МИНОБРНАУКИ РОССИИ

Орский гуманитарно-технологический институт (филиал)
федерального государственного бюджетного образовательного учреждения
высшего образования «Оренбургский государственный университет»
(Орский гуманитарно-технологический институт (филиал) ОГУ)

Кафедра программного обеспечения

Методические указания
для обучающихся по освоению дисциплины

«Б.1.В.ОД.4 Структуры и алгоритмы обработки данных»

Уровень высшего образования

БАКАЛАВРИАТ

Направление подготовки

09.03.01 Информатика и вычислительная техника

(код и наименование направления подготовки)

Программное обеспечение средств вычислительной техники и автоматизированных систем

(наименование направленности (профиля) образовательной программы)

Тип образовательной программы

Программа академического бакалавриата

Квалификация

Бакалавр

Форма обучения

Очная

Год начала реализации программы (набора)

2014, 2015, 2016, 2017

г. Орск 2017

Методические указания для обучающихся по освоению дисциплины «Б.1.В.ОД.4 Структуры и алгоритмы обработки данных» предназначены для обучающихся очной формы обучения направления подготовки 09.03.01 Информатика и вычислительная техника, профиля «Программное обеспечение средств вычислительной техники и автоматизированных систем»

Составитель _____ О.В. Подсобляева

Методические указания рассмотрены и одобрены на заседании кафедры программного обеспечения, протокол № 9 от «07» июня 2017 г.

Заведующий кафедрой программного обеспечения _____ Е.Е.Сурина

1 Методические указания по проведению лекционных занятий

Лекционные занятия в высшем учебном заведении являются основной формой организации учебного процесса и должны быть нацелены на выполнение ряда задач:

- ознакомить студентов со структурой дисциплины;
- изложить основной материал программы курса дисциплины;
- ознакомить с новейшими подходами и проблематикой в данной области;
- сформировать у студентов потребность к самостоятельной работе с учебной, нормативной и научной литературой.

Лекционное занятие представляет собой систематическое, последовательное, монологическое изложение преподавателем-лектором учебного материала, как правило, теоретического характера.

Цель лекции – организация целенаправленной познавательной деятельности студентов по овладению программным материалом учебной дисциплины.

Чтение курса лекций позволяет дать связанное, последовательное изложение материала в соответствии с новейшими данными науки, сообщить слушателям основное содержание предмета в целостном, систематизированном виде.

В ряде случаев лекция выполняет функцию основного источника информации, когда новые научные данные по той или иной теме не нашли отражения в учебниках.

Организационно-методической базой проведения лекционных занятий является рабочий учебный план направления подготовки. При подготовке лекционного материала преподаватель обязан руководствоваться учебными программами по дисциплинам кафедры, тематика и содержание лекционных занятий которых представлена в рабочих программах, учебно-методических комплексах.

При чтении лекций преподаватель имеет право самостоятельно выбирать формы и методы изложения материала, использовать различные технические средства обучения.

Рекомендации по работе студентов с конспектом лекций.

Изучение дисциплины студенту следует начинать с проработки рабочей программы, особое внимание, уделяя целям и задачам, структуре и содержанию курса.

При конспектировании лекций студентам необходимо излагать услышанный материал кратко, своими словами, обращая внимание, на логику изложения материла, аргументацию и приводимые примеры. Необходимо выделять важные места в своих записях. Если непонятны какие-либо моменты, необходимо записывать свои вопросы, постараться найти ответ на них самостоятельно. Если самостоятельно не удалось разобраться в материале, впоследствии необходимо либо на следующей лекции, либо на лабораторном занятии или консультации обратиться к ведущему преподавателю за разъяснениями.

Успешное освоение курса предполагает активное, творческое участие студента путем планомерной, повседневной работы. Лекционный материал следует просматривать в тот же день. Рекомендуемую дополнительную литературу следует прорабатывать после изучения данной темы по учебнику и материалам лекции.

Каждая тема имеет свои специфические термины и определения. Усвоение материала необходимо начинать с усвоения этих понятий. Если какое-либо понятие вызывает затруднения, необходимо посмотреть его суть и содержание в словаре (Интернете), выписать его значение в тетрадь для подготовки к занятиям.

При подготовке материала необходимо обращать внимание на точность определений, последовательность изучения материала, аргументацию, собственные примеры, анализ конкретных ситуаций. Каждую неделю рекомендуется отводить время для повторения пройденного материала, проверяя свои знания, умения и навыки по контрольным вопросам и тестам.

2 Методические указания по практическим занятиям

Изучение дисциплины «Алгоритмы и структуры данных» предполагает посещение обучающимися не только лекций, но и лабораторных работ. Лабораторные работы со студентами предназначены для проверки усвоения ими теоретического материала дисциплины.

Основные цели лабораторных работ:

- закрепить основные положения дисциплины;
- проверить уровень усвоения и понимания студентами вопросов, рассмотренных на лекциях и самостоятельно изученных по учебной литературе;
- научить пользоваться нормативной и справочной литературой для получения необходимой информации о конкретных технологиях;
- оказать помощь в приобретении навыков расчета точностных характеристик;
- восполнить пробелы в пройденной теоретической части курса и оказать помощь в его усвоении.

Для контроля знаний, полученных в процессе освоения дисциплины на лабораторных занятиях обучающиеся выполняют задания реконструктивного уровня и комплексное практическое задание.

Целью выполнения задания реконструктивного уровня и комплексного задания студентами является систематизация, закрепление и расширение теоретических знаний, полученных в ходе изучения дисциплины.

Ниже приводятся общие методические указания, которые относятся к занятиям по всем темам:

- в начале каждого лабораторного занятия необходимо сформулировать цель, поставить задачи;
- далее необходимо проверить знания студентами лекционного материала по теме занятий;
- в процессе занятия необходимо добиваться индивидуальной самостоятельной работы студентов;
- знания студентов периодически контролируются путем проведения текущей аттестации (рубежного контроля), сведения о результатах которой доводятся до студентов и подаются в деканат;
- время, выделенное на отдельные этапы занятий, указанное в рабочей программе, является ориентировочным; преподаватель может перераспределить его, но должна быть обеспечена проработка в полном объеме приведенного в рабочей программе материала;
- на первом занятии преподаватель должен ознакомить студентов с правилами поведения в лаборатории и провести инструктаж по охране труда и по пожарной безопасности на рабочем месте;
- преподаватель должен ознакомить студентов со всем объемом лабораторных работ и требованиями, изложенными выше;
- преподаватель уделяет внимание оценке активности работы студентов на занятиях, определению уровня их знаний на каждом занятии.

На лабораторных работах решаются задачи из всех разделов изучаемой дисциплины.

Практическая работа №1
АЛГОРИТМЫ СОРТИРОВКИ МАССИВОВ

Цель работы

В данной работе рассматриваются наиболее известные алгоритмы внутренней сортировки данных в применении к числовым массивам. Выполнение работы преследует следующие цели:

изучение основных идей, алгоритмов, приемов программирования, связанных с классической задачей сортировки массивов;

получение навыков сравнения алгоритмов по эффективности.

Основные сведения

Задача внутренней сортировки

Сортировкой называется перестановка записей таблицы (или в частном случае – элементов массива) в соответствии с некоторым заданным отношением порядка (по алфавиту, по возрастанию числового значения поля и т.п.). Сортировка, во-первых, позволяет просматривать, обрабатывать, выдавать записи таблицы в нужной последовательности, и, во-вторых, дает возможность выполнять быстрый (бинарный) поиск данных в таблице.

Различают алгоритмы внутренней сортировки (когда вся сортируемая таблица может быть полностью размещена в оперативной памяти) и алгоритмы внешней сортировки (когда таблица велика и должна считываться в память из файла по частям).

Ввиду того, что сортировка – одна из наиболее часто используемых процедур обработки данных, к эффективности алгоритмов сортировки предъявляются очень высокие требования.

Производительность алгоритмов может оцениваться как с помощью теоретически полученных оценок, так и эмпирически, путем экспериментального сравнения алгоритмов.

Теоретические оценки производительности чаще всего выражаются в виде оценок скорости роста времени сортировки в зависимости от числа записей сортируемой таблицы «с точностью до **O**-большого». Говорят, например: «Данный алгоритм имеет оценку $\mathbf{O(N^2)}$». Это означает, что при увеличении числа записей **N**, к примеру, в 10 раз, время работы алгоритма возрастает примерно в $10^2 = 100$ раз. При этом следует различать оценки максимального времени (т.е. для случая самых неудачных для этого алгоритма данных) и среднего времени (т.е. математического ожидания времени сортировки для случайной таблицы).

Оценки «с точностью до **O**-большого» имеют смысл для сравнения поведения алгоритмов при больших значениях **N**. Практически для больших **N** считаются приемлемыми алгоритмы с оценкой $\mathbf{O(N \cdot log(N))}$ или близкой к этому значению, в то время как алгоритмы сортировки с оценкой $\mathbf{O(N^2)}$ считаются неприемлемо медленными. В то же время при небольших размерах таблицы (скажем, несколько десятков записей) простые алгоритмы с оценкой $\mathbf{O(N^2)}$ могут работать даже быстрее, чем усложненные алгоритмы с лучшей оценкой.

Экспериментальная оценка алгоритмов дает более конкретный результат, однако время работы оцениваемой программы зависит, кроме качества алгоритма, еще от многих факторов, в частности, от производительности ЭВМ и от конкретных сортируемых данных, которые при одном и том же размере могут быть более или менее удачными для испытываемого алгоритма. Чтобы уменьшить зависимость экспериментальной оценки от производительности ЭВМ, желательно, кроме времени работы, фиксировать также количество выполненных операций, характерных для данного алгоритма. Для сортировки такими операциями являются сравнение ключей элементов таблицы и перестановка (присваивание) записей. Кроме того, полезно сравнить на одних и тех же исходных данных исследуемый алгоритм с каким-либо из хорошо известных алгоритмов. Для устранения влияния конкретных данных на оценку алгоритма следует многократно

повторять испытания со случайными исходными данными, однако такие эксперименты требуют много машинного времени, а также грамотной оценки результатов с помощью методов математической статистики.

При сортировке таблиц различают ключ записи (т.е. ту часть записи, которая используется для сравнения записей) и саму запись, которая, кроме ключа, может содержать дополнительные данные. Алгоритмы сортировки состоят, главным образом, из сравнения ключей и перестановки записей и отличаются друг от друга тем, что и в каком порядке сравнивается и переставляется. В задаче сортировки массивов ключами являются сами элементы массива, и они же переставляются. Несущественность этого различия позволяет изучать алгоритмы сортировки для массивов, при необходимости без труда получая модификации этих алгоритмов для таблиц общего вида.

Ниже приводится сжатый обзор наиболее популярных алгоритмов внутренней сортировки.

Простые алгоритмы сортировки массивов

К простым алгоритмам принято относить следующие три очевидных и несложных в реализации алгоритма, не отличающихся, к сожалению, высокой эффективностью.

Алгоритм пузырька (BubbleSort)

Идея алгоритма заключается в следующем. Сравним элементы массива с индексами 1 и 2. Если первый больше второго, то поменяем эти элементы местами. Затем таким же образом сравним (и, если нужно, переставим) элементы с индексами 2 и 3, потом 3 и 4 и т.д. После сравнения элементов (**N-1**) и **N** первый проход алгоритма завершается. Можно гарантировать, что после этого прохода максимальный элемент массива находится на последнем месте (т.е. имеет индекс **N**). На втором проходе сравниваем пары 1 и 2, 2 и 3, ... (**N-2**) и (**N-1**). Далее аналогично. После **N-1** прохода все элементы займут свои законные места.

Можно попытаться сократить число проходов, отмечая с помощью специального флажка, были ли сделаны какие-либо перестановки на очередном проходе. Если ни одной перестановки за весь проход не было, то массив, очевидно, уже отсортирован и работу алгоритма можно завершить досрочно. Однако такое усовершенствование редко дает заметный выигрыш. Нетрудно показать, что для массива, заполненного случайным образом, с вероятностью 75% все равно будут выполнены все проходы алгоритма.

Ненамного лучшие результаты дает и другая модификация алгоритма пузырька, называемая «шейкер-сортировкой». Она отличается тем, что на нечетных проходах пузырек проходит слева направо (индекс в цикле возрастает), а на четных – справо налево (индекс убывает). Таким образом, после первых двух проходов на свои места станут максимальный и минимальный элементы массива.

Алгоритм простого выбора (SelectionSort)

Идея этого алгоритма еще проще. Найдем минимальный элемент массива и поменяем его местами с первым элементом. Затем повторим ту же процедуру, начиная со второго элемента массива, затем начиная с третьего и т.д. После **N-1** проходов все элементы станут на места.

Алгоритм простых вставок (InsertionSort)

Идея заключается в следующем. Пусть к некоторому моменту работы алгоритма первые **k** элементов массива уже отсортированы, т.е. расположены по возрастанию. На очередном проходе постараемся добиться, чтобы стали отсортированными (**k+1**) элементов. Для этого запомним значение элемента (**k+1**) в рабочей переменной **R** и будем сравнивать **R** со значениями элементов **k**, (**k-1**), (**k-2**) и т.д. Если значение сравниваемого элемента больше **R**, то этот элемент перемещается на одну позицию правее. Сравнения продолжаются, пока не будет найдено место, куда должен быть помещен элемент **R** (это случится либо когда очередной сравниваемый элемент меньше или равен **R**, либо когда мы дойдем до начала массива).

Таким образом, на очередном проходе отсортированная часть массива удлиняется на 1 элемент. Начав со значения **k=1**, можно за **N-1** проход отсортировать весь массив.

Алгоритм простых вставок можно улучшить, если выбирать место для вставки **k+1**-го элемента не последовательным просмотром элементов от **k** до **1**, а бинарным поиском (т.е. сравнить **R** с элементом **j := (k+1) div 2**, затем продолжить поиск на одном из интервалов **[1..j-1]** или **[j+1..k]** и т.д.). Этот подход, называемый алгоритмом бинарных вставок, позволяет существенно сократить число сравнений, но, к сожалению, не влияет на число перестановок.

Сравнение простых алгоритмов

Все три вышеописанных алгоритма и все их модификации имеют как максимальную, так и среднюю оценку $\mathbf{O(N^2)}$, а потому используются только в случае небольших массивов или отсутствия жестких требований к времени сортировки.

Эксперименты показывают, что алгоритм пузырька является обычно наиболее медленным из трех, а алгоритмы выбора и вставок дают примерно одинаковое время сортировки.

В некоторых случаях можно ожидать, что исходный массив окажется «почти отсортированным», т.е. лишь небольшое число элементов будет нарушать порядок. Такое бывает, например, если массив ранее уже был отсортирован, но потом данные в нем подверглись небольшим модификациям. В этом случае вне конкуренции оказываются алгоритмы вставок, которые показывают очень высокую скорость на почти отсортированных массивах. Эта особенность алгоритмов вставок используется в описанном ниже алгоритме Шелла (п. 0).

Усовершенствованные алгоритмы сортировки массивов

Усложнение алгоритмов позволяет значительно повысить эффективность сортировки больших массивов. Перечислим наиболее известные алгоритмы этого класса.

Алгоритм Шелла

Разобьем элементы сортируемого массива на **h** цепочек, каждая из которых состоит из элементов, отстоящих друг от друга на расстояние **h** (здесь **h** – произвольное натуральное число). Первая цепочка будет содержать элементы с индексами **1**, **h+1**, **2h+1**, **3h+1** и т.д., вторая – **2**, **h+2**, **2h+2** и т.д., последняя цепочка – **h**, **2h**, **3h** и т.д. Отсортируем каждую цепочку как отдельный массив, используя для этого метод простых вставок. Затем выполним все вышеописанное для ряда убывающих значений **h**, причем последний раз – для **h=1**.

Очевидно, массив после этого окажется отсортированным. Неочевидно, что все проходы при **h>1** не были пустой тратой времени. Тем не менее, оказывается, что дальние переносы элементов при больших **h** настолько приближают массив к отсортированному состоянию, что на последний проход остается очень мало работы. Эксперименты показывают, что для больших значений **N** оценка среднего времени работы алгоритма примерно $\mathbf{O(N^{1.26})}$. Это значительно лучше, чем $\mathbf{O(N^2)}$ для простых алгоритмов.

Большое значение для эффективности алгоритма Шелла имеет удачный выбор убывающей последовательности значений **h**. Желательно, чтобы при соседних значениях **k** значения $\mathbf{h_k}$ не были кратны друг другу. В литературе обычно рекомендуется использовать одну из двух последовательностей: $\mathbf{h_{k+1} = 3h_k+1}$ или $\mathbf{h_{k+1} = 2h_k+1}$. В обоих случаях в качестве начального $\mathbf{h_k}$ выбирается такое значение из последовательности, при котором все сортируемые цепочки имеют длину не меньше 2. Чтобы воспользоваться, например, первой из этих формул, надо сначала положить $\mathbf{h_1:=1}$, а затем в цикле увеличивать значение **h** по формуле $\mathbf{h_{k+1}:=3*h_k+1}$, пока для очередного $\mathbf{h_k}$ не будет выполнено неравенство $\mathbf{h_k >= (N–1)\ div\ 3}$. Это значение $\mathbf{h_k}$ следует использовать на первом проходе алгоритма, а затем можно получать следующие значения по обратной формуле: $\mathbf{h_{k-1} := (h_k-1)\ div\ 3}$, вплоть до $\mathbf{h_1}$=1.

Более сложными формулами определяется последовательность Седжвика:

$$h_k = \begin{cases} 9 \cdot 2^k - 9 \cdot 2^{k/2} + 1, & \textit{если k четно;} \\ 8 \cdot 2^k - 6 \cdot 2^{k+1/2} + 1, & \textit{если k нечетно.} \end{cases}$$

Доказано, что при выборе $\mathbf{h_k}$ по Седжвику среднее время работы алгоритма есть $\mathbf{O(N^{7/6})}$, а максимальное – $\mathbf{O(N^{4/3})}$.

На практике удобно раз и навсегда вычислить достаточное количество членов последовательности Седжвика (вот они: 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, 8929, 16001, 36289, 64769, 146305, 260609, 587521, 1045505, ...), затем по заданному **N** выбрать такое **k**, при котором $\mathbf{h_k >= (N–1)\ div\ 3}$, а далее в цикле выбирать значения последовательности $\mathbf{h_k}$ по убыванию **k**.

Алгоритм быстрой сортировки (QuickSort)

В основе этого алгоритма – операция ***разделения массива***. Пусть **x** – некоторый произвольно выбранный элемент сортируемого массива (разделяющий элемент). Операция разделения имеет целью переставить элементы массива таким образом, чтобы сначала шли элементы, меньшие или равные **x** (не обязательно в порядке возрастания), а затем элементы, большие или равные **x**. При этом совершенно неважно, где именно окажется после разделения сам элемент **x**.

Чтобы выполнить разделение, начнем просматривать элементы массива слева направо и остановимся на первом из элементов, большем или равном **x**. Пусть индекс этого элемента – **i**. Аналогичным образом, начав с правого конца массива, остановимся на самом правом элементе, меньшем или равном **x** (индекс элемента – **j**). Поменяем местами элементы **i** и **j**, а затем продолжим идти вправо от **i** и влево от **j**, меняя местами пары элементов, стоящих не на месте. Разделение окончится, когда будет **i>j**.

Алгоритм быстрой сортировки можно теперь описать таким образом. Возьмем произвольный элемент массива **x** и выполним для него операцию разделения. Если левый и/или правый отрезки массива содержат более одного элемента, то выполним для каждого из этих отрезков ту же операцию, что и для всего массива (т.е. выберем произвольный элемент, выполним разделение и т.д.).

Организовать вышеописанное многократное выполнение операции разделения проще всего с помощью рекурсивного определения процедуры сортировки. Нерекурсивное описание алгоритма оказывается сложнее, т.к. при необходимости сортировать два отрезка массива, образовавшиеся после разделения, приходится явно сохранять в стеке один из отрезков (точнее, два граничных значения его индексов), пока сортируется другой. В то же время нерекурсивный вариант обычно оказывается более эффективным и, что немаловажно, он позволяет уменьшить используемую глубину стека. Для этого следует всегда из двух отрезков массива, подлежащих сортировке, заносить в стек более длинный. При этом можно гарантировать, что максимальное количество отрезков, одновременно находящихся в стеке, не может превысить $\mathbf{\log_2(N)}$.

На эффективность алгоритма быстрой сортировки влияет также выбор разделяющего элемента **x**. Хотя теоретически **x** может выбираться произвольным образом, легко видеть, что выбор в качестве **x** первого или последнего элемента приводит к очень плохим результатам, если исходный массив оказывается уже сортированным или близким к сортированному. В связи с этим принято выбирать в качестве **x** либо элемент из середины массива, либо элемент со случайно выбранным индексом. Иногда используют чуть более сложное правило: взять первый элемент массива, последний элемент, элемент из середины массива и выбрать в качестве **x** средний по величине из этих трех.

Алгоритм **QuickSort** имеет оценку среднего времени $\mathbf{O(N \cdot \log(N))}$ и на практике оказывается быстрее всех прочих, однако в худшем случае он может потребовать времени порядка $\mathbf{O(N^2)}$.

В некоторых случаях может оказаться разумным скомбинировать **QuickSort** с алгоритмом пузырька. Дело в том, что достаточно громоздкая процедура разделения выполняет очень мало полезной работы в конце, когда разделяемые отрезки массива становятся короткими. Можно не разделять отрезки, длина которых не больше некоторого

L. После этого можно «досортировать» массив, выполнив **L-1** проход «пузырька». При этом нет необходимости организовывать отдельные проходы «пузырька» для множества коротких массивов, проще пропустить «пузырек» по всему сортируемому массиву. Практика показывает, что разумные значения **L** не превышают 3 – 4.

Алгоритм пирамидальной сортировки (HeapSort)

В основе этого алгоритма лежит понятие ***пирамиды***.

Массив $\mathbf{A_1}$, $\mathbf{A_2}$, ... $\mathbf{A_N}$ называется пирамидой, если:

$\mathbf{A_k \geq A_{2k}}$ для всех **k** таких, что $\mathbf{2k \leq N}$;

$\mathbf{A_k \geq A_{2k+1}}$ для всех **k** таких, что $\mathbf{2k+1 \leq N}$.

Пирамиду можно рассматривать как представленное в виде массива двоичное дерево, у которого значение, связанное с вершиной-родителем, не меньше, чем значения, связанные с сыновьями. При этом $\mathbf{A_1}$ – корень дерева, $\mathbf{A_2}$ и $\mathbf{A_3}$ – его сыновья, $\mathbf{A_4}$, … $\mathbf{A_7}$ – внуки и т.д.

Алгоритм сортировки состоит из двух фаз: построения пирамиды и преобразования пирамиды в отсортированный массив. В основе каждой из фаз лежит операция ***просеивания*** элементов через пирамиду.

Суть операции просеивания заключается в следующем. Предположим, имеется «почти правильная» пирамида, лишь для одного из элементов которой $\mathbf{A_k}$, возможно, нарушены приведенные выше неравенства. Чтобы восстановить правильность пирамиды, следует сначала сравнить между собой сыновей $\mathbf{A_k}$, т.е. элементы $\mathbf{A_{2k}}$ и $\mathbf{A_{2k+1}}$. Обозначим номер большего из этих элементов буквой **r**. Теперь следует сравнить $\mathbf{A_r}$ с отцом (элементом $\mathbf{A_k}$) и, если неравенство $\mathbf{A_k \geq A_r}$ нарушено, то поменять местами $\mathbf{A_k}$ и $\mathbf{A_r}$. Теперь оба неравенства для $\mathbf{A_k}$ будут выполняться, однако могут оказаться нарушенными аналогичные неравенства для $\mathbf{A_r}$. Поэтому следует положить **k:=r** и повторять циклически те же действия, пока не окажется, что на очередной итерации элемент $\mathbf{A_k}$ либо уже не имеет сыновей, либо для него выполнены требуемые неравенства. На этом просеивание элемента $\mathbf{A_k}$ заканчивается.

Если представить пирамиду как двоичное дерево, то просеивание будет выглядеть как «падение» элемента $\mathbf{A_k}$ вдоль одной из ветвей дерева, пока он не займет подобающее ему место.

Построение пирамиды начинается с просеивания элемента, индекс которого **k := N div 2** (это самый правый из элементов, имеющих сыновей в дереве). После просеивания этого элемента отрезок, начиная с индекса **k** и до конца массива, будет соответствовать требованиям к пирамиде (потому что для элементов с номерами, большими **k**, никаких неравенств проверять не надо, у них нет сыновей!). Далее уменьшаем значение **k** на единицу и опять выполняем просеивание. Построение пирамиды будет завершено, когда будет просеян элемент $\mathbf{A_1}$. Отметим, что первый элемент построенного массива-пирамиды – это его наибольший элемент.

Фаза преобразования пирамиды в отсортированный массив начинается с обмена значениями первого (наибольшего) элемента пирамиды с последним элементом. При этом наибольший элемент становится на свое окончательное место и не считается далее частью пирамиды (т.е. **N := N-1**). Сама пирамида может оказаться испорченной за счет замены корня. Чтобы восстановить пирамиду, следует таким же образом, как при построении пирамиды, еще раз просеять только новый корневой элемент $\mathbf{A_1}$. Когда пирамида (уменьшившаяся на один элемент) восстановлена, ее корневой элемент – наибольший среди элементов пирамиды. Он опять обменивается с последним ее элементом, пирамида уменьшается еще на один элемент и т.д., пока пирамида не выродится в один элемент, за которым будут следовать отсортированные элементы массива.

Для алгоритма **HeapSort** имеются гарантированные оценки как максимального, так и среднего времени работы порядка $\mathbf{O(N \cdot log(N))}$, однако эксперименты показывают, что этот алгоритм обычно работает несколько медленнее, чем **QuickSort**, а в некоторых случаях уступает и алгоритму Шелла.

Алгоритмы слияния

Алгоритмы этой группы раньше широко использовались для сортировки последовательных файлов на магнитной ленте, однако и в задачах внутренней сортировки алгоритмы слияния показывают хорошую производительность. Их существенным недостатком является потребность во вспомогательном массиве того же размера, что и сортируемый массив.

Разнообразные алгоритмы слияния основываются на процедуре слияния двух уже отсортированных массивов в один отсортированный массив. Такая процедура реализуется легко и эффективно, за один проход по массивам. Рассмотрим два наиболее известных алгоритма сортировки путем многократных слияний.

Алгоритм простого слияния основан на следующем рассуждении. Пусть дан массив из **N** элементов. Каждый элемент исходного массива можно формально рассматривать как подмассив длины 1, причем, конечно, отсортированный. Сгруппируем попарно подмассивы из элементов с индексами 1 и 2, 3 и 4 и т.д. Будем сливать эти пары, записывая результат слияния во вспомогательный массив. (Возможен также другой вариант слияния: элементы 1 и **(N div 2)+1**, 2 и **(N div 2)+2** и т.п. Иногда это удобнее с точки зрения программирования.) В результате во вспомогательном массиве образуется **(N div 2)** сортированных подмассивов длины 2, а при нечетном **N** – еще один подмассив длины 1. Повторим операцию слияния, считая теперь источником вспомогательный массив, а приемником – исходный, и сливая подмассивы длины 2 в подмассивы длины 4 (последний подмассив опять может получиться неполным). Повторяем слияния до тех пор, пока не получится один массив длины **N**. Если для этого потребуется нечетное число проходов, то результат окажется во вспомогательном массиве и его нужно будет еще перенести в исходный массив.

Алгоритм естественного слияния исходит из того, что в исходном массиве почти наверняка уже есть отрезки, где элементы идут по возрастанию. Назовем такие отрезки ***сериями***, допуская в том числе «серии» длиной 1. Проход алгоритма состоит из двух фаз. На фазе распределения из массива выделяются серии, причем нечетные по счету серии записываются во вспомогательный массив от его начала, а четные – от конца по убыванию индекса. На фазе слияния сливаются пары серий, одна от начала вспомогательного массива, другая от конца, и результат записывается в исходный массив от его начала. Проходы повторяются, пока не останется одна серия. При реализации этого алгоритма следует учитывать, что число сливаемых серий на фазе слияния может оказаться меньше, чем число серий, записанных на фазе распределения. Такое возможно, если две серии (например, первая и третья) сами сольются в одну, т.е. если последний элемент первой серии не больше, чем первый элемент третьей серии. Из-за этого при слиянии количество нечетных и четных серий может оказаться различным. Лишние серии, оставшиеся без пары, просто переписываются из вспомогательного массива в исходный.

Оба описанных выше алгоритма слияния имеют оценку **O(N·log(N))** как для среднего, так и для максимального времени выполнения.

Экспериментальная оценка производительности алгоритма

Оценка производительности алгоритма имеет первостепенное значение при определении пригодности этого алгоритма для решения конкретных типов задач. Важнейшей частью данной работы является сбор статистики о времени работы алгоритмов при сортировке различных массивов данных.

Трудоемкость алгоритма может измеряться либо в единицах времени, либо в единицах числа операций, наиболее характерных для алгоритмов данного типа. Для алгоритмов сортировки обычно измеряют число операций сравнения элементов массива и отдельно – число операций присваивания элементов. Остальными операциями – инициализацией и изменением счетчиков циклов, проверками окончания и т.п. – можно пренебречь, поскольку их число либо значительно меньше, чем числа сравнений и присваиваний, либо, в крайнем случае, прямо пропорционально им.

Поскольку для большей части алгоритмов трудоемкость может сильно зависеть от более удачных или менее удачных для данного алгоритма данных, для экспериментальной оценки трудоемкости чаще всего используют случайные данные. Результаты одного прогона алгоритма на случайном массиве данных не несут надежной информации, однако при увеличении числа прогонов с усреднением результатов обычно удается получить достаточно объективную оценку алгоритма. Эти вопросы подробно исследуются в теории математической статистики, здесь же можно лишь отметить, что случайная ошибка определения трудоемкости убывает обратно пропорционально квадратному корню из числа экспериментов.

Оценка трудоемкости непосредственно в единицах времени тоже является достаточно полезной, поскольку показывает, чего можно ждать от использования конкретной программы сортировки.

ОС Windows предоставляет несколько различных API-функций для измерения системного времени. Однако наиболее простая из этих функций **GetTickCount** дает достаточно низкую точность (10 мс для версий от Windows XP и выше). При этом для получения достаточно малой относительной ошибки измерений требуется, чтобы измеряемые интервалы были, по крайней мере, не меньше секунды. Таким образом, доверять можно только таким оценкам времени сортировки, которые получены для больших массивов данных, сортировка которых требует нескольких секунд. Учитывая высокую эффективность усовершенствованных алгоритмов сортировки и большую производительность современных процессоров, размеры массивов должны при этом измеряться, по крайней мере, миллионами элементов, что не всегда удобно. Можно использовать функцию **QueryPerformanceCounter**, однако она, обеспечивая высокую точность измерения времени, не позволяет отделить время данного процесса от затрат времени системой и другими приложениями.

Наиболее подходящим средством измерения времени работы алгоритма представляются API-функции **GetProcessTimes** и **GetThreadTimes**. Эти функции позволяют получить чистое процессорное время, затраченное, соответственно, всем процессом или только одной из его нитей. При этом имеется возможность отделить время работы в режиме задачи (т.е. время выполнения прикладного кода) от времени работы в режиме системы (выполнения API-функций и т.п.). Номинальная точность измерения времени – 100 нс.

Выполнение задания

В данной лабораторной работе требуется запрограммировать и протестировать алгоритмы внутренней сортировки, указанные в варианте задания.

В каждом варианте задания требуется запрограммировать один или два (в зависимости от сложности) алгоритма сортировки и проверить их работу на ряде тестовых примеров.

В качестве тестовых массивов следует использовать:

массив из 1000 первых натуральных чисел в порядке возрастания (т.е. пример уже отсортированного массива);

массив из 1000 первых натуральных чисел в порядке убывания (т.е. пример массива, отсортированного «наоборот»);

не менее 4 сгенерированных массивов псевдослучайных чисел разного размера, от 100 до 100000 элементов (максимальный размер массива может быть изменен в зависимости от производительности процессора).

Таблица результатов должна для каждого алгоритма и для каждого тестового массива включать время работы алгоритма, число выполненных сравнений и присваиваний элементов массива.

По крайней мере, для одного не очень большого примера следует для визуальной проверки правильности работы алгоритма выдать на экран весь сортируемый массив до и после сортировки.

Отчет о работе должен включать:

постановку задачи, вариант задания, список бригады;

исходные тексты разработанных процедур сортировки (не обязательно полный текст всей программы);

итоговую таблицу результатов тестирования;

выводы, которые можно сделать по этим результатам.

Отчет и программа представляются также в электронном виде.

Пример выполнения задания

Дан вариант задания: запрограммировать алгоритм BubbleSort и рекурсивный вариант алгоритма QuickSort, получить статистику работы алгоритмов на ряде массивов.

Программа была реализована в системе программирования Borland Delphi как консольное приложение Windows. Исходные тексты программы прилагаются в проекте **Laba3**.

Ниже приводится таблица результатов работы программы для ряда тестовых массивов. Использовался процессор Intel E2180 @2.00 ГГц.

		BubbleSort			**QuickSort**		
Данные в массиве	N	Время (мс)	Сравнения	Присваивания	Время (мс)	Сравнения	Присваивания
Возрастающие	1000	15.625	499500	0	0.000	12724	4812
Убывающие	1000	0.000	499500	1498500	0.000	12724	6309
Случайные	100	0.000	4950	8193	0.000	1115	874
Случайные	1000	15.625	499500	757047	0.000	17149	10598
Случайные	10000	703.125	49995000	75153522	0.000	210353	132601
Случайные	100000	74578.125	4999950000	7509426354	31.250	2544779	1605720

Как видно из таблицы, алгоритм BubbleSort показал приемлемые результаты только на массиве из 100 элементов, а для больших массивов продемонстрировал квадратичный рост числа выполненных операций. При этом число выполненных сравнений не зависит от конкретных данных, оставаясь одним и тем же даже для заранее отсортированного массива.

Алгоритм QuickSort показал значительно лучшие результаты уже для массива из 1000 элементов, а для N=100000 он проработал быстрее, чем BubbleSort, более чем в 2000 раз.

Таким образом, эксперименты подтвердили непригодность алгоритма BubbleSort для сортировки больших массивов и высокую эффективность, достигаемую в этом случае алгоритмом QuickSort.

Варианты заданий

Запрограммировать и протестировать следующие алгоритмы, описанные выше

1) Нерекурсивный вариант алгоритма QuickSort со случайным выбором разделяющего элемента и с выбором среднего по величине из трех элементов;
2) Алгоритм простого выбора и алгоритм HeapSort;
3) Комбинация QuickSort и пузырька для 2-3 различных значений L;
4) Алгоритм бинарных вставок и алгоритм Шелла;
5) Алгоритм простых вставок и алгоритм естественного слияния;
6) Алгоритм шейкер-сортировки и алгоритм простого слияния.

Практическая работа №2
«Кольцевые односвязные списки»

ПОСТАНОВКА ЗАДАЧИ

Сформировать линейный односвязный список (ЛОС) с заданным указателем sag, работающий с типом данных Integer. Составить программу, которая должна из заданного списка удалить первый и последний элементы.

Составить программу, которая:

- обеспечивает ввод данных типа Integer с клавиатуры;
- создает линейный односвязный список из введенных данных с клавиатуры;
- обеспечивает диалог посредством вывода информационных сообщений и вариантов выполнения дальнейших действий;
- удаляет первый и последний элементы.
- в данной программе будут реализованы следующие возможности работы с ЛОС:

0 - Выход из программы
1 - Создание ЛОС
2 - Добавление элемента в начало списка
3 - Добавление элемента в середину списка, перед указанным значением
4 - Добавление элемента в середину списка, после указанного значения
5 - Добавление элемента в конец списка
6 - Удаление элемента в начале списка
7 - Удаление элемента ЛОС стоящего перед указанным значением списка
8 - Удаление элемента ЛОС стоящего после указанного значения списка
9 - Удаление определенного элемента в списке
10 - Удаление элемента в конце списка
11 - Удаление первого и последнего элементов ЛОС
12 - Очистка ЛОС
13 - Поиск элемента по его значению
14 - Сортировка элементов ЛОС
15 - Подсчет количества идентичных по содержанию элементов с указанным

ОПИСАНИЕ АЛГОРИТМА РЕШЕНИЯ ЗАДАЧИ

Ввод данных осуществляется в диалоговом режиме.

Пользователь информируется о вариантах работы в данной программе, об особенностях ввода значений в программе.

Далее осуществляется ввод самого списка. Создается линейный односвязный список, с указанием на конец списка (NIL) и по мере ввода данных, ЛОС наполняется, при этом идет сортировка значений элементов по возрастанию.

После ввода необходимого количества элементов и ввода нулевого значения, созданный и отсортированный ЛОС выводиться на экран.

Далее, следуя указаниям программы, пользователь нажимает Enter, для продолжения работы программы, на экран выводиться перечень возможных вариантов работы в данной программе.

После выбора нужного номера операции, в нашем случае (11 - Удалить первый и последний элементы ЛОС) и нажатия на Enter. Происходит удаление первого и последнего элементов ЛОС, с выводом на экран итогового вида ЛОС.

ОПИСАНИЕ ИСПОЛЬЗУЕМЫХ СТРУКТУР ДАННЫХ

Для хранения данных в соответствии с постановкой задачи необходимо в программе создать Линейный Односвязный Список (ЛОС).

Описание типа используемых данных

```
Type
chisla = setof '0'..'9'; {множество}
TE= Integer; {описание целочисленного типа}
```

WE= String; {описание строкового типа}
PE= ^EL; {описание типа указателя}
EL= Record{описание типа - запись}
inf: TE; {информационная часть элемента, тип Integer}
inf2: WE; {информационная часть элемента, тип String}
next: PE{адресная часть элемента}
End;
Var
Sag, {указатель начала списка}
q, qq: PE; {переменные указателей}
oper, st, st2: TE; {переменные целочисленного типа}
w, stroka: WE; {переменные строкового типа}

ОПИСАНИЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

Начальный вид окна программы.

Для начала ввода данных в ЛОС, надо определиться с каким типом данных Вы хотели бы работать. После того, как Вы решили с каким типом данных Вы будете дальше работать, Вам нужно ввести номер варианта дальнейшей работы (1 или 2).
Для выполнения условий данной лабораторной, выбираем тип Integer (тип целочисленный) предел его от -32768 до 32767.

Далее, осуществляется ввод самого списка. Создается линейный односвязный список, с указанием на конец списка (NIL) и по мере ввода данных, ЛОС наполняется, при этом идет сортировка значений элементов по возрастанию.

После ввода необходимого количества элементов и ввода нулевого значения, созданный и отсортированный ЛОС выводиться на экран. (Рис.2)

Далее, следуя указаниям программы, пользователь нажимает Enter для продолжения работы программы, и на экран выводиться перечень возможных вариантов работы в данной программе.(Рис.3)

После выбора нужного номера операции, для выполнения условий нашей задачи, выбираем (11 - Удалить первый и последний элементы ЛОС) и нажимаем на Enter. Происходит удаление первого и последнего элементов ЛОС, с выводом на экран итогового вида ЛОС.(Рис.4)

Видно, что с поставленной задачей наша программа справилась. Были удалены первый и последний элементы ЛОС, а потом был выведен итоговый вид ЛОС.

ЛИСТИНГ ПРОГРАММЫ

```
Type
chisla = setof '0'..'9'; //множество
TE= Integer; //описание целочисленного типа
WE= String; //описание строкового типа
PE= ^EL; //тип указателя
EL= Record //описание типа - запись
inf: TE; //информационная часть элемента, тип Integer
inf2: WE; //информационная часть элемента, тип String
next: PE //адресная часть элемента
End;
Var
Sag, //указатель, на начало списка
q, qq: PE; //переменные указателей
oper, st, st2: TE; //переменные целочисленного типа
w, stroka: WE; //переменные строкового типа
Procedure Print(sag: PE); {выводЛОС}
Var
q: PE; //адресная переменная
```

```
Begin
q:= sag^.next; //запоминаем адрес первого элемента ЛОС
ifq= Nilthen {проверяем ЛОС на пустоту и если он пустой выводим сообщение о том, что ЛОС пустой и выводим варианты дальнейшей работы программы}
begin
WriteLn(rus('ЛОС пустой, выводить нечего!'));
WriteLn('');
WriteLn(rus('_____'));
WriteLn(rus('Что Вы хотите сделать?'));
WriteLn(rus(''));
WriteLn(rus('1 - СоздатьЛОС'));
WriteLn(rus(''));
WriteLn(rus('0 - Выйти'));
WriteLn(rus(''));
WriteLn(rus('Введите номер требуемой операции '));
WriteLn(rus('_____'));
WriteLn('');
end
Else {если ЛОС не пустой продолжаем выполнение процедуры}
Begin {проверяем, с каким типом данных происходит работа программы}
Ifst = 1 then {если пользователь выбрал вариант работы, работа с типом Integer}
//st = 1 – работа с типом данных, Integer
Begin
While q<>Nil do {проходим по всему ЛОС пока не дойдем до указателя конца списка (Nil)}
Begin
Write('[',q^.inf,'] '); {выводим на экран значение элемента ЛОС – тип Integer}
q:=q^.next; //запоминаем адрес следующего элемента
End;
WriteLn;
end
else
Begin
Whileq<>Nildo {проходим по всему ЛОС пока не дойдем до указателя конца списка (Nil)}
Begin
Write('[',q^.inf2,'] '); {выводим на экран значение элемента ЛОС – тип String}
q:=q^.next; //запоминаем адрес следующего элемента
End;
WriteLn(' ');
end;
End;
End;
Procedure Proverka (var w: WE); {проверка превышения значения типа Integer}
Var
a, i: TE;
c: char;
b: chisla;
Begin
w:= '';
ReadLn(w); //ввод числа с клавиатуры – тип String
While (w = '') or (w='-')do {проверяем, если пользователь не ввел данные или ввел только знак минуса выводим на экран сообщения о не корректные вводе}
Begin
WriteLn(Rus('Вы не ввели данные или они не корректны, попробуйте еще раз')); WriteLn(' ');
Proverka(w); //выполняем рекурсивный вход в процедуру
End;
fori:= 1 tolength(w) do {запускаем цикл проверки числа на корректность ввода, число введено, как строка. По этому мы можем поэлементно проверить каждую цифру}
```

```
begin
b:= ['0','1','2','3','4','5','6','7','8','9']; //множество состоящее из цифр
c:=w[i]; {берем каждую цифру из числа проходя от первой до последней}
if (cinb) or (c='-') and (i=1) then {сравниваем есть ли цифра из введенного числа во множестве заданных цифр и проверяем какое число было введено, отрицательное или положительное и не стоит ли знак минус в середине числа}
else
a:= 1; {если число не корректно делаем пометку для дальнейшей проверки}
end;
ifa = 1 then {если число не прошло проверку выводим
сообщение о не корректном вводе числа}
begin
WriteLn(rus('Вы ввели не корректные данные !'));
WriteLn(' ');
Proverka(w); {выполняем рекурсивный вход в
процедуру}
end;
if (length(w)<5) then {если длина числа меньше 5 знаков заканчиваем проверку, так как число не превышает максимального значения типа Integer, а корректность ввода мы уже проверили}
else {если число больше то проверяем его
дальше}
begin
if (length(w)>5) and (w[1]<> '-') then {если длина числа больше пяти
знаков, и при этом первый знак, не знак
минуса то выводим сообщение о
превышении максимального
значения типа Integer}
begin
Write(rus('Вы ввели не число или число превышающее диапазон '));
WriteLn(rus('типа Integer (-32768..32767) '));
WriteLn('');
WriteLn(rus('Введите другое число'));
Proverka(w); {выполняем рекурсивный вход в
процедуру}
end;
if (w[1]= '-') and (length(w)>4) and (w>'-32768') then {если первый знак числа, знак минуса, а число по длине меньше или равно четырем знакам или число больше чем четыре знака и в ходе сравнения строка со значением введенного числа, меньше или равна строке по значению с максимальным пределом типа Integer, то идем дальше. Иначе, выводим сообщение о превышении максимального значения типа Integer}
begin
Write(rus('Вы ввели не число или число превышающее диапазон '));
WriteLn(rus('типа Integer (-32768..32767) '));
WriteLn('');
WriteLn(rus('Введите другое число'));
Proverka(w); {выполняем рекурсивный вход в
процедуру}
end;
if (length(w)>4) and (w>'32767') then {если число по длине меньше или равно четырем знакам или число больше чем четыре знака и в ходе сравнения строка со значением введенного числа, меньше или равна строке по значению с максимальным пределом типа Integer, то идем дальше. Иначе выводим сообщение о превышении максимального значения типа Integer}
begin
Write(rus('Вы ввели не число или число превышающее диапазон '));
WriteLn(rus('типа Integer (-32768..32767) '));
WriteLn('');
WriteLn(rus('Введите другое число'));
```

```
Proverka(w); {выполняем рекурсивный вход в
процедуру}
end;
end;
End;
Procedure Gou(w: WE); forward; //Опережающееописаниепроцедуры
Procedure Create2(var sag: PE); {Процедура ввода элементов в ЛОС с сортировкой по возрастанию}
Var
q, qq, s: PE; //адресные переменные
a: TE; //переменная для ввода данных
Begin
writeln(' ');
WriteLn(rus('Введите элементы в ЛОС: '));
WriteLn(rus('Ввод завершите 0'));
ifst2<>1 then //если ЛОС еще не был создан
begin
New(sag); //создаем указатель начала списка
q:=sag^.next; //запоминаем адрес первого элемента ЛОС
end;
Ifst = 1 then {если пользователь выбрал вариант работы, с типом Integer}
//st = 1 – работа с типом данных, Integer
begin
Proverka(w); {вход в процедуру проверки корректности
ввода данных}
a:= StrToInt(w); {перевод числа из строкового типа данных в целочисленный}
end
else {если пользователь выбрал вариант работы, работа с типом String}
//st = 2 – работа с типом данных, String
ReadLn(stroka); //вводданныхтипа String
if(q= nil) or (st2=1) then {если ЛОС был пустым или ЛОС уже создавался}
Begin
New(q); {создаем адресную переменную для
первого элемента}
Ifst = 1 then {если пользователь выбрал вариант работы, работа с типом Integer}
//st = 1 – работа с типом данных, Integer
q^.inf:= a {заносим в информационную часть, значение данных}
else
q^.inf2:= stroka; {заносим в информационную часть, значение данных}
q^.next:= sag^.next; {в адресную часть второго элемента
заносим адресную часть первого
элемента}
sag^.next:= q; {в адресную часть первого элемента заносим адрес созданного указателя}
Ifst = 1 then {если пользователь выбрал вариант работы, работа с типом Integer}
//st = 1 – работа с типом данных, Integer
begin
Proverka(w); {вход в процедуру проверки корректности ввода данных}
a:= StrToInt(w); {перевод числа из строкового типа данных в целочисленный}
end
else
ReadLn(stroka); //вводданныхтипа String
End;
Ifst = 1 then {если пользователь выбрал вариант работы, работа с типом Integer}
//st = 1 – работа с типом данных, Integer
begin
Whilea<>0 do {запускаем цикл выполняющийся пока не будет введен ноль}
Begin
New(q); {создаем адресную переменную для первого элемента}
```

```
q^.inf:= a; {заносим в информационную часть, значение данных}
q^.next:= sag^.next; {в адресную часть второго элемента заносим адресную часть первого элемента}
sag^.next:= q; {в адресную часть первого элемента заносим адрес созданного указателя}
//---------сортировка
qq:= sag^.next; //запоминаем адрес первого элемента ЛОС
Whileqq<>Nildo {запускаем цикл выполняющийся пока не будет достигнут конец списка}
begin
s:= qq; //запоминаем адрес первого элемента ЛОС
qq:= qq^.next; {запоминаем адрес следующего элемента
ЛОС}
if (qq<>nil) then {если адресная переменная не равна указателю конца списка}
begin
begin
if (qq^.inf<s^.inf) then {если информационная часть следующего элемента меньше чем предыдущего, тогда:}
begin
a:= qq^.inf; {запоминаем значение информационной части следующего элемента}
qq^.inf:=s^.inf; {в информационную часть следующего элемента запоминаем значение информационной части
предыдущего элемента}
s^.inf:= a; {в информационную часть предыдущего элемента запоминаем значение следующего элемента}
end;
end;
end;
end; //---------конецсортировки
Proverka(w); {вход в процедуру проверки корректности ввода данных}
a:= StrToInt(w); {перевод числа из строкового типа данных в целочисленный}
end;
end
else
begin
Whilestroka<>'0' do {запускаем цикл выполняющийся пока не будет введен ноль}
Begin
New(q); {создаем адресную переменную для первого элемента}
q^.inf2:= stroka; {заносим в информационную часть, значение данных}
q^.next:= sag^.next; {в адресную часть второго элемента
заносим адресную часть первого
элемента}
sag^.next:= q; {в адресную часть первого элемента заносим адрес созданного указателя}
//---------сортировка
qq:= sag^.next; //запоминаем адрес первого элемента ЛОС
Whileqq<>Nildo {запускаем цикл выполняющийся пока не будет достигнут конец списка}
begin
s:= qq; //запоминаем адрес первого элемента ЛОС
qq:= qq^.next; {запоминаем адрес следующего элемента
ЛОС}
if (qq<>nil) then {если адрес переменной не равен указателю конца списка}
begin
begin
if (qq^.inf2<s^.inf2) then {если информационная часть следующего элемента меньше чем предыдущего, тогда:}
begin
a:= qq^.inf2; {запоминаем значение информационной части следующего элемента}
qq^.inf2:=s^.inf2; {в информационную часть следующего элемента запоминаем значение информационной части
```

```
предыдущего элемента}
s^.inf2:= a; {в информационную часть предыдущего элемента запоминаем значение следующего
элемента}
end;
end;
end;
end;
//---------конецсортировки
ReadLn(stroka); //вводданныхтипа String
end;
end;
WriteLn('');
WriteLn(rus('ВашитоговыйЛОС:'));
print(sag); //вывод ЛОС на экран
WriteLn(' ');
ifst2=1 then //если ЛОС наполняется первый раз
Gou(w); {процедура перемещения между
процедурами}
End;
Procedure Vopros ( w: WE); {процедура вывода списка вариантов дальнейших вариантов работы в
ЛОС}
Begin
WriteLn(rus('_____'));
WriteLn(rus('Что Вы хотите выполнить дальше?'));
WriteLn('');
WriteLn(rus('2 - Добавить элемент в начало списка'));
Write(rus('3 - Добавить элемент в середину списка, перед указанным ‘));
WriteLn(rus('значением'));
Write(rus('4 - Добавить элемент в середину списка, после указанного ‘));
WriteLn(rus('значения'));
WriteLn(rus('5 - Добавить элемент в конец списка'));
WriteLn(rus(''));
WriteLn(rus('6 - Удалить элемент в начале списка'));
Write(rus('7 - Удалить элемент ЛОС стоящий перед указанным ‘));
WriteLn(rus('значением списка'));
Write(rus('8 - Удалить элемент ЛОС стоящий после указанного “));
WriteLn(rus('значения списка'));
WriteLn(rus('9 - Удалить определенный элемент в списке'));
WriteLn(rus('10 - Удалить элемент в конце списка'));
WriteLn(rus('11 - Удалить первый и последний элементы ЛОС'));
WriteLn('');
WriteLn(rus('12 - ОчиститьЛОС'));
WriteLn('');
WriteLn(rus('13 - Поиск элемента по его значению'));
WriteLn('');
WriteLn(rus('14 - Сортировка элементов'));
WriteLn('');
Write(rus('15 - Подсчитать количество идентичных по содержанию '));
WriteLn(rus('элементов с указанным'));
WriteLn('');
WriteLn(rus('0 - Выход'));
WriteLn('');
WriteLn(rus('_____'));
End;
ProcedureGou( w: WE); {процедура перемещения между
процедурами}
Begin
```

```
WriteLn(rus('Нажмите Ввод для продолжения'));
ReadLn(W); //ждем нажатия Enter
Whilew<>" do {проверяем был ли просто нажат Enter или пользователь что-то ввел. цикл будет работать пока пользователь не нажмет просто Enter, без ввода данных}
begin
WriteLn('');
ifw='' then {если был нажат Enter без ввода данных продолжаем работу процедуры}
else {иначе выдаем сообщение}
WriteLn(rus('Будьте внимательны, нужно просто нажать на ввод!'));
ReadLn(W); //ждемнажатия Enter
end;
WriteLn('');
WriteLn('');
Vopros(w); {процедура вывода списка вариантов дальнейших вариантов работы в ЛОС}
WriteLn;
WriteLn('');
proverka(w); {вход в процедуру проверки корректности ввода данных}
oper:=strtoint(w); {перевод числа из строкового типа данных в целочисленный}
Whileoper=1 do {если ЛОС не пустой то выводим
сообщение}
Begin
Write(rus('Ваш ЛОС не пустой, для создания нового '));
Writeln(rus('удалите все элементы текущего.'));
Writeln(rus('Выберите вариант дальнейших действий.'));
proverka(w); {вход в процедуру проверки корректности ввода данных}
oper:= StrToInt(w); {перевод числа из строкового типа данных в целочисленный}
end;
end;
. . .
Procedure Delet11 (var sag: PE); {процедура удаления первого и последнего, элементов в ЛОС}
var
k, kk, q, qq: PE; //адресные переменные
Begin
WriteLn(rus('Ваш ЛОС выглядел так:'));
WriteLn('');
print(sag); //процедура вывода ЛОС
WriteLn('');
q:= sag^.next; //запоминаем адрес первого элемента
k:= sag^.next; //запоминаем адрес первого элемента
Ifq^.next=Nilthen {если адрес второго элемента содержит указывает на конец списка}
Begin
sag^.next:=q^.next; {в адресную часть первого элемента заносим указатель на конец списка}
dispose(k); //удаляем адрес первого элемента
WriteLn(rus('В ЛОС был только один элемент!')); //выводим сообщение
WriteLn(' ');
End
Else //иначе
Begin
qq:=sag^.next; //запоминаем адрес первого элемента
q:=qq^.next; //запоминаем адрес второго элемента
if (q^.next= Nil) then {если адресная часть второго элемента указывает на конец списка (это означает что в ЛОС только два элемента)}
Begin
sag^.next:=Nil; {в адресную часть начала ЛОС заносим указатель на конец списка}
dispose(q); //удаляем адрес первого элемента
dispose(q); //удаляем адрес второго элемента
End
```

```
Else //иначе
Begin
q:= sag^.next; //запоминаем адрес второго элемента
sag^.next:=q^.next; {в адресную часть первого элемента заносим адрес второго элемента}
dispose(k); //удаляем адрес первого элемента
q:= sag^.next; //запоминаем адрес первого элемента
qq:=q^.next; //запоминаем адрес второго элемента
While (q<>Nil)do {цикл удаления последнего элемента
проходим по списку, до тех пор, пока указатель не будет равен значению, указателя на конец списка}
Begin
kk:=q; //запоминаем адрес текущего элемента
q:=q^.next; //запоминание следующего адреса
k:= qq; //запоминаем адрес следующего элемента
qq:=qq^.next; //запоминание следующего адреса
ifqq=Nilthen {если адресная часть следующего элемента указывает на значение конца списка значит мы нашли адрес последнего элемента ЛОС}
Begin
kk^.next:= qq; {заносим в адресную часть предыдущего элемента значение конца списка}
q:=kk^.next; {запоминание в переменную указателя, значение конца списка для выхода из цикла}
dispose(k); //удаляем адрес последнего элемента
End;
end;
End;
End;
q:=sag^.next; //запоминаем адрес начального элемента
ifq= Nilthen {если адрес начального элемента указывает на конец списка выводим сообщение}
Begin
WriteLn(rus('Ваш итоговый ЛОС:'));
WriteLn('');
print(sag); //процедура вывода ЛОС на экран
WriteLn('');
proverka(w); {вход в процедуру проверки корректности ввода данных}
oper:= StrToInt(w); {перевод числа из строкового типа данных в целочисленный}
ifoper = 0 then //если переменная равна нулю
exit; //завершаем работу программы
While (oper <> 1) do {цикл работает пока переменная не равна
единице}
Begin {выводит сообщение, о не верно выбранной операции}
WriteLn('');
WriteLn(rus('Выбрана не та операция!'));
WriteLn(rus('_____'));
WriteLn('');
WriteLn(rus('Введите номер требуемой операции '));
WriteLn('');
proverka(w); {вход в процедуру проверки корректности ввода данных}
oper:= StrToInt(w); {перевод числа из строкового типа данных в целочисленный}
ifoper = 0 then //если переменная равна нулю
exit; //завершаем работу программы
end;
end
Else
Begin
Write(rus('После удаления первого и последнего элементов Ваш ЛОС’));
Writeln(rus(‘ выглядит так:'));
WriteLn('');
print(sag); //процедура вывода ЛОС на экран
```

```
WriteLn('');
Gou(w); {процедура перемещения между
процедурами}
End;
End;
. . .
Begin //Выполнение самой программы
WriteLn(rus('Лабораторная работа студента группы ВСМ 6 05'));
WriteLn(rus('Антонова А.Е.'));
WriteLn(rus('Номер лабораторной # 1.18'));
WriteLn('');
WriteLn(rus('Для более удобной работы в программе'));
WriteLn(rus('разверните окно программы на максимальный размер'));
WriteLn('');
WriteLn(rus('Нажмите Ввод для продолжения'));
ReadLn(W); //ждем нажатия Enter
Whilew<>'' do {проверяем был ли просто нажат Enter или пользователь что-то ввел. цикл будет работать пока пользователь не нажмет просто Enter, без ввода данных}
begin
WriteLn('');
ifw='' then {если был нажат Enter без ввода данных продолжаем работу процедуры}
else {иначе выдаем сообщение}
WriteLn(rus('Будьте внимательны, нужно просто нажать на ввод!'));
ReadLn(W); //ждемнажатия Enter
end;
WriteLn(' ');
WriteLn(rus('В данной лабораторной, присутствуют следующие'));
WriteLn(rus('возможности работы с ЛОС:'));
WriteLn(rus('_____'));
WriteLn('');
WriteLn(rus('1 - Создать ЛОС'));
WriteLn('');
WriteLn(rus('2 - Добавить элемент в начало списка'));
Write(rus('3 - Добавить элемент в середину списка, перед указанным’));
WriteLn(rus(‘ значением'));
Write(rus('4 - Добавить элемент в середину списка, после указанного’));
WriteLn(rus(‘ значениея'));
WriteLn(rus('5 - Добавить элемент в конец списка'));
WriteLn(rus(''));
WriteLn(rus('6 - Удалить элемент в начале списка'));
Write(rus('7 - Удалить элемент ЛОС стоящий перед указанным ’));
WriteLn(rus(‘значением списка'));
Write(rus('8 - Удалить элемент ЛОС стоящий после указанного ’));
WriteLn(rus(‘значением списка'));
WriteLn(rus('9 - Удалить определенный элемент в списке'));
WriteLn(rus('10 - Удалить элемент в конце списка'));
WriteLn(rus('11 - Удалить первый и последний элементы ЛОС'));
WriteLn('');
WriteLn(rus('12 - ОчиститьЛОС'));
WriteLn('');
WriteLn(rus('13 - Поиск элемента по его значению'));
WriteLn('');
WriteLn(rus('14 - Сортировка элементов'));
WriteLn('');
Write(rus('15 - Подсчитать количество идентичных по содержанию’));
WriteLn(rus(‘элементов'));
WriteLn(rus(' с указанным'));
```

```
WriteLn('');
WriteLn(rus('0 - Выход'));
WriteLn('');
WriteLn(rus('_____'));
WriteLn('');
Write(rus('Вы будете работать с числами (Тип Integer) - 1 '));
WriteLn(rus('или со строками (Тип String) - 2'));
While (st <> 1) and (st <> 2) do {цикл проверки ввода номера варианта
работы}
Begin
WriteLn(' ');
WriteLn(rus('Выберите номер нужного варианта работы'));
WriteLn(' ');
ReadLn(w);
if (w <> '1') and (w <> '2') then //если выбран не тот вариант работы
begin //выводим сообщение
WriteLn(' ');
WriteLn(rus('Не правильно выбран номер варианта работы'));
end
else
st:= StrToInt(w); {перевод числа из строкового типа данных в целочисленный}
end;
WriteLn(' ');
Create2(sag); //процедурасозданияЛОС
st2:=1;
WriteLn('');
WriteLn(rus('Нажмите Ввод для продолжения'));
ReadLn(W); //ждем нажатия Enter
Whilew<>'' do {проверяем был ли просто нажат Enter или пользователь что-то ввел. цикл будет
работать пока пользователь не нажмет просто Enter, без ввода данных}
begin
WriteLn('');
ifw='' then {если был нажат Enter без ввода данных продолжаем работу процедуры}
else //иначе выдаем сообщение
WriteLn(rus('Будьте внимательны, нужно просто нажать на ввод!'));
ReadLn(W); //ждемнажатия Enter
end;
WriteLn('');
WriteLn('');
Vopros(w); {процедура вывода списка вариантов дальнейших вариантов работы в ЛОС}
WriteLn;
Begin {Определение какая операция должна
быть запущена}
proverka(w); {вход в процедуру проверки корректности ввода данных}
oper:= StrToInt(w); {перевод числа из строкового типа данных в целочисленный}
if oper=1 then {если была выбрана первая операция, а ЛОС не пуст, выводим сообщение}
Begin
Write(rus('Ваш ЛОС не пустой, для создания нового '));
Writeln(rus('удалите все элементы текущего.'));
Writeln(rus('Выберите вариант дальнейших действий.'));
proverka(w); {вход в процедуру проверки корректности ввода данных}
oper:= StrToInt(w); {перевод числа из строкового типа данных в целочисленный}
end;
whileoper<>0 do {цикл работает пока не выбрана операция выхода из программы}
begin
if (oper>=1) and (oper<=15) then {проверка проходит между пятнадцатью
операциями}
```

```
begin
if oper= 1 then
begin
WriteLn;
WriteLn('');
Write(rus('Вы будете работать с числами (Тип Integer) - 1 '));
WriteLn(rus('или со строками (Тип String) - 2'));
st:=0;
While (st <> 1) and (st <> 2) do {цикл проверки ввода номера варианта
работы}
Begin //выводим сообщение
WriteLn(' ');
WriteLn(rus('Выберите номер нужного варианта работы'));
WriteLn(' ');
ReadLn(w);
if (w <> '1') and (w <> '2') then //если выбран не тот вариант работы
begin
WriteLn(' ');
WriteLn(rus('Не правильно выбран номер варианта работы'));
end
else
st:= StrToInt(w); {перевод числа из строкового типа данных в целочисленный}
end;
Create2(sag); //процедура создания ЛОС
WriteLn;
end;
. .
if oper= 11 then
begin
WriteLn;
Delet11(sag);
WriteLn;
end;
. . .
else
Begin
Writeln(rus('Не корректный ввод варианта операции '));
Writeln(rus('Попробуйте еще раз'));
proverka(w); {вход в процедуру проверки корректности ввода данных}
oper:= StrToInt(w); {перевод числа из строкового типа данных в целочисленный}
Whileoper=1 do {если ЛОС не пустой то выводим
сообщение}
Begin
Write(rus('Ваш ЛОС не пустой, для создания нового '));
Writeln(rus('удалите все элементы текущего.'));
Writeln(rus('Выберите вариант дальнейших действий.'));
proverka(w); {вход в процедуру проверки корректности ввода данных}
oper:= StrToInt(w); {перевод числа из строкового типа данных в целочисленный}
end;
end;
end;
if oper= 0 then //если переменная равна нулю, то
begin
exit; //выход из программы
end;
end;
end. //Конец программы
```

4 Методические указания по итоговому контролю

Итоговый контроль знаний по дисциплине «Структуры и алгоритмы данных» проводится в форме экзамена. Для подготовки к итоговому контролю знаний по дисциплине «Структуры и алгоритмы данных» обучающиеся используют перечень вопросов, приведенный в фонде оценочных средств. Экзамен проводится в устной форме. В экзаменационный билет включен один теоретический вопрос. На подготовку студенту отводится 20-25 минут. На дифференцированном зачете ответы обучающегося оцениваются с учетом их полноты, правильности и аргументированности с учетом шкалы оценивания.

Оценка «отлично» выставляется студенту, если он глубоко и прочно усвоил программный материал, исчерпывающе, последовательно, четко и логически его излагает, умеет тесно увязывать теорию с практикой, свободно справляется с вопросами и другими видами применения знаний, причем не затрудняется с ответом при видоизменении заданий, использует в ответе профессиональные термины, правильно обосновывает принятое решение.

Оценка «хорошо» выставляется студенту, если он твердо знает материал, грамотно и по существу излагает его, не допуская существенных неточностей в ответе на вопрос, правильно применяет теоретические положения при решении практических вопросов, владеет необходимыми навыками и приемами их выполнения.

Оценка «удовлетворительно» выставляется студенту, если он имеет знания только основного материала, но не усвоил его деталей, допускает неточности, недостаточно правильные формулировки, нарушения логической последовательности в изложении программного материала.

Оценка «неудовлетворительно» выставляется студенту за отсутствие знаний по дисциплине, представления по вопрос, непонимание материала по дисциплине, наличие коммуникативных «барьеров» в общении, отсутствие ответа на предложенный вопрос.

5 Учебно-методическое обеспечение дисциплины

5.1 Основная литература

1. Кузниченко, М. А. Динамические структуры данных [Электронный ресурс] : учебное пособие / М. А. Кузниченко. - Электрон. текстовые дан. (1 файл: Kb). - Орск : ОГТИ, 2011. -Adobe Acrobat Reader [Режим доступа http: //artlib.osu.ru/web/books/metod_all/3448_20130201.pdf]

5.2 Дополнительная литература

2. Стрекалова, И.И. Структуры и алгоритмы обработки данных: методические указания / И.И. Стрекалова; Оренбургский гос. ун-т. – Оренбург: ОГУ, 2012. – 107 с. [Режим доступа http://artlib.osu.ru/web/books/metod_all/3302_20121022.pdf]

Гагарина, Л. Г. Алгоритмы и структуры данных [Текст] : учебное пособие / Л. Г. Гагарина, В. Д. Колдаев. - Москва : Финансы и статистика, 2009. - 304 с. : ил. - ISBN 978-5-279-03351-5. (аб. ТБ-15)

5.3 Периодические издания

1. Журнал «Вестник компьютерных и информационных технологий »
2. Журнал «Информационные технологии и вычислительные системы»
3. Журнал «Стандарты и качество»
4. Журнал «Прикладная информатика»

5.4 Интернет-ресурсы

5.4.1 Современные профессиональные базы данных и информационные справочные системы:

1. Информационная система «Единое окно доступа к образовательным ресурсам» - http://window.edu.ru/
2. КиберЛенинка - https://cyberleninka.ru/
3. Университетская информационная система Россия – uisrussia.msu.ru
4. Бесплатная база данных ГОСТ – https://docplan.ru/

5.4.2 Тематические профессиональные базы данных и информационные справочные системы:

1. Портал искусственного интеллекта – AIPortal
2. Web-технологии – Web-технологии
3. Электронная библиотека Института прикладной математики им. М.В. Келдыша – Электронная библиотека публикаций Института прикладной математики им. М.В. Келдыша РАН

5.4.3 Электронные библиотечные системы

1. ЭБС «Университетская библиотека онлайн» – http://www.biblioclub.ru/
2. ЭБС Znanium.com – https://znanium.com/

Дополнительные Интернет-ресурсы

1. https://www.ixbt.com - Интернет-издание о компьютерной технике, информационных технологиях и программных продуктах. На сайте публикуются новости IT, статьи с обзорами и тестами компьютерных комплектующих и программного обеспечения.
2. http://www.intuit.ru – ИНТУИТ – Национальный открытый университет.

5.5 Программное обеспечение, профессиональные базы данных и информационные справочные системы современных информационных технологий

Тип программного обеспечения	Наименование	Схема лицензирования, режим доступа
Интернет-браузер	Internet Explorer	Является компонентом операционной системы Microsoft Windows
	Opera	Бесплатное ПО, http://www.opera.com/ru/terms
	Mozilla Firefox	Свободное ПО, https://www.mozilla.org/en-US/foundation/licensing/
	Google Chrome	Бесплатное ПО, http://www.google.com/intl/ru/policies/terms/
Векторный графический редактор, редактор диаграмм и блок-схем	Microsoft Visio Standard 2007	Сертификат Microsoft Open License № 46284547 от 18.12.2009 г., академическая лицензия на рабочее место
Операционная система	Microsoft Windows	Подписка Enrollment for Education Solutions (EES) по государственному контракту: ➢ № 2К/17 от 02.06.2017 г.;
Офисный пакет	Microsoft Office	

6 Материально-техническое обеспечение дисциплины

Учебные аудитории для проведения занятий лекционного типа, семинарского типа, для проведения групповых и индивидуальных консультаций, текущего контроля и промежуточной аттестации. Для проведения практических работ используются компьютерный класс (ауд. № 4-113, 4-116, 4-117), оборудованный средствами оргтехники, программным обеспечением, персональными компьютерами, объединенными в сеть с выходом в Интернет.

Аудитории оснащены комплектами ученической мебели, техническими средствами обучения, служащими для представления учебной информации большой аудитории.

Помещения для самостоятельной работы обучающихся оснащены компьютерной техникой, подключенной к сети «Интернет», и обеспечением доступа в электронную информационно-образовательную среду Орского гуманитарно-технологического института (филиала) ОГУ (ауд. № 4-307).

Наименование помещения	Материальное-техническое обеспечение
Учебные аудитории: - для проведения занятий лекционного типа, семинарского типа, - для групповых и индивидуальных консультаций; - для текущего контроля и промежуточной аттестации	Учебная мебель, классная доска, мультимедийное оборудование (проектор, экран, ноутбук с выходом в сеть «Интернет»)
Компьютерные классы № 4-113, 4-116, 4-117	Учебная мебель, компьютеры (29) с выходом в сеть «Интернет», проектор, экран, лицензионное программное обеспечение
Помещение для самостоятельной работы обучающихся, для курсового проектирования (выполнения курсовых работ)	Учебная мебель, компьютеры (3) с выходом в сеть «Интернет» и обеспечением доступа в электронную информационно-образовательную среду Орского гуманитарно-технологического института (филиала) ОГУ, программное обеспечение

Для проведения занятий лекционного типа используются следующе наборы демонстрационного оборудования и учебно-наглядные пособия:

- презентации к курсу лекций.