю позицию.

При третьем просмотре рассматриваются уже первые $(n-2)$ элемента массива. Таким образом, для выполнения сортировки требуется максимально *n* просмотров.

Во время каждого просмотра **необходимо фиксировать наличие обменов.** Если при очередном просмотре обменов не было, то массив уже упорядочен, сортировка заканчивается.

Рисунок 6 демонстрирует выполнение сортировки методом прямого обмена для массива из 10 элементов.

Исходный массив	29	68	52	12	66	89	57	20	97	21
Проход 1	29	52	12	66	68	57	20	89	21	97
Проход 2	29	12	52	66	57	20	68	21	89	97
Проход 3	12	29	52	57	20	66	21	68	89	97
Проход 4	12	29	52	20	57	21	66	68	89	97
Проход 5	12	29	20	52	21	57	66	68	89	97
Проход 6	12	20	29	21	52	57	66	68	89	97
Проход 7	12	20	21	29	52	57	66	68	89	97
Проход 8	12	20	21	29	52	57	66	68	89	97

Рис. 6. Метод прямого обмена

Число сравнений: минимальное – $(n-1)$, максимальное – $n^2/2$.

Число обменов: минимальное – 0, среднее – $(n^2/4)$, максимальное – $n^2/2$.

Следовательно, алгоритм сортировки методом прямого обмена имеет **сложность** $O(n^2)$ [1].

Прямой обмен с запоминанием позиции последней перестановки

Для улучшения рассмотренного выше алгоритма может быть учтено то обстоятельство, что за один просмотр входного множества на свое место могут «всплыть» не один, а два и более элементов. Это легко учесть, запоминая в каждом просмотре позицию последней перестановки и устанавливая эту позицию в качестве границы между упорядоченным и еще неупорядоченным подмножествами для следующего просмотра [8].

В качестве примера рассмотрим сортировку следующей последовательности (рис. 7).

Исходный массив	57	52	21	20	97	68	12	29	66	89
Проход 1	52	21	20	57	68	12	29	66	89	97
Проход 2	21	20	52	57	12	29	66	68	89	97
Проход 3	20	21	52	12	29	57	66	68	89	97
Проход 4	20	21	12	29	52	57	66	68	89	97
Проход 5	20	12	21	29	52	57	66	68	89	97
Проход 6	12	20	21	29	52	57	66	68	89	97
Проход 7	12	20	21	29	52	57	66	68	89	97

Рис. 7. Прямой обмен с запоминанием позиции последней перестановки

Шейкерная сортировка

Другим улучшением алгоритма сортировки методом прямого обмена является так называемая шейкерная сортировка (шейкером называются два накрывающих друг друга стакана, в которых встряхиванием вверх-вниз готовят коктейль) [1].

Рассмотрим шейкерную сортировку более подробно.

Пусть на вход алгоритма сортировки с помощью прямого обмена поступает практически отсортированная последовательность элементов

10 1 2 3 4 5 6 7 8 9.

Для упорядочения этой последовательности достаточно выполнения двух проходов.

Теперь пусть имеется другая, также практически отсортированная, последовательность элементов

2 3 4 5 6 7 8 9 10 1.

Для упорядочения этой последовательности элементов требуется уже 10 проходов, каждый из которых перемещает элемент 1 на одну позицию вправо.

Шейкерная сортировка предлагает чередовать направление последовательных просмотров. При первом прямом проходе (слева направо) наибольший элемент переместится в позицию *n*. При первом обратном проходе просматриваются элементы с *(n-1)*-го по 1-й, и наименьший элемент переместится в позицию 1.

При втором прямом проходе просматриваются элементы со 2-го по *(n-1)*-й, и наибольший элемент помещается в позицию *(n-1)*. При втором обратном проходе просматриваются элементы с *(n-2)*-го по 2-й, и наименьший элемент помещается в позицию 2 и т.д.

Так же, как и в предыдущем алгоритме, во время каждого просмотра необходимо фиксировать наличие обменов. Если при очередном просмотре обменов не было, то массив уже упорядочен, сортировка заканчивается.

Рисунок 8 демонстрирует выполнение шейкерной сортировки для массива из 10 элементов.

Исходный массив	29	68	52	12	66	89	57	20	97	21
Прямой проход 1	29	52	12	66	68	57	20	89	21	97
Обратный проход 1	12	29	52	20	66	68	57	21	89	97
Прямой проход 2	12	29	20	52	66	57	21	68	89	97
Обратный проход 2	12	20	29	21	52	66	57	68	89	97
Прямой проход 3	12	20	21	29	52	57	66	68	89	97
Обратный проход 3	12	20	21	29	52	57	66	68	89	97

Рис. 8. Шейкерная сортировка

Шейкерная сортировка не всегда лучше сортировки методом прямого обмена. В частности, если массив из n элементов упорядочен в обратном порядке, то шейкерная сортировка также потребует n проходов.

Эта сортировка позволяет сократить число сравнений, хотя порядком оценки по-прежнему остается n^2. Число же пересылок не меняется. Поэтому алгоритм шейкерной сортировки также имеет **сложность** $O(n^2)$.

Этот алгоритм весьма эффективен для задач восстановления упорядоченности, когда исходная последовательность уже была упорядочена, но подверглась не очень значительным изменениям. Упорядоченность в последовательности с одиночным изменением будет гарантированно восстановлена всего за два прохода.

Сортировка с помощью прямого выбора

Сортировка с помощью прямого выбора включает в себя следующие шаги.

1. Среди элементов массива S_1, ..., S_n выбрать элемент с наибольшим значением.
2. Найденный элемент поменять местами с элементом S_n.
3. Выполнить шаги 1 и 2 для оставшихся $n–1$ элементов, $n–2$ элементов и т.д. до тех пор, пока не останется один, самый маленький элемент.

Рисунок 9 демонстрирует выполнение сортировки с помощью прямого выбора для того же массива из 10 элементов.

Исходный массив	29	68	52	12	66	89	57	20	97	21
Проход 1	29	68	52	12	66	89	57	20	21	97
Проход 2	29	68	52	12	66	21	57	20	89	97
Проход 3	29	20	52	12	66	21	57	68	89	97
Проход 4	29	20	52	12	57	21	66	68	89	97
Проход 5	29	20	52	12	21	57	66	68	89	97
Проход 6	29	20	21	12	52	57	66	68	89	97
Проход 7	12	20	21	29	52	57	66	68	89	97
Проход 8	12	20	21	29	52	57	66	68	89	97

Проход 9	12	20	21	29	52	57	66	68	89	97

Рис. 9. Метод прямого выбора

Число сравнений не зависит от начального порядка элементов и равно $n\cdot(n-1)/2$.

Число перестановок: минимальное – 0, максимальное – $(n-1)$.

Алгоритм сортировки методом прямого выбора имеет **сложность** $O(n^2)$ [1].

Сортировка с помощью прямого выбора с одновременным поиском максимума и минимума

Довольно простая модификация рассмотренной выше сортировки предусматривает поиск в одном цикле просмотра входного множества сразу и **минимума**, и **максимума**, и обмен их с первым и с последним элементами множества соответственно. Хотя итоговое количество сравнений и пересылок в этой модификации не уменьшается, достигается экономия на количестве итераций внешнего цикла (рис. 10).

Исходный массив	29	68	52	12	66	89	57	20	97	21
Проход 1	12	68	52	29	66	89	57	20	21	97
Проход 2	12	20	52	29	66	21	57	68	89	97
Проход 3	12	20	21	29	66	52	57	68	89	97
Проход 4	12	20	21	29	57	52	66	68	89	97
Проход 5	12	20	21	29	52	57	66	68	89	97

Рис. 10. Прямой выбор с одновременным поиском максимума и минимума

Приведенные выше алгоритм сортировки прямым выбором практически **нечувствительны к исходной упорядоченности**. В любом случае поиск минимума (максимума) требует полного просмотра входного множества.

Сортировка с помощью прямого включения

Сортировка с помощью прямого включения (сортировка вставками) основана на последовательной вставке элементов в уже упорядоченную последовательность.

Алгоритм сортировки заключается в следующем. Сначала считается упорядоченным один первый элемент. Второй элемент вставляется в нужное место (т.е. перед первым или остается на своем месте).

Далее третий элемент включается в нужное место уже упорядоченной последовательности из двух элементов, за ним четвертый и т.д. до n-го.

Пусть необходимо вставить i-й ($i \geq 2$) элемент в уже упорядоченную последовательность из $(i-1)$-го элемента. Элемент S_i последовательно сравнивается с каждым элементом S_k $(i-1 \leq k \leq 0)$ и, либо вставляется на

свободное место, если $S_i \geq S_k$, либо элемент S_k сдвигается на одну позицию вправо, и процесс выполняется для элемента S_{k-1}.

Весь процесс вставки элемента заканчивается либо когда найден первый элемент S_k, такой, что $S_k \leq S_i$, либо достигнута левая граница упорядоченной последовательности (элемент S_i меньше всех элементов упорядоченной последовательности).

Очевидным улучшением описанного выше процесса является установка **барьера** в нулевом элементе массива **S ($S_0 = S_i$)**, что позволит избежать проверки на выход за левую границу массива.

Рисунок 11 демонстрирует выполнение сортировки с помощью прямого включения для того же массива из 10 элементов.

Исходный массив		29	68	52	12	66	89	57	20	97	21
	Барьер										
Проход 1	68	29	68	52	12	66	89	57	20	97	21
Проход 2	52	29	52	68	12	66	89	57	20	97	21
Проход 3	12	12	29	52	68	66	89	57	20	97	21
Проход 4	66	12	29	52	66	68	89	57	20	97	21
Проход 5	89	12	29	52	66	68	89	57	20	97	21
Проход 6	57	12	29	52	57	66	68	89	20	97	21
Проход 7	20	12	20	29	52	57	66	68	89	97	21
Проход 8	97	12	20	29	52	57	66	68	89	97	21
Проход 9	21	12	20	21	29	52	57	66	68	89	97

Рис. 11. Метод прямого включения

Если количество элементов в массиве велико, то другим улучшением описанного выше метода сортировки является использование двоичного поиска для определения места вставки очередного элемента. Такой алгоритм называется **методом сортировки с двоичным включением**.

Число сравнений: минимальное – *(n–1)*, среднее – $(n^2/4)$, максимальное – $(n^2/2)$.

Число перемещений: минимальное – 0, среднее – $(n^2/4)$, максимальное – $(n^2/2)$.

Алгоритм сортировки методом прямого включения имеет сложность $O(n^2)$ [1].

Сортировка методом Шелла

Все рассмотренные выше алгоритмы сортировки имеют сложность $O(n^2)$. Большинство из этих алгоритмов за одно сравнение перемещает элемент только на одну позицию. Для получения метода сортировки, существенно лучшего по сложности, чем рассмотренные выше алгоритмы, необходим некоторый механизм, с помощью которого элементы могли бы перемещаться за один раз на большие расстояния.

Такой метод был предложен Л. Шеллом и называется **сортировкой с убывающим шагом, сортировкой с помощью включения с уменьшающимися расстояниями** или просто **сортировкой Шелла.**

Как видно из названия, сортировка Шелла является расширением и улучшением сортировки с помощью прямого включения.

Суть метода заключается в следующем. Весь исходный массив S разбивается на группы с шагом h_1. В первую группу войдут элементы S_1, S_{h1+1}, S_{2h1+1}, ..., во вторую S_2, S_{h1+2}, S_{2h1+2}, ... и т.д. Затем каждая из h_1 групп сортируется методом прямого включения.

Далее описанный выше процесс повторяется для шага h_2 ($h_2<h_1$), h_3 ($h_3<h_2$) и т.д. Процесс заканчивается после того, как будет выполнена сортировка с шагом $h_t=1$.

Такой метод гарантированно дает упорядоченную последовательность элементов, так как на последнем шаге весь массив сортируется обычной сортировкой с помощью прямого включения.

Эффективность работы алгоритма достигается за счет того, что на каждом проходе либо сортируется относительно малое число элементов, либо элементы уже довольно хорошо упорядочены, и требуется сравнительно немного перестановок.

Однако эффективность алгоритма во многом зависит от выбора последовательности $h_1, h_2, ..., h_t$ ($h_t=1$, $h_{i+1}<h_i$).

Н. Вирт предлагает следующую формулу для вычисления количества проходов t и последовательности шагов $h_1, ..., h_t$:

$$t = \lfloor \log_2 n \rfloor - 1,\ h_t = 1,\ h_{k-1} = 2h_k + 1,$$

которая дает следующие значения $h_t=1$, $h_{t-1}=3$, $h_{t-2}=7$, $h_{t-3}=15$, $h_{t-4}=31$ и т.д. [1].

Для массива из 20 элементов $t=3$, $h_1=7$, $h_2=3$, $h_3=1$. Работа алгоритма рассмотрена на примере (рисунок 12). На рисунке закрашены элементы массива, составляющие одну группу.

Исходный массив	19	10	68	99	55	11	68	30	56	83	8	91	35	63	7	67	7	30	75	96
Индексы	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Проход 1	7	10	7	8	55	11	63	19	56	68	30	75	35	68	30	67	83	99	91	96
Проход 2	7	10	7	8	19	11	35	30	30	63	55	56	67	68	75	68	83	99	91	96
Проход 3	7	7	8	10	11	19	30	30	35	55	56	63	67	68	68	75	83	91	96	99

Рис. 12. Сортировка методом Шелла

Алгоритм сортировки Шелла имеет **сложность** O(n·log n), что значительно лучше рассмотренных выше алгоритмов.

Линейная сортировка

Если значение каждого элемента массива меньше или равно количеству элементов массива, то в данном случае наиболее эффективным и по

быстродействию, и по простоте алгоритма является алгоритм линейной сортировки. Данный алгоритм заключается в следующем.

Пусть дан массив *A(n)* и значение каждого элемента в массиве *A* не превышает значение *m (m≤n).*

1. Обнулим все элементы вспомогательного массива *B(m).*

2. Для каждого элемента массива *A* выполнить следующие действия. Если $A_i=k$, то $B_k=B_k+1$.

3. Для каждого элемента массива *B* выполнить следующие действия. Если $B_i=f$, то в массив *A* записать *f* раз значение *i*.

На рисунке 13 приведен пример работы алгоритма линейной сортировки.

Массив A	8	3	5	3	9	1	8	4	3	1
Массив B	2	0	3	1	1	0	0	2	1	0
Отсортированный массив A	1	1	3	3	3	4	5	8	8	9

Рис. 13. Линейная сортировка

Как видно из алгоритма, его **сложность определяется как O(3·n).**

Задание на лабораторную работу

Написать программу, состоящую из следующих пунктов.

1. Сортировка с помощью прямого обмена.
2. Сортировка с помощью прямого выбора.
3. Сортировка с помощью прямого включения.
4. Сортировка Шелла.
5. Линейная сортировка.
6. Характеристики сортировок.

В пунктах 1–5 продемонстрировать соответствующую сортировку для массива из 10 элементов. Вывести на экран исходный массив и промежуточные результаты каждого прохода. Для сортировки Шелла использовать массив из 20-ти элементов.

В пункте 6 для заданного в диалоге количества элементов массива подсчитать время сортировки, число сравнений и перестановок (сдвигов) для каждого из алгоритмов.

Контрольные вопросы

1. Какова сложность каждого из рассмотренных выше алгоритмов сортировки?

2. Определить среднее число обрабатываемых элементов для каждого из алгоритмов сортировки. Эту оценку можно получить из произведения среднего числа проходов и среднего количества элементов, обрабатываемых на каждом проходе.

Лабораторная работа № 6
Внешняя сортировка простым слиянием

Цель лабораторной работы – сравнение и анализ алгоритмов внешней сортировки простым слиянием.

Теоретическая часть

Рассмотренные в предыдущей лабораторной работе алгоритмы внутренней сортировки невозможно применять для упорядочения данных, которые из-за своего размера не помещаются в оперативную память машины и находятся на внешних запоминающих устройствах.

Пусть данные хранятся в последовательном файле, характерной особенностью которого является то, что в каждый момент времени доступна только одна компонента.

Рассмотрим алгоритмы сортировки таких файлов.

Двухфазная сортировка простым слиянием

Слияние означает объединение двух или более последовательностей в одну упорядоченную последовательность с помощью повторяющегося выбора из доступных в данный момент элементов.

Одна из сортировок на основе слияния называется **сортировкой простым слиянием**. Она выполняется следующим образом.

1. Файл *A* разбивается на два файла *B* и *C*. При этом первый элемент файла *A* помещается в файл *B*, второй элемент в файл *C*, третий – в файл *B*, четвертый – в файл *C* и т.д.

2. Файлы *B* и *C* сливаются в файл *A*, при этом одиночные элементы образуют упорядоченные пары.

3. Полученный файл *A* вновь обрабатывается, как указано в пунктах 1 и 2. При этом упорядоченные пары переходят в упорядоченные четверки.

4. Повторяя предыдущие шаги, сливаем четверки в восьмерки и т.д., каждый раз удваивая длину слитых последовательностей до тех пор, пока не будет упорядочен целиком весь файл [1].

Рассмотрим работу алгоритма на примере (рис. 14). Пусть задан файл *A*:

A – 55 12 87 76 98 24 84 27
B – 55 87 98 84
C – 12 76 24 27
A – 12 55 76 87 24 98 27 84
B – 12 55 24 98
C – 76 87 27 84
A – 12 55 76 87 24 27 84 98
B – 12 55 76 87
C – 24 27 84 98
A – 12 24 27 55 76 84 87 98

Рис. 14. Двухфазная сортировка простым слиянием

Поскольку на каждом проходе размерность серии удваивается, то сортировка заканчивается, когда $p \geq n$, где p – размер серии, а n – размер исходного файла.

Количество проходов: $k = [\log_2 n]$. Файлы A, B и C будут $\log_2 n$ раз прочитаны и столько же раз записаны. По определению при каждом проходе все множество из n элементов копируется ровно 1 раз, следовательно, общее **число пересылок** $M = n \cdot [\log_2 n]$.

Число сравнений C еще меньше, чем M, т.к. при копировании остатка последовательности сравнения не производятся.

Поскольку сортировка слиянием использует внешнюю память, то временные затраты на операцию пересылки на несколько порядков превышают временные затраты на операции сравнения.

Действия по однократной обработке всего множества данных называются **фазой**. Рассмотренная выше сортировка состоит из фазы разделения и фазы слияния, поэтому она называется **двухфазной сортировкой простым слиянием.**

Однофазная сортировка простым слиянием

Фаза разделения файла A на два файла B и C не относится к сортировке. Она непродуктивна, хотя и занимает половину всех операций по переписи.

Очевидным улучшением (по быстродействию, но не по занимаемой памяти) рассмотренного выше алгоритма является **объединение фазы разделения с фазой слияния.**

Вместо слияния в один файл результаты слияния необходимо сразу распределять по двум файлам, которые станут исходными для последующего прохода.

Такая сортировка называется **однофазной сортировкой простым слиянием.** Очевидно, что для такой сортировки требуется уже не два, а четыре дополнительных файла.

Рассмотрим выполнение однофазной сортировки на примере (рис. 15). Пусть задан файл A:

A – 55 12 87 76 98 24 84 27
B – 55 87 98 84
C – 12 76 24 27
D – 12 55 24 98
E – 76 87 27 84
B – 12 55 76 87
C – 24 27 84 98
A – 12 24 27 55 76 84 87 98

Рис. 15. Однофазная сортировка простым слиянием

Задание на лабораторную работу

Написать программу, состоящую из следующих пунктов.

1. Двухфазная сортировка простым слиянием.
2. Однофазная сортировка простым слиянием.
3. Характеристики сортировок.

В пунктах 1–2 продемонстрировать соответствующую сортировку для файлов из 15 элементов. Вывести на экран исходный файл и промежуточные результаты каждой фазы сортировки.

В пункте 3 для заданного в диалоге количества элементов файла A подсчитать время сортировки, количество чтений из файла, количество записей в файл и количество сравнений. Результаты сортировок поместить в отдельные файлы B и C.

Предусмотреть в программе возможность вывода на форму последовательности значений заданного в диалоге файла (A, B или C) в интервале от t до k, где t<k и k<n.

Контрольные вопросы

1. Чем отличаются алгоритмы внутренней сортировки от алгоритмов внешней сортировки?
2. Что такое слияние последовательностей?
3. Дайте определение фазы.
4. Чем отличается однофазное слияние от двухфазного?

Лабораторная работа № 7

Внешняя сортировка естественным слиянием

В случае прямого слияния мы не получаем никакого преимущества, если данные уже являются частично упорядоченными. Размер сливаемых при каждом проходе последовательностей не зависит от существования более длинных уже упорядоченных последовательностей, которые можно было бы просто объединить.

Сортировка, при которой всегда сливаются две самые длинные из возможных последовательностей, называется **естественным слиянием** [1].

Максимальную упорядоченную последовательность будем называть **серией**.

Пусть имеется начальный файл *A*. Каждый проход состоит из фазы распределения серий из файла *A* поровну в файлы *B* и *C* и фазы слияния, объединяющей серии из файлов *B* и *C* в файл *A*.

Процесс сортировки заканчивается, как только в файле *A* останется только одна серия.

Рассмотрим сортировку естественным слиянием на примере (рис. 16). Пусть задан файл A:

A – 17 31 05 59 13 41 43 76 11 23 29 47 03 07 71 02 19 57 37 61
B – 17 31 13 41 43 76 03 07 71 37 61
C – 05 59 11 23 29 47 02 19 57
A – 05 17 31 59 11 13 23 29 41 43 47 76 02 03 07 19 57 71 37 61
B – 05 17 31 59 02 03 07 19 57 71
C – 11 13 23 29 41 43 47 76 37 61
A – 05 11 13 17 23 29 31 41 43 47 59 76 02 03 07 19 37 57 61 71
B – 05 11 13 17 23 29 31 41 43 47 59 76
C – 02 03 07 19 37 57 61 71
A – 02 03 05 07 11 13 17 19 23 29 31 37 41 43 47 57 59 61 71 76

Рис. 16. Двухфазная сортировка естественным слиянием

На основе рассмотренного выше алгоритма легко получить **алгоритм однофазной сортировки естественным слиянием** с четырьмя файлами.

Задание на лабораторную работу

Написать программу, состоящую из следующих пунктов.

1. Двухфазная сортировка естественным слиянием.
2. Однофазная сортировка естественным слиянием.
3. Характеристики сортировок.

В пунктах 1–2 продемонстрировать соответствующую сортировку для файлов из 15 элементов. Вывести на экран исходный файл и промежуточные результаты каждой фазы сортировки.

В пункте 3 для заданного в диалоге количества элементов файла A подсчитать время сортировки, количество чтений из файла, количество записей в файл и количество сравнений. Результаты сортировок поместить в отдельные файлы B и C.

Предусмотреть в программе возможность вывода на форму последовательности значений заданного в диалоге файла (A, B или C) в интервале от t до k, где t<k и k<n.

Контрольные вопросы

1. Чем отличаются алгоритм сортировки простым слиянием от алгоритма сортировки естественным слиянием?
2. Дайте определение серии.
3. Может ли количество серий в файлах отличаться более чем на единицу?

Лабораторная работа № 8

Внутренняя сортировка с внешним слиянием

В рассмотренных выше алгоритмах внешней сортировки совсем не используется преимущество современных ПЭВМ – наличие довольно большой оперативной памяти.

Рассмотрим алгоритм сортировки, который использует как внутреннюю сортировку одним из изученных ранее способов, так и слияние, присущее внешним сортировкам.

Пусть предназначенный для сортировки файл A содержит 5000 записей $R_1...R_{5000}$. Пусть во внутренней памяти машины помещается одновременно только 1000 записей.

В этом случае необходимо отсортировать во внутренней памяти каждый из пяти подфайлов $R_1...R_{1000}$, $R_{1001}...R_{2000}$, ..., $R_{4001}...R_{5000}$ по отдельности, а затем слить полученные подфайлы.

Назовем описанный выше алгоритм **алгоритмом внутренней сортировки с последующим внешним слиянием.**

Рассмотрим работу описанного выше алгоритма на примере (рис. 17). Как и в предыдущих алгоритмах, нам потребуется четыре дополнительных файла.

A – $R_1...R_{1000}$ $R_{1001}...R_{2000}$ $R_{2001}...R_{3000}$ $R_{3001}...R_{4000}$ $R_{4001}...R_{5000}$
B – $R_1...R_{1000}$ $R_{2001}...R_{3000}$ $R_{4001}...R_{5000}$
C – $R_{1001}...R_{2000}$ $R_{3001}...R_{4000}$
D – $R_1...R_{2000}$ $R_{4001}...R_{5000}$
E – $R_{2001}...R_{4000}$
B – $R_1...R_{4000}$
C – $R_{4001}...R_{5000}$
D – $R_1...R_{5000}$
E – пусто

Рис. 17. Внутренняя сортировка с внешним слиянием

Процесс продолжается до тех пор, пока не останется один упорядоченный файл.

Задание на лабораторную работу

Написать программу, состоящую из следующих пунктов.

1. Внутренняя сортировка с внешним слиянием.
2. Характеристики сортировки.

В пункте 1 продемонстрировать соответствующую сортировку для файлов из 15 элементов, объем оперативной памяти – 3 элемента. Вывести на экран исходный файл и промежуточные результаты каждой фазы сортировки.

В пункте 2 для заданного в диалоге количества элементов файла A подсчитать время сортировки, количество чтений из файла, количество записей в файл и количество сравнений для сортировок, в которых объем оперативной памяти варьируется в интервале от 1 до 10 % от объема исходного файла. Результаты сортировок поместить в отдельные файлы B1÷B10.

Предусмотреть в программе возможность вывода на форму последовательности значений заданного в диалоге файла (A, B1÷B10) в интервале от t до k, где $t<k$ и $k<n$.

Контрольные вопросы

1. В чем принципиальное отличие алгоритма внутренней сортировки с внешним слиянием от рассмотренных ранее алгоритмов простым и естественным слиянием?

2. Определите сложность алгоритма внутренней сортировки с внешним слиянием.

Лабораторная работа № 9

Сортировка методом поглощения

Пусть, как и во внутренней сортировке с внешним слиянием мы можем использовать внутреннюю память компьютера. Тогда в память из конца исходного файла *A* считываем первую порцию данных, упорядочиваем и записываем на то же место в исходный файл.

Далее для всех последующих частей файла *A* справа налево выполняем следующие действия. Считываем очередную порцию данных в оперативную память, сортируем, сливаем с уже отсортированной частью файла *A* (отсортированная часть файла поглощает очередную порцию еще не отсортированных данных), записывая результат на место поглощаемой части и далее до конца файла.

Таким образом, еще не упорядоченная часть файла как бы поглощается часть за частью.

Если при слиянии взяты все записи поглощаемой части, поглощение завершается. Слияние также завершается, если исчерпана ранее упорядоченная часть. Поглощение ею очередной части произошло.

Рассмотрим пример сортировки методом поглощения (рис. 18). Пусть в основной памяти одновременно можно обработать 3 элемента.

A	–	34 78 54 32 01 67 23 29 87 23 56 12 04 67 22
ОП	–	04 22 67
A	–	34 78 54 32 01 67 23 29 87 23 56 12 04 22 67
ОП	–	12 23 56
A	–	34 78 54 32 01 67 23 29 87 04 12 22 23 56 67
ОП	–	23 29 87
A	–	34 78 54 32 01 67 04 12 22 23 23 29 56 67 87
ОП	–	01 32 67
A	–	34 78 54 01 04 12 22 23 23 29 32 56 67 67 87
ОП	–	34 54 78
A	–	01 04 12 22 23 23 29 32 34 54 56 67 67 78 87

Рис. 18. Сортировка методом поглощения

Задание на лабораторную работу

Написать программу, состоящую из следующих пунктов.

1. Сортировка методом поглощения.
2. Характеристики сортировки.

В пункте 1 продемонстрировать соответствующую сортировку для файлов из 15 элементов, объем оперативной памяти – 3 элемента. Вывести на экран исходный файл и промежуточные результаты каждой фазы сортировки.

В пункте 2 для заданного в диалоге количества элементов файла A подсчитать время сортировки, количество чтений из файла, количество записей в файл и количество сравнений для сортировок, в которых объем оперативной памяти варьируется в интервале от 1 до 10 % от объема исходного файла. Результаты сортировок поместить в отдельные файлы B1÷B10.

Предусмотреть в программе возможность вывода на форму последовательности значений заданного в диалоге файла (A, B1÷B10) в интервале от t до k, где t<k и k<n.

Контрольные вопросы

1. В чем принципиальное отличие алгоритма сортировки методом поглощения от рассмотренных ранее алгоритмов простым и естественным слиянием?
2. Определите сложность алгоритма сортировки методом поглощения.

Лабораторная работа № 10
Анализ и сравнение алгоритмов внешней сортировки

Задание на лабораторную работу

Разработать программный стенд для анализа алгоритмов внешней сортировки данных, который позволяет для заданных в программе исходных данных проанализировать практическую эффективность наиболее известных алгоритмов решения данной задачи.

Входными данными стенда являются:

1. Входные параметры обрабатываемых данных
    a. размер исходного файла;
    b. размер доступной оперативной памяти;
2. Перечень сравниваемых алгоритмов
    a. двухфазная сортировка простым слиянием;
    b. однофазная сортировка простым слиянием;
    c. двухфазная сортировка естественным слиянием;
    d. однофазная сортировка естественным слиянием;
    e. внутренняя сортировка с внешним слиянием;
    f. сортировка методом поглощения.

Выходные данные

1. Табличный анализ выбранных алгоритмов для заданных исходных данных;
    a. время сортировки;
    b. количество чтений из файла;
    c. количество записей в файл;

d. количество сравнений

2. Графическое представление характеристик выбранных алгоритмов для заданных исходных данных.
3. Предусмотреть возможность выбора из исходного перечня любого сочетания сравниваемых алгоритмов

2. Тематика курсовых работ

Задание 1. Анализ и сравнение функций хеширования.

Задан массив из n элементов (n≥1000). Анализ заключается в подсчете и сравнении количества коллизий. Данные отобразить в табличной и графической формах.

1. Анализ хеш-функции вычисления хеш-адреса методом деления
   1.1. m – простое число, наиболее близкое к количеству требуемых адресов;
   1.2. m – нечетное число, которое хотя и не является простым, но не содержит простых сомножителей, меньших 20, наиболее близкое к количеству требуемых адресов;
   1.3. m – нечетное число, наиболее близкое к количеству требуемых адресов;
   1.4. m – четное число, наиболее близкое к количеству требуемых адресов;
2. Анализ хеш-функции вычисления хеш-адреса методом середины квадрата. Если суммарное количество цифр, которые необходимо отбросить для получения середины, нечетно, то возможны два варианта:
   2.1. Обрезать больше справа;
   2.2. Обрезать больше слева.
3. Анализ хеш-функции вычисления хеш-адреса методом свертывания.
   3.1. Свертывание слева направо;
   3.2. Свертывание справа налево.
4. Анализ хеш-функции вычисления хеш-адреса методом умножения. Проанализировать работу хеш-функции для различных значений A.
   4.1. A=(√5-1)/2≈0,6180339887;
   4.2. A=0.61;
   4.3. Ваш вариант;
   4.4. Ваш вариант.
5. Анализ хеш-функции вычисления хеш-адреса методом преобразования систем счисления.
   5.1. 11-тиричная система счисления;
   5.2. 13-тиричная система счисления;
   5.3. Ваш вариант;
   5.4. Ваш вариант.

6. Для каждой из рассмотренных выше хеш-функций выбрать наилучший по числу коллизий вариант хеширования. Для заданного массива исходных ключей из n элементов (n≥10000) выполнить сравнение пяти перечисленных выше функций хеширования между собой по числу коллизий и времени заполнения массива.

Задание 2. Разработка библиотеки функций для поддержки работы со сбалансированными деревьями.

1. Должны быть разработаны следующие функции:
   - 1.1. Создать пустое сбалансированное дерево;
   - 1.2. Выполнить LL-поворот;
   - 1.3. Выполнить RR-поворот;
   - 1.4. Выполнить LR-поворот;
   - 1.5. Выполнить RL-поворот;
   - 1.6. Добавить новое значение;
   - 1.7. Удалить значение;
   - 1.8. Модифицировать значение;
   - 1.9. Поиск значения;
   - 1.10. Обход слева направо;
   - 1.11. Визуализация сбалансированного дерева.
2. Выполнить оценку временных характеристик работы алгоритма оптимального бинарного поиска и алгоритма поиска в сбалансированном дереве. Данные отобразить в табличной и графической формах.
3. Выполнить оценку временных характеристик работы алгоритма сортировки прямым включением и сортировки с помощью сбалансированного дерева. Данные отобразить в табличной и графической формах.

Задание 3. Сравнение и анализ неискажающих алгоритмов сжатия

1. Найти теорию по трем наиболее известным алгоритмам неискажающего сжатия. Запрограммировать данные методы сжатия.
2. Сравнить по времени работу данных методов сжатия при сжатии:
   - 2.1. Текстовых файлов;
   - 2.2. Числовых файлов;
   - 2.3. Смешанных файлов;
   - 2.4. Графических файлов;
   - 2.5. Аудиофайлов;
   - 2.6. Видеофайлов.
3. Результаты сравнения отобразить в табличной и графической формах.

Задание 4. Сравнение и анализ искажающих алгоритмов сжатия

1. Найти теорию по трем наиболее известным алгоритмам искажающего сжатия. Запрограммировать данные методы сжатия.
2. Сравнить по времени работу данных методов сжатия при сжатии:
   - 2.1. Текстовых файлов;

2.2. Числовых файлов;
2.3. Смешанных файлов;
2.4. Графических файлов;
2.5. Аудиофайлов;
2.6. Видеофайлов.

3. Результаты сравнения отобразить в табличной и графической формах.

Задание 5. Разработка библиотеки функций для поддержки работы с B-деревьями.

1. Должны быть разработаны следующие функции
   1.1. Создать пустое B-дерево;
   1.2. Разбиение страницы на две;
   1.3. Слияние двух страниц в одну;
   1.4. Добавить новое значение;
   1.5. Удалить значение;
   1.6. Модифицировать значение;
   1.7. Поиск значения;
   1.8. Визуализация B-дерева.
2. Выполнить оценку временных характеристик работы алгоритма оптимального бинарного поиска и алгоритма поиска в B-дереве. Данные отобразить в табличной и графической формах.

Задание 6. Сравнение и анализ методов поддержки самоорганизующихся таблиц

1. Задан исходный массив из n элементов ($n \geq 1000$), массив вероятностей обращения к каждому элементу этого массива и поисковый массив (исходный массив, расположенный в порядке убывания вероятностей поиска элементов). Рассчитать среднее число сравнений:
   1.1. Для исходного массива (должно быть $\approx n2/2$);
   1.2. Для массива после применения метода перемещения в начало;
   1.3. Для массива после применения метода транспозиции;
   1.4. Для массива после применения смешанного метода.
2. Данные отобразить в табличной и графической формах.

Задание 7. Сравнение и анализ алгоритмов поиска символьной информации.

1. На основе заданной совокупности символьных ключей M1(n, k), где n – количество ключей, k – максимальное число символов ключа, сформировать три следующие структуры хранения:
   1.1. Дерево цифрового поиска;
   1.2. Используя хещ-функцию свертывания организовать структуру хранения исходной совокупности ключей методом цепочек;
   1.3. Упорядоченный массив ключей.

2. Сформировать массив поисковых ключей M2(2·n, k) добавив к массиву M1 n новых ключей. Для различных n и k (не менее 10-ти выборок) рассчитать:

2.1. Время поиска ключей из массива M2 в трех структурах хранения;

2.2. Среднее число сравнений для каждой из трех структур хранения;

2.3. Количество найденных ключей для каждой из трех структур хранения.

3. Данные отобразить в табличной и графической формах.

Задание 8. Сравнение и анализ алгоритмов сортировки методом прямого обмена.

1. Запрограммировать следующие алгоритмы сортировки:

1.1. Классический прямой обмен;

1.2. Прямой обмен с запоминанием позиции последней перестановки;

1.3. Шейкерная сортировка;

1.4. Шейкерная сортировка с запоминанием позиций последней перестановки

2. Выполнить сравнение времени сортировки перечисленными выше методами сортировки:

2.1. Неупорядоченного массива;

2.2. Упорядоченного в обратной последовательности массива;

2.3. Частично упорядоченного массива (процент упорядоченности задается на форме)

3. Данные отобразить в табличной и графической формах.

Задание 9. Сравнение и анализ алгоритмов сортировки методом прямого выбора.

1. Запрограммировать следующие алгоритмы сортировки:

1.1. Классический прямой выбор. Сортировка массива.

1.2. Прямой выбор с одновременным поиском максимума и минимума. Сортировка массива.

1.3. Классический прямой выбор. Сортировка односвязного списка.

1.4. Прямой выбор с одновременным поиском максимума и минимума. Сортировка односвязного списка.

2. Выполнить сравнение времени сортировки перечисленными выше методами сортировки:

2.1. Неупорядоченной последовательности;

2.2. Упорядоченной последовательности;

2.3. Упорядоченной в обратном порядке последовательности;

2.4. Частично упорядоченной последовательности (процент упорядоченности задается на форме)

3. Данные отобразить в табличной и графической формах.

Задание 10. Сравнение и анализ алгоритмов сортировки методом прямого включения.

1. Запрограммировать следующие алгоритмы сортировки:
   1.1. Классическое прямое включение;
   1.2. Сортировка с двоичным включением;
   1.3. Сортировка прямым включением со смешанной стратегией (до k элементов – последовательный поиск, более k элементов – двоичный).
2. Опытным путем определить k, либо сделать вывод о том, что использование на первых шагах последовательного поиска не улучшает временные характеристики сортировки с двоичным включением.
3. Выполнить сравнение времени сортировки перечисленными выше методами сортировки:
   3.1. Неупорядоченного массива;
   3.2. Упорядоченного массива;
   3.3. Упорядоченного в обратной последовательности массива;
   3.4. Частично упорядоченного массива (процент упорядоченности задается на форме)
4. Данные отобразить в табличной и графической формах.

Задание 11. Сравнение и анализ алгоритмов сортировки методом прямого включения.

1. Запрограммировать следующие алгоритмы сортировки:
   1.1. Классическое прямое включение. Сортировка массива.
   1.2. Классическое прямое включение. Сортировка односвязного списка.
   1.3. Классическое прямое включение. Сортировка двусвязного списка.
2. Выполнить сравнение времени сортировки перечисленными выше методами сортировки:
   2.1. Неупорядоченной последовательности;
   2.2. Упорядоченной последовательности;
   2.3. Упорядоченной в обратном порядке последовательности;
   2.4. Частично упорядоченной последовательности (процент упорядоченности задается на форме)
3. Данные отобразить в табличной и графической формах.

Задание 12. Сравнение и анализ алгоритмов сортировки методом Шелла.

1. Запрограммировать следующие алгоритмы сортировки
   1.1. Сортировка Шелла. Алгоритм вычисления шага – Вирт;
   1.2. Сортировка Шелла. Алгоритм вычисления шага – Кнут;

1.3. Сортировка Шелла. Алгоритм вычисления шага – Ваш вариант;

1.4. Сортировка Шелла. Алгоритм вычисления шага – Ваш вариант.

2. Выполнить сравнение времени сортировки перечисленными выше методами сортировки:

2.1. Неупорядоченной последовательности;

2.2. Упорядоченной последовательности;

2.3. Упорядоченной в обратном порядке последовательности;

2.4. Частично упорядоченной последовательности (процент упорядоченности задается на форме)

3. Данные отобразить в табличной и графической формах.

Задание 13. Сравнение и анализ алгоритмов сортировки односвязных списков.

1. Запрограммировать следующие алгоритмы сортировки односвязного списка

1.1. Прямой обмен;

1.2. Прямой выбор;

1.3. Прямое включение.

2. Выполнить сравнение времени сортировки перечисленными выше методами сортировки:

2.1. Неупорядоченной последовательности;

2.2. Упорядоченной последовательности;

2.3. Упорядоченной в обратном порядке последовательности;

2.4. Частично упорядоченной последовательности (процент упорядоченности задается на форме)

3. Данные отобразить в табличной и графической формах.

Задание 14. Разработка библиотеки функций для поддержки работы с бинарными деревьями поиска.

1. Должны быть разработаны следующие функции:

1.1. Создать пустое бинарное дерево поиска;

1.2. Добавить новое значение;

1.3. Удалить значение;

1.4. Модифицировать значение;

1.5. Поиск значения

1.6. Обход слева направо

1.7. Визуализация бинарного дерева поиска.

2. Выполнить оценку временных характеристик работы алгоритма оптимального бинарного поиска и алгоритма поиска в бинарном дереве поиска. Данные отобразить в табличной и графической формах.

3. Выполнить оценку временных характеристик работы алгоритма сортировки прямым обменом и сортировки с помощью бинарного дерева поиска. Данные отобразить в табличной и графической формах.

Задание 15. Сравнение и анализ алгоритмов внешней сортировки

1. Запрограммировать следующие алгоритмы внешней сортировки:
   - 1.1. Сортировка простым слиянием.
   - 1.2. Сортировка естественным слиянием.
   - 1.3. Сортировка методом поглощения.
   - 1.4. Сортировка многофазным слиянием.
2. Выполнить сравнение времени сортировки перечисленными выше методами сортировки. Данные отобразить в табличной и графической формах.

Задание 16. Исследование алгоритма внешней сортировки многофазным слиянием

1. Запрограммировать алгоритм внешней сортировки многофазным слиянием.
2. Выполнить сравнение характеристик данного алгоритма сортировки при m=3÷10, где m – число вспомогательных файлов.
3. Данные отобразить в табличной и графической формах.

СПИСОК ЛИТЕРАТУРЫ

1. Вирт Н. Алгоритмы и структуры данных. Новая версия для Оберона. Изд-во ДМК Пресс, 2010, 272 с.

2. Кнут Д. Искусство программирования для ЭВМ. Сортировка и поиск: Пер. с англ. – М.: Мир, 1978. Т. 3.

3. Лэнгсан Й., Огенстайн М., Тененбаум А. Структуры данных для персональных ЭВМ. –М.: Мир, 1989.

4. Баженова М.М., Москвина Л.А., Поттосин И.В. Практическое программирование. Структуры данных и алгоритмы. – М.: Логос, 2001.

5. Лорин Г. Сортировка и системы сортировки. -М.: Наука, 1983.

6. Флорес И. Структуры и управление данными. – М. Финансы и статистика. 1982.

7. Ахо Х., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. –М.: Мир, 1979.

8. Мейер Бертран. Инструменты, алгоритмы и структуры данных. Курс лекций. Издательство: НОУ Интуит, 2016. http://www.book.ru/

СОДЕРЖАНИЕ