



**«Московский государственный технический университет
имени Н.Э. Баумана»
(национальный исследовательский университет)**

Факультет «Информатика и системы управления»
Кафедра «Компьютерные системы и сети»

Г.С. Иванова, Т.Н. Ничушкина, Е.К. Пугачев

Методы обработки данных и оценки программ

Электронное учебное издание

Методические указания по выполнению лабораторных работ
по дисциплине «Технология разработки программных систем»

Москва
МГТУ им. Н.Э. БАУМАНА
2018

УДК 004.021

Рецензент:

Г.С. Иванова, Т.Н. Ничушкина, Е.К. Пугачев

Методы обработки данных и оценки программ. Электронное учебное издание. Методическое пособие по выполнению лабораторных работ по дисциплине «Технология разработки программных систем» - М.: МГТУ имени Н.Э. Баумана, 2018. 63 с.

Издание содержит: описание основных структур и методов обработки данных, критерии оценки алгоритмов и структур данных, примеры структур данных, способы оценки и повышения эффективности программ, способы тестирования программ, варианты заданий и порядок выполнения лабораторных работ, предусмотренных учебным планом МГТУ им. Н.Э.Баумана. Для студентов МГТУ имени Н.Э. Баумана направления «Информатика и вычислительная техника».

**Иванова Галина Сергеевна
Ничушкина Татьяна Николаевна
Пугачев Евгений Константинович**

Методы обработки данных и оценки программ.

© 2018 МГТУ имени Н.Э. Баумана

Оглавление

Введение.....	7
Глава 1. Структуры и методы обработки данных	8
1.1 Классификация абстрактных структур данных	8
1.2 Структуры данных с неявными связями	9
1.3 Структуры данных с явными связями	12
1.3.1 Линейные структуры с явными связями	12
1.3.2 Древовидные структуры с явными связями.....	13
1.3.3 Сетевые структуры с явными связями	14
1.4 Методы обработки данных	15
1.4.1 Методы упорядочения данных.....	15
1.4.2 Методы поиска в последовательных структурах данных	17
1.5 Корректировка структур данных.....	20
1.6 Методы обработки списковых структур данных.....	21
1.6.1 Метод К- и А- индексов	21
1.6.2 Гнездовой способ организации	22
1.7 Обработка древовидных списков	23
1.8 Выбор способа реализации абстрактных структур данных.....	26
1.9 Вопросы для самопроверки	32
Глава 2. Оценка эффективности и качества программы	33
2.1 Основные понятия	33
2.2 Способы уменьшения времени выполнения	34
2.3 Способы экономии памяти.	35
2.4 Критерии качества программных продуктов.	36
2.5 Вопросы для самопроверки	41
Глава 3. Тестирование программного обеспечения.....	42
3.1 Основные принципы тестирования.....	42
3.2 Ручное тестирование программных продуктов	43
3.2.1 Инспекции исходного текста.....	43
3.2.2 Сквозные просмотры.....	44
3.2.3 Проверка за столом.....	45
3.2.4 Оценка посредством просмотра.....	45
3.2.5 Перечень вопросов для структурного контроля текста	45
3.3 Тестирование по принципу «белого ящика».....	47
3.3.1 Покрытие операторов.....	48
3.3.2 Покрытие решений (переходов).....	48
3.3.3 Покрытие условий	49
3.3.4 Покрытие решений/условий.....	50
3.3.5 Комбинаторное покрытие условий.....	50
3.4 Тестирование по принципу «черного ящика».....	51
3.4.1 Эквивалентное разбиение	52
3.4.2 Анализ граничных значений	54
3.4.3 Анализ причинно-следственных связей.....	56
3.4.4 Предположение об ошибке.....	57
3.4.5 Примеры тестирования	57
3.5 Общая стратегия тестирования	60
3.6 Вопросы для самопроверки	61
Заключение	62
Список литературы	63

Предисловие

Методическое пособие предназначено для получения теоретических сведений, связанных с разработкой и оценкой программных продуктов.

Методическое пособие составлено в соответствии с самостоятельно устанавливаемым образовательным стандартом (СУОС), основной образовательной программой по направлению подготовки 09.03.01 «Информатика и вычислительная техника» бакалавров.

Методическое пособие включает материал, позволяющий подготовиться к выполнению лабораторных работ по дисциплине «Технология разработки программных систем» кафедры ИУ6. *Цель выполнения лабораторных работ* – приобретение навыков при выборе структур данных и методов их обработки, а также тестирования и оценки качества программ.

Методическое пособие поможет овладеть:

- методиками расчета основных параметров, на основании которых осуществляется выбор структур данных и методов их обработки применительно к конкретной задаче;
- методами тестирования программных продуктов;
- методиками оценки качества программ, оценки и повышения эффективности программ.

В результате использования методического пособия и выполнения лабораторных работ студент должен знать:

- основные методы реализации операций, таких как поиск, упорядочение и корректировка;
- основные способы организации данных, касающиеся как статических, так и динамических структур;
- методы тестирования, используемые на разных этапах разработки;
- методы оценки эффективности и качества программ, а также способы их повышения.

При выполнении лабораторных работ приобретает следующие умения:

- проводить сравнительный анализ структур данных;
- проводить оценку методов обработки данных;
- проводить тестирование программного продукта;
- определять способы повышения эффективности и качества программы.

В результате студент получает следующие навыки:

- навыки обоснования выбора структур данных и алгоритмов их обработки;
- навыки проведения эффективного тестирования;
- навыки оценки эффективности программ;
- навыки оценки качества программ.

Для успешного выполнения лабораторных работ необходимо предварительное освоение следующих дисциплин:

1. Основы программирования.
2. Информатика.
4. Объектно-ориентированное программирование.
5. Языки интернет-программирования.

Цели и задачи методического пособия.

Основной целью методического пособия является проверка студента или подтверждение студентом соответствия своих знаний и умений квалификационным требованиям, предъявляемым к бакалаврам по направлению 09.03.01 «Информатика и вычислительная техника». Перечень этих требований приведен в СУОС и основной образовательной программе по данному направлению.

Задачей методического пособия является подтверждение студентами способности разрабатывать компоненты программных комплексов и баз данных, используя современные инструментальные средства и технологии программирования, а также осуществлять постановку и выполнять эксперименты по проверке корректности и эффективности программ.

Методические указания по освоению материала

Основной целью освоения материала методического пособия является самостоятельное изучение студентами важных разделов дисциплины «Технология разработки программных систем».

Для успешного освоения материала необходимо внимательно разбирать примеры, целесообразно синтезировать и прорабатывать аналогичные примеры. Ответы на вопросы позволят обучающемуся самому оценить степень понимания и усвоения теоретических положений, методик и методов. Проработка материала обеспечит качественное выполнение лабораторных работ, связанных с постановкой и решением задач проектирования, тестирования и оценки программ.

Для оценки степени понимания и усвоения теоретических положений, методик и методов, приобретения умений и навыков служит информативный отчет по каждой лабораторной работе, где полно и точно представлены результаты и их защита.

Введение

Существуют разные классы программных продуктов и разные технологии их разработки. Независимо от класса программного продукта стоит задача создания качественного программного продукта. Также существуют общие вопросы, которые возникают независимо от того, какая технология будет использоваться. Методическое пособие позволяет найти ответы на ряд таких вопросов. В частности, при разработке программ необходимо найти ответы, связанные с выбором структур данных и методов их обработки. Правильные решения позволяют обеспечить эффективную работу разрабатываемого обеспечения, а также позволяют экономно использовать ресурсы вычислительной системы, например, ресурсы оперативной и внешней памяти, а также процессорное время. Важно отметить, что выбор структуры данных зависит от многих факторов, например, от режима работы разрабатываемой системы, от решаемых задач предметной области и др.

В первой части методического пособия на проработку выносятся способы организации структур данных, способы оценки применимости различных структур применительно к конкретным задачам, а также способы оценки алгоритмов, реализующие различные методы обработки данных для операций поиска, упорядочения и корректировки данных. Например, студентам предлагается изучить основные методы поиска, после чего они определяют применимость методов поиска к различным структурам данных, учитывая способ доступа и др. Аналогично студенты могут проработать методы сортировки и корректировки.

Во второй части методического пособия на проработку выносятся способы оценки эффективности и качества программ. Также студенты могут освоить способы повышения эффективности и качества.

В третьей части методического пособия на проработку выносятся методы тестирования программных продуктов.

Глава 1. Структуры и методы обработки данных

Задачи выбора структур данных и методов их обработки являются очень важными. При решении данных задач требуется высокая квалификация разработчиков. Исходными составляющими для решения задач являются описание набора и типов хранимых данных, перечень операций над данными и др.

При решении вышеуказанных задач необходимо выполнить следующие действия:

- Разработать логическую структуру данных.
- Определить способ реализации структуры данных.
- Выбрать эффективный метод поиска с учетом способа доступа к элементу данных в структуре.
- Определить эффективный способ упорядочивания данных.
- Выбрать метод корректировки данных и оценить его влияние на операции поиска и упорядочения и др.

1.1 Классификация абстрактных структур данных

Под *структурой данных* понимают совокупность правил и ограничений, которые отражают связи, существующие между отдельными частями (элементами) данных.

В процессе проектирования программного продукта разработчик обычно создает модель представления данных, не зависящую от реализации, используя при этом абстрактные структуры данных. Классификация таких структур приведена на рисунке 1.1. Модель представления данных может быть как простой, так и комбинированной (включающей различные абстрактные структуры).

СТРУКТУРЫ ДАННЫХ

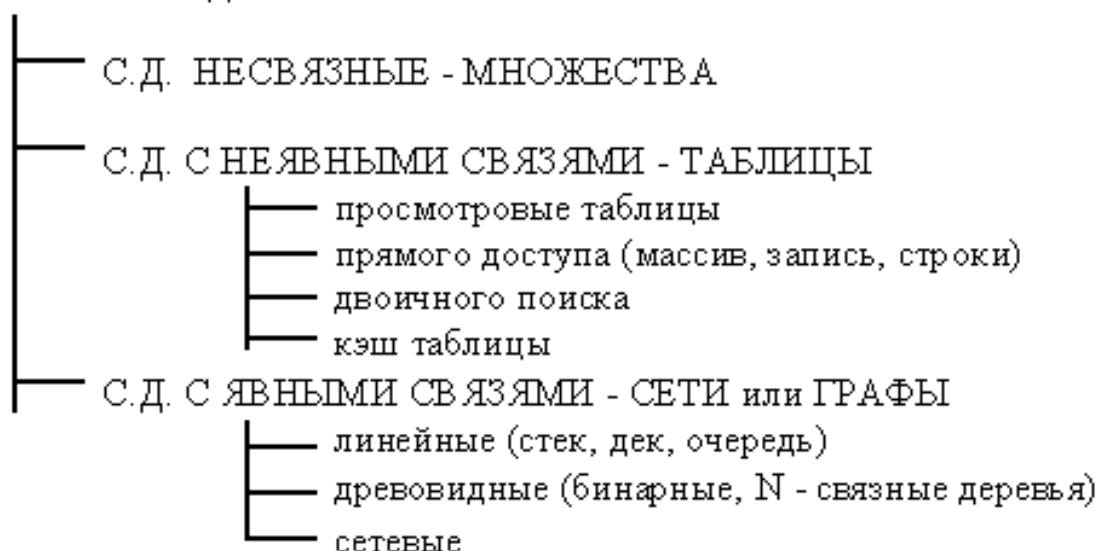


Рисунок 1.1 – Классификация абстрактных структур данных

К структурам данных, где элементы не связаны, относится множество, которое представляет собой неупорядоченную совокупность взаимно независимых элементов. Используется, если необходимо определять принадлежность какого-либо элемента множеству аналогичных элементов. В языках программирования применительно к множествам предусмотрен ряд операций.

В структурах данных, где элементы связаны неявно имеется определенный порядок расположения этих элементов, например, в виде таблицы. При необходимости можно определить положение одного элемента относительно другого.

В структурах с явными связями каждый элемент имеет дополнительно служебную информацию, которая используется для непосредственного указания ссылок на другой элемент.

1.2 Структуры данных с неявными связями

Структуры данных с неявными (статическими) связями элементов – таблицы – множество элементов, с каждым из которых связан ключ. Используются обычно для хранения данных. Основная операция – поиск информации по ключу. Различают четыре типа таблиц (просмотровые, прямого доступа, двоичного поиска и таблицы с перемешиванием) с различной *эффективностью* поиска

данных. Эффективность поиска в таблицах различных типов оценивают, используя две характеристики:

S – длина поиска – количество записей, которые необходимо просмотреть, чтобы найти запись с заданным ключом;

L – средняя длина поиска при постоянной частоте обращения.

Просмотровые таблицы – таблицы, в которых записи просматривают последовательно $S_i = i$; $L = (N+1)/2$, где N – количество записей.

Недостаток: очень велико среднее время поиска. Достоинство: простота создания и добавления данных.

Таблицы прямого доступа – таблицы, в которых ключ – однозначно определяет адрес. Чаще всего ключом является натуральное число или ключ приводит к натуральному числу – номеру элемента.

В таблицах прямого доступа ключ можно не хранить. Длина поиска и средняя длина поиска таблиц прямого доступа соответственно равны:

$$S = 1, L = 1.$$

Недостаток: подобная организация таблицы редко применима, т. к. ключи часто бывают непоследовательными или изменяются в недопустимо больших диапазонах. Достоинство: быстрый поиск.

Частными случаями таблиц прямого доступа являются:

Массив – множество элементов, расположенных таким образом, что упорядоченное множество целых чисел (ключ) однозначно определяет позицию каждого элемента, а также обеспечивает возможность организации прямого доступа к каждому элементу. Если упорядоченное множество содержит N чисел, то массив N -мерный. Числа, входящие в упорядоченное множество, называются индексами. Каждый индекс имеет свой диапазон изменения.

Строка – одномерный массив символов.

Запись – конечное упорядоченное множество элементов, в общем случае различных типов. Ключом является порядковый номер поля или его имя.

Логический порядок элементов в просмотровых таблицах и таблицах прямого доступа при реализации обычно совпадает с физическим порядком распо-

ложения элементов. Элементами являются записи. Записи могут быть фиксированной, переменной и неопределенной длины. В зависимости от типа записи таблицы реализуются массивом записей или массивом векторов (статических или динамических), в котором основной массив содержит адреса векторов записей.

Для массива записей фиксированной длины адреса промежуточных записей задаются формулой:

$$A_i = A_1 + (i-1) * l,$$

где

A_i – адрес i -ой записи;

A_1 – адрес первой записи;

l – длина записи.

Достоинство: экономное использование памяти, т. к. записи располагаются одна за другой без промежутков. Недостаток: необходимо заранее знать общее число записей, их длину и диапазон изменения ключей.

Таблицы с перемешиванием (кэш-таблицы) – попытка построить таблицу с $L \cong 1$ при нерегулярных ключах. Для построения кэш-таблицы выбирается функция перемешивания (кэш-функция), *определенная на множестве значений ключа*, которая разбивает весь диапазон ключей на эквивалентные классы максимально равномерно. Далее для класса ключей адрес определяется как при прямом доступе, а внутри класса – последовательно (см. пример на рисунке 1.2).

Недостаток: часто сложно построить кэш-функцию, так как при слишком большом количестве классов часть из них может оказаться пустыми, что увеличивает объем используемой памяти. Оптимальным считается заполнение таблицы на 60-70%. Достоинство: среднее время поиска близко к 1.

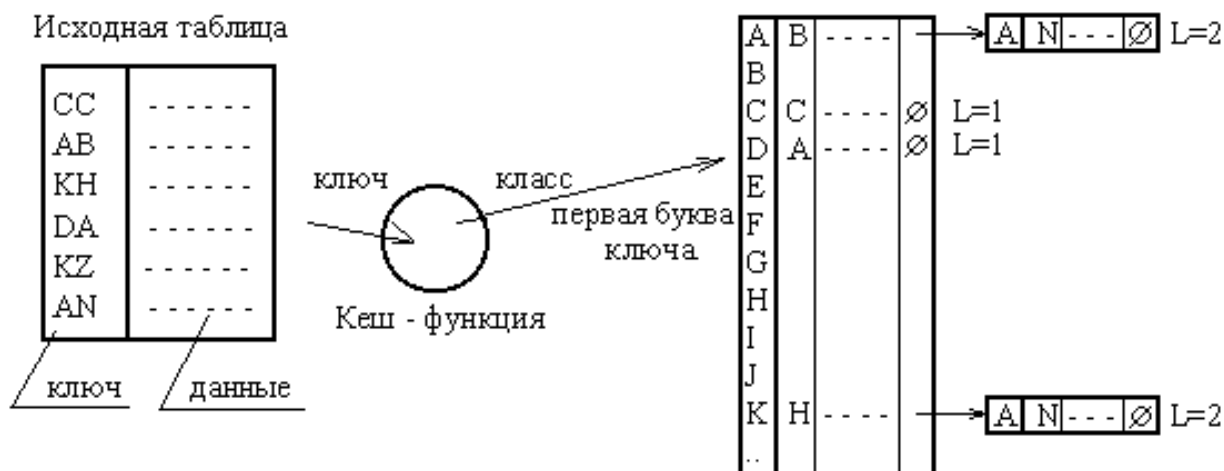


Рисунок 1.2 - Пример построения кэш-таблицы

Таблицы двоичного поиска – записи в таблице располагают по возрастанию или убыванию ключей, что делает применимым двоичный (дихотомический) поиск (см. далее), при котором $L = (\log_2 N)/2$.

Недостаток: добавляя записи необходимо сохранять сортировку. Достоинство: среднее время поиска существенно меньше, чем при последовательном просмотре.

Реализуются массивом или бинарным деревом.

1.3 Структуры данных с явными связями

Структуры с явными (динамическими) связями – структуры, каждый элемент которых содержит указатели на следующие элементы (связи представлены в явном виде). В основном используются при моделировании данных с изменяемыми связями между элементами, т.к. несут в явном виде информацию о связях.

1.3.1 Линейные структуры с явными связями

Линейные структуры с явными связями – элемент расположен между двумя другими (если он не первый и не последний) – изменяемая последователь-

ность элементов. Такие структуры могут быть реализованы в виде изменяемых векторов или с использованием списков (одно-, двухсвязных или кольцевых).

Частные случаи:

Очередь – последовательность элементов, добавление элементов к которой осуществляется в конец, а удаление – из начала.

Стек – последовательность элементов, добавление и удаление элементов в которую осуществляется с одного конца.

Дек – последовательность элементов, добавление и удаление элементов в которую возможны с любого конца.

1.3.2 Древовидные структуры с явными связями

Древовидные структуры с явными связями – записи располагаются по уровням следующим образом:

- на 1-м уровне расположена только одна запись (корень дерева);
- любая запись i -го уровня связана (адресуется) только с одной записью $(i-1)$ -го уровня.

Записи i -го уровня, адресованные общей записью $(i-1)$ -го уровня, образуют группу этой записи. Максимальное число записей в группе называют порядком (степенью) дерева. Число занятых уровней называют рангом.

Обычно реализуются на двусвязных или n -связных списках. Сбалансированные деревья могут реализовываться массивом. Возможна реализация с использованием матрицы связности, как сетевой структуры.

Двоичное дерево – каждая вершина адресуется 0-2 вершин следующего уровня. N -мерное дерево – каждая вершина указывает на 0- n вершин более низкого уровня.

1.3.3 Сетевые структуры с явными связями

Сетевые структуры с явными связями – сети или графы. Граф – совокупность конечного множества вершин и конечного множества неупорядоченных пар вершин, называемых ребрами. В ориентированном графе пары вершин упорядочены. В этом случае вершины называются узлами, а ребра дугами.

Реализуются различными способами: от матриц связности или инцидентности до n -связных списков (см. рисунок 1.8).

1.4 Методы обработки данных

1.4.1 Методы упорядочения данных

Частными случаями упорядочения являются сортировки (по возрастанию, по убыванию, по не возрастанию, по не убыванию). Понятие *сортировка* обычно применяют к линейным структурам данных. Понятие *упорядочение* используют применительно к нелинейным структурам данных, например, к деревьям.

1.4.1.1 Методы упорядочения обменом

Метод, базирующийся на попарном сравнении соседних элементов, заключается в следующем. Сравнивается первый элемент со вторым и, если он больше (меньше), то элементы меняются местами. Далее второй элемент сравнивается с третьим и т. д. После анализа последнего элемента процесс повторяется с начала. Количество сравнений определяется выражением:

$$C = N(N-1), \text{ где } N - \text{ количество элементов.}$$

Метод, базирующийся на поиске минимального (максимального) элемента, заключается в следующем. Определяется минимальный (максимальный) элемент всего набора и осуществляется обмен с первым элементом. Затем определяется наименьший (наибольший) элемент из оставшегося набора и меняется местами со вторым и т. д. Количество сравнений по данному методу определяется по формуле: $C = N(N-1)/2$.

1.4.1.2 Метод вставки

Элемент массива a_i (начиная со второго) сравнивается последовательно с предшествующими a_j , где $j = i-1, i-2, \dots$ до тех пока не будет найден элемент с меньшим значением, чем a_i . Пусть этот элемент с номером j , где $j < i$. Тогда все элементы с номерами $j+1, \dots, i-1$ сдвигаются на одну позицию, а i -й элемент ставится на место $(j+1)$ -го элемента. Если все впереди стоящие элементы больше i -

го, то они сдвигаются на одну позицию, а i -й элемент ставится на первое место. Количество сравнений определяется по формуле:

$$C = N(N-1)/4$$

1.4.1.3 Метод Шелла

Метод Шелла состоит в том, что упорядочиваемый массив делится на ряд групп, каждая из которых упорядочивается методом вставки. Количество операций сравнений оценивается следующим образом: $C \leq 0,5 N^{3/2}$.

1.4.1.4 Метод квадратичной выборки

Упорядочиваемый массив, состоящий из N элементов, делится на $\sqrt{N}$ групп по $\sqrt{N}$ элементов в каждой. Для каждой группы необходимо выделить поле памяти, достаточное для размещения значения ключевого признака. Совокупность таких полей образует зону линейного накопления S_N . В результате просмотра элементов каждой группы определяется наименьший элемент, который заносится в соответствующее поле зоны линейного накопления S_N . Затем, просматриваются элементы в S_N , и наименьший из элементов заносится в зону формирования S_R . Далее осуществляется просмотр в группе, элемент которой записан в зону S_R . Наименьший элемент из оставшихся в группе заносится в зону накопления S_N на место перенесенного в зону формирования S_R . После этого снова просматриваются элементы в S_N , и наименьший из элементов заносится в зону формирования S_R . Процесс повторяется, пока в зоне S_N не останется элементов. На рисунке 1.3 приведен пример схемы сортировки по методу квадратичной выборки. Количество сравнений в процессе сортировки: $C = (N-1)\sqrt{N}/2$.

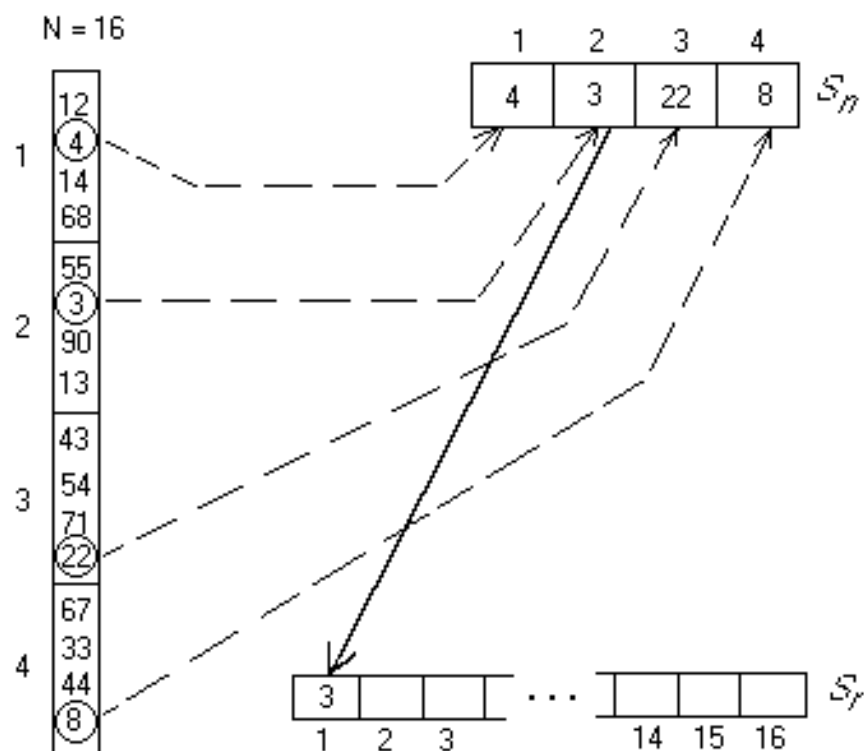


Рисунок 1.3 - Схема сортировки по методу квадратичной выборки

1.4.1.5 Метод слияния

В основе метода слияния лежит процедура слияния двух отсортированных массивов А и В. Слияние заключается в поочередной пересылке элементов из массивов А и В в аналогично отсортированный массив С. Порядок пересылки зависит от результата сравнения очередных элементов массивов А и В.

Упорядочение слиянием состоит в том, что обрабатываемый массив разделяется на равные группы элементов. Группы упорядочиваются, а затем попарно сливаются, образуя новые группы. Элементы внутри групп упорядочиваются методом вставки.

1.4.2 Методы поиска в последовательных структурах данных

Поиском называется процедура выделения из множества записей подмножества, элементы которого удовлетворяют заранее поставленному условию. Поисковым признаком может служить номер записи в массиве или значение ключа.

Условия поиска могут быть различными: по совпадению, по попаданию в интервал, по удовлетворению арифметическому условию, по удовлетворению семантическому условию, по нескольким условиям. Если задано только одно значение признака поиска, то такой поиск называется единичным; если же задано множество признаков поиска, то это групповой поиск.

Эффективность различных алгоритмов поиска оценивается количеством сравнений пар признаков, необходимым для выполнения условия поиска.

1.4.2.1 Метод последовательного поиска

В основе метода последовательный просмотр записей массива с целью отыскания записей, для которых значение ключевого признака p совпадает со значением признака поиска q . Метод является универсальным в том смысле, что применим как к упорядоченным, так и к неупорядоченным массивам данных.

При реализации метода необходимо учитывать дополнительные условия:

- *массив неупорядочен и неизвестно число элементов, удовлетворяющих поисковому признаку;

- *массив неупорядочен, но известно число элементов, удовлетворяющих поисковому признаку;

- *массив упорядочен (если в массиве имеется несколько записей с одинаковыми значениями ключевых признаков, то упорядоченность массива имеет существенное значение).

Среднее число сравнений для реализации данного метода вычисляется следующим образом

$$C = (N+1)/2, \text{ где } N - \text{число записей в массиве.}$$

1.4.2.2 Метод двоичного поиска

Существует группа методов поиска, в основе которых лежит деление массива на части. Метод деления на две части называется двоичным (дихотомическим) поиском.

Пусть массив упорядочен и длина его известна. На первом шаге сравнивается значение ключевого признака средней записи p_i . Если $p_i = q$, то после проверки соседних элементов (они могут иметь тоже значение ключевого признака) поиск прекращается. Если $p_i > q$, то дальнейший поиск будет осуществляться в 1-й половине массива. Если $p_i < q$, то поиск будет продолжаться во 2-й половине массива. Затем в выделенном подмассиве определяется средняя запись и производится сравнение p_i с q и т. д.

Метод называют двоичным в связи с тем, что после каждого сравнения принимается одно из двух альтернативных решений.

Среднее число операций сравнения при поиске в массиве равно

$$C_{cp} = [(N+1) \log_2 (N+1)]/N - 1$$

1.4.2.3 Метод вычисления адреса

Метод вычисления адреса (адресный поиск) применяется к упорядоченным массивам, если значения ключевых признаков не повторяются и их число не превышает числа записей в массиве. При этом записи в массиве должны быть фиксированной длины. Номера записей и числовые значения ключевых признаков связаны между собой адресной функцией: $i = f(p)$, где i – номер записи (или адрес элемента в памяти); p – значение ключевого признака.

Вид функции может быть любым и выбирается при разработке системы поиска. Простейшая адресная функция имеет вид: $i = kp - a$, где a – константа и для простоты обработки берется равной $a = kP_{\min} - 1$, где $P_{\min}$ – минимальное значение ключевого признака; k – целочисленная константа.

На рисунке 1.4 представлен пример использования данного метода для быстрого поиска нужного гнезда сотрудников определенного возраста.

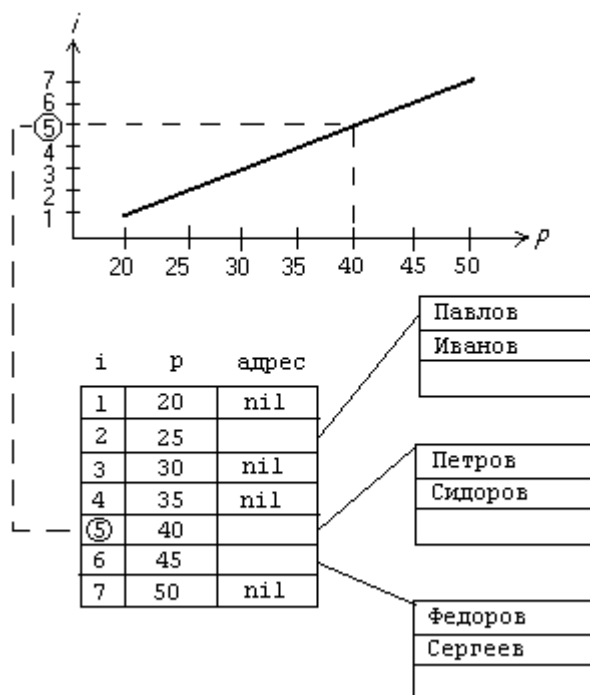


Рисунок 1.4 - Пример использования метода вычисления адреса

1.5 Корректировка структур данных

Корректировка представляет собой процедуру внесения изменений в структуру данных. Различают следующие виды корректировки: вставка (добавление), аннулирование (удаление), замена элементов.

Опишем некоторые особенности корректировки последовательных структур данных, реализованных в виде сортированных массивов. При необходимости сохранения сортировки для вставки требуется наличие свободных участков памяти между записями массива для размещения вставляемых записей. Заранее спланировать необходимый объем памяти и зарезервировать место (адреса) невозможно, поэтому реализация вставки невозможна без реорганизации массива. Для вставки новой записи требуется сдвиг всех последующих записей (в среднем – половины элементов массива).

При удалении записей также необходим сдвиг всех последующих записей (также в среднем половины элементов массива). Аннулирование записей воз-

можно без реорганизации массива. Появляющиеся при этом свободные участки памяти могут быть отмечены как свободные в специальном массиве, тогда каждый раз при выполнении операции над данными придется проверять, не удалены ли данные.

Операция замены возможна без реорганизации массива.

Корректировка может выполняться двумя способами. В первом случае изменения вносятся в основной массив сразу. Во втором случае, применяемом при очень больших размерах массивов, записи с измененными данными накапливаются в специальном массиве изменений. При достижении определенного объема массив изменений объединяется с основным массивом.

1.6 Методы обработки списковых структур данных

Основным преимуществом списковой структуры является гибкость и высокая скорость корректировки (добавления и удаления элементов). Однако поиск в них выполняется только последовательно. Имеются способы, позволяющие ускорить поиск в списковых структурах. К ним относятся: использование адресной функции (см. ранее), К- и А-индексов; гнездовой способ организации.

1.6.1 Метод К- и А- индексов

Значения ключей записей, номера которых образуют арифметическую прогрессию, называют К-индексами. Адреса записей, значения ключей которых образуют арифметическую прогрессию, называют А-индексами.

В списковой структуре, снабженной теми или иными индексами, поиск ведется в два этапа: в массиве индексов и в основной структуре.

При поиске элемента с ключом q в списковой структуре, имеющей массив К-индексов, находится индекс K_i , такой, что $K_i \leq q < K_{i+1}$ (i – номер индекса в массиве индексов). Далее организуется вход в список по адресу, записанному в K_i , и поиск в этой структуре методом перебора.

В случае А-индексов для искомого q вычисляется $i = q/z$, где z – интервал значения ключа для А-индекса. Далее элементы списка просматриваются последовательно, начиная с адреса, записанного в i -м А-индексе.

1.6.2 Гнездовой способ организации

Гнездовой способ заключается в том, что множество элементов разбивается на группы по определенному принципу. Затем эти группы объединяют в отдельные гнезда с одинаковой структурой. Далее строят дополнительную структуру, в которой все элементы связываются. Каждый элемент содержит ключевой признак и адрес только одного гнезда.

На рисунке 1.5 представлен пример организации гнездового способа с помощью списка гнезд. В этом примере записи базы данных делятся на группы по принципу принадлежности первой буквы фамилии. Для доступа к гнезду организован список гнезд, в котором каждый элемент имеет три поля: ключевой признак гнезда (первая буква фамилии), адрес гнезда и адрес следующего элемента списка гнезд. Для хранения самих гнезд используется структура с неявными связями – массив.

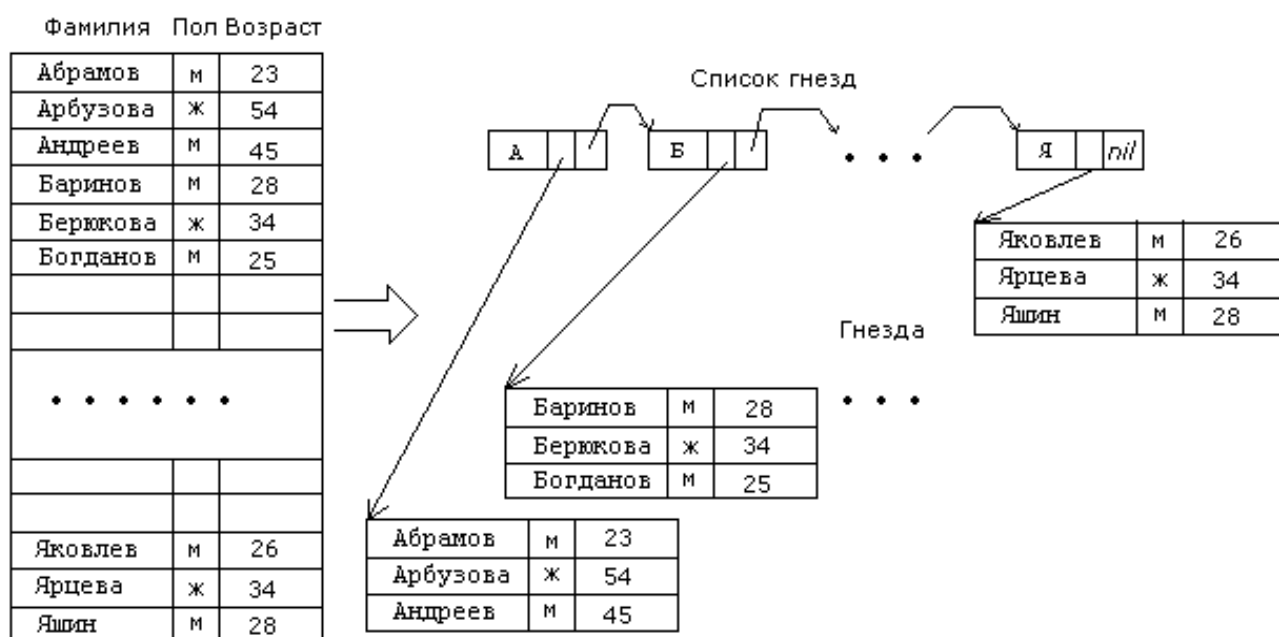


Рисунок 1.5 - Пример организации данных с помощью списка гнезд

Алгоритм А.

Первый элемент массива следует поместить в корень дерева. Значение ключа второго элемента p_2 сравниваем с ключом p_1 корня дерева. Если $p_2 < p_1$, то добавляем p_2 к левой от корня ветви. Если $p_2 \geq p_1$, то p_2 добавляем к правой от корня ветви. Для добавления следующих элементов осуществляют поиск первого свободного места, продвигаясь по дереву следующим образом: если $p_i < p_k$, то необходимо перейти к вершине, находящейся на левой ветви от p_k , иначе необходимо перейти к вершине, находящейся на правой ветви от p_k . Если такой вершины нет, то она достраивается и в нее помещается i -й элемент массива.

Алгоритм Б.

Исходные записи должны иметь вид упорядоченного массива. За корень дерева принимается средняя запись массива, которая разделяет массив на примерно равные части. Средние записи в обеих частях массива образуют вершины второго уровня, а массив оказывается разделенным на 4 части. Средние части каждой из четырех частей помещаются на третьем уровне бинарного дерева и т. д.

Включение новой записи в бинарное дерево происходит так же, как и при формировании дерева по алгоритму А.

Исключение концевой вершины сводится к удалению ее адреса из предшествующей вершины. При исключении вершины, имеющей одно поддереву, бинарное дерево распадается на части. Для восстановления связности дерева необходимо корень поддерева присоединить к основной части дерева, используя алгоритм А. При исключении вершины с двумя поддеревьями бинарное дерево распадается на три части. Корни поддеревьев в любом порядке присоединяются к основной части также согласно алгоритму А.

Древовидные структуры данных часто используются для реализации структур данных, элементы которых необходимо искать по составному ключу, включающему несколько признаков. В этом случае каждый признак соответствует вершине дерева: первый – вершине 1-го уровня; второй – вершине 2-го уровня и

т. д. Если у двух составных ключей несколько начальных признаков совпадают, то они имеют общие вершины в дереве.

1.8 Выбор способа реализации абстрактных структур данных

Способ реализации абстрактных структур данных зависит от: частоты изменения данных, их структуры и ограничений на время доступа. Реализация включает физическую структуру данных, а также процедуры и функции, выполняющие операции над ними.

Физическая структура данных помимо информационных полей может содержать служебные поля, в которых хранится информация, необходимая для работы со структурой. Например, физическая структура односвязного списка представляет собой совокупность дескриптора и одинаковых по формату и размеру записей, размещенных произвольно в некоторой области памяти и связанных друг с другом в линейно упорядоченную цепочку с помощью указателей. Дескриптор может содержать информацию: код структуры, имя списка, адрес начала, текущее число элементов и т. п. В линейном двусвязном списке имеется два указателя: прямой (указывает на следующий элемент) и обратный (указывает на предыдущий элемент).

Односвязный список всегда линейный. Двусвязный может быть нелинейным, если второй указатель каждого элемента в списке задает порядок произвольного типа. На рисунке 1.7 представлен двусвязный список, в котором первый указатель осуществляет связь между элементами какой-либо группы, а второй осуществляет связь внутри подгруппы. S – указатель начала всего списка.

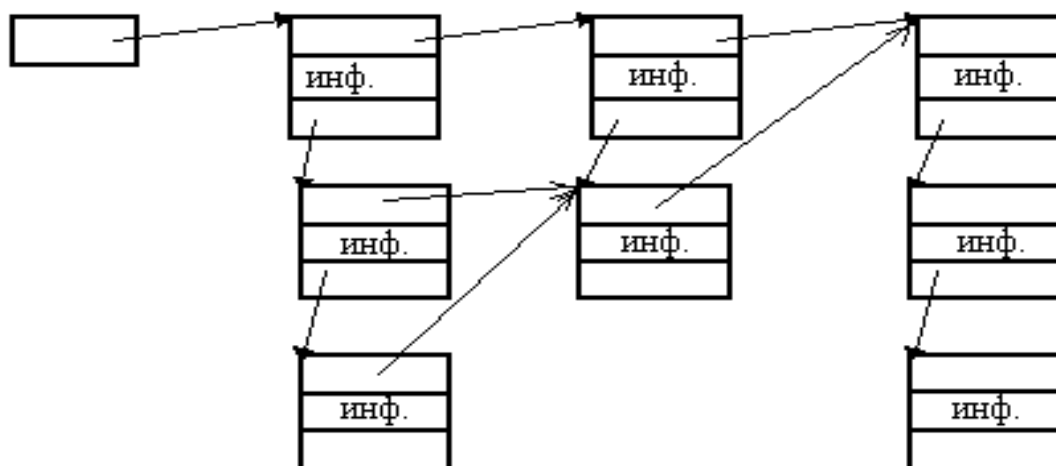


Рисунок 1.7 - Пример двусвязного нелинейного списка

Для конкретной задачи обычно можно предложить несколько реализаций выбранной структуры, которые отличаются временами выполнения отдельных операций (например, см. рисунок 1.8 – девять способов реализации графа).

Сравнение вариантов выполняется по требуемым объемам памяти (*емкостной сложности*) и времени выполнения операций (*временной сложности*).

Емкостная сложность оценивается по физической структуре с учетом используемых типов данных.

Для оценки временной сложности можно использовать следующую методику: 1) разрабатывается фрагмент алгоритма и по нему примерно определяется последовательность выполнения команд, реализующих ту или иную операцию над данными; 2) полученный результат преобразуется по таблице соответствий (см. таблицу 1.1) к времени выполнения фрагмента алгоритма в машинных тактах.

Таблица 1.1 - Коэффициенты приведения команд Borland Pascal (Pentium)

Операция	Обозначение	Время, тактов	Операции	Обозначение	Время, тактов
$a:=b$	$t_{=}$	2	a/b	$t_{/}$	28
$c:=a\pm b, a>b$	t_{+}	2	$a:=b^{\wedge}.num$	$t_{=\wedge}$	2
$A:=a\pm b$	$t_{=+}$	2^{*}	$b:=b^{\wedge}.p$	$t_{\wedge}$	2^{*}
$a:=a\pm const$	t_{++}	1	сдвиги	$t \rightarrow$	1
$A\pm const$	t_{+c}	1	$a[i]$	t_i	2
$a*b$	t_{*}	20	$a[i,j]$	$t_{i,j}$	26
If	t_{if}	$t_{пр}+1+p1\times t_{1+}+p2\times t2^{**}$	for	t_{for}	$t_{уст}+t_{пр}+1+ +n(t_{тела}+t_{пр}+2)^{***}$

* – операции накопления/переадресации (при использовании в цикле коэффициент равен 1);

** - $t_{пр}$ – время проверки условия, $p1, p2$ – вероятности выбора соответствующих ветвей, $t1, t2$ – времена выполнения ветвей;

*** - $t_{уст}$ – время установки начального значения индекса, $t_{пр}$ – время проверки условия, $t_{тела}$ – время выполнения тела цикла; для цикла вида

for i:=1 to n do ... получаем: $t=2+2+1+n(t_{тела}+2+2)=5+n(t_{тела}+4)$.

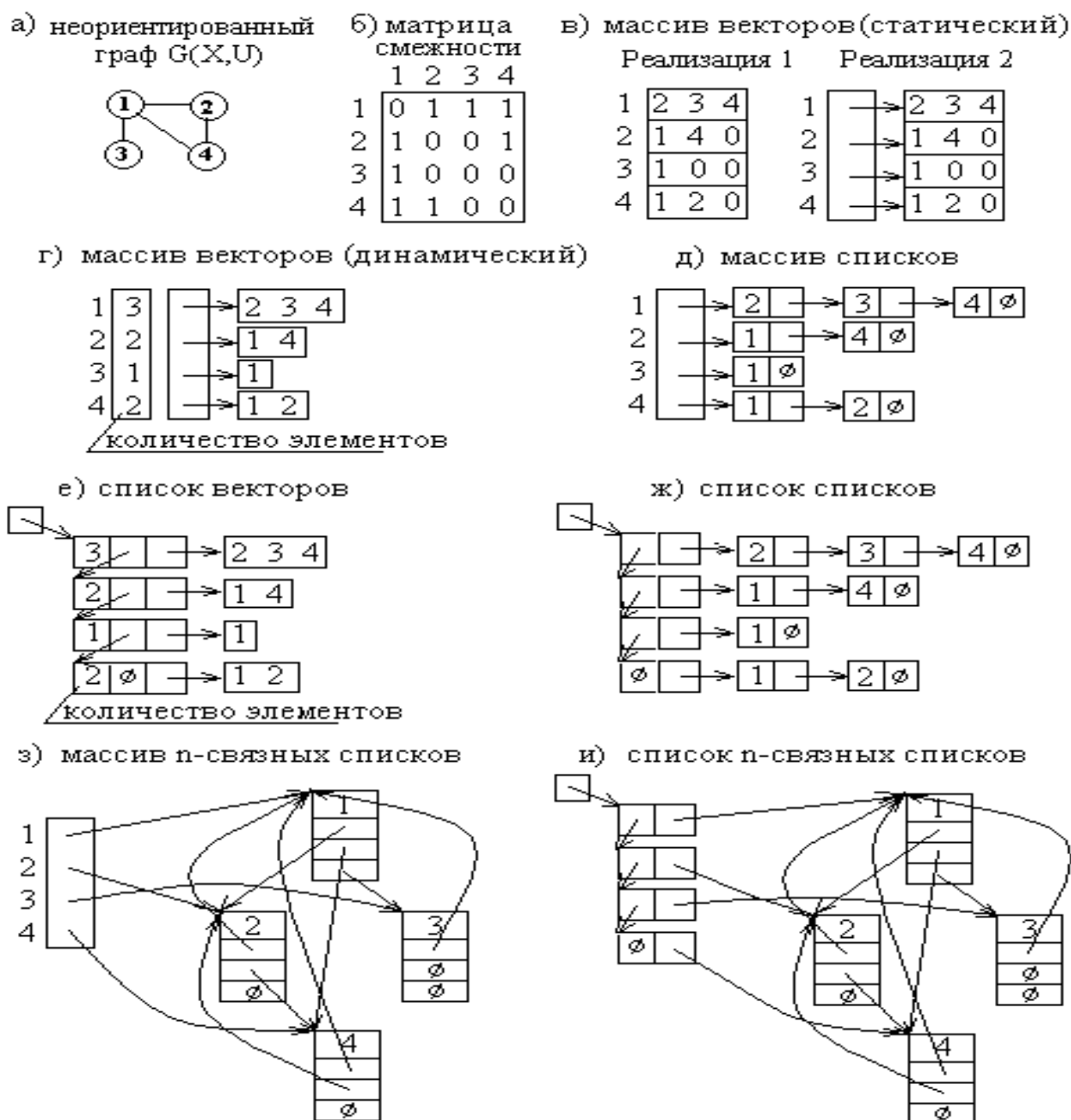


Рисунок 1.8 - Различные способы реализации графов

Пример. Выполнить оценки памяти и вычислительной сложности выполнения операций поиска и удаления элемента для двух реализаций хранения последовательности n целых чисел: в виде массива и в виде линейного односвязного списка (см. рис. 1.9).

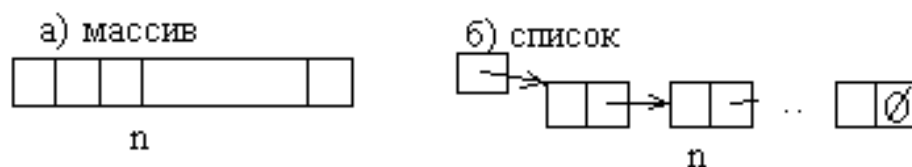


Рисунок 1.9 – Массив (а) и односвязный список (б)

Решение.

1. *Оценка памяти:*

а) для массива: $V = l_{эл} * n$, если в массиве хранятся целые числа, то $V = 2n$ (байт)

б) для списка: $V = (l_{эл} + l_{ук}) * n + l_{ук}$,

если в списке хранятся целые числа, а указатели имеют длину 4 байта, то $V = 6n + 4$ (байт)

2. *Оценка времени поиска i-го элемента данных:*

а) для массива:

поиск выполняется по индексу $A_i = A_0 + (i-1)*2 = A_0 - 2 + i*2 = const + i*2$,

операция умножения на 2 выполняется посредством сдвига на 1 разряд влево,

откуда $t = t_{++} + t_{\rightarrow} = 2$ (такта)

б) для списка:

доступ к элементу выполняется последовательно

$f := first; \text{ for } j := 1 \text{ to } i \text{ do } f := f \wedge p;$

определяем $t = 2 + 2 + 1 + (i-1)*(t_{\wedge} + 2 + 2) = 5 + 5(i-1)$,

минимальное время – первого элемента $t_{min} = 5$,

максимальное время поиска – последнего элемента $t_{max} = 5 + (n-1)(t_{\wedge} + 4) = 5 + 5n - 5 = 5n$,

в среднем $t = (t_{max} + t_{min})/2 = 2.5 + 2.5n$ (тактов).

3. *Оценка времени удаления i-го элемента:*

а) для массива:

при удалении i-го элемента выполняется сдвиг остальной части массива:

$\text{for } j := i \text{ to } n-1 \text{ do } a[j] := a[j+1];$

откуда $t = t_{+c} + 2 + 2 + (n-i)(t_{+c} + t_i + t_i + t_{=}) = 5 + 7(n-i)$,

минимальное время работы цикла (при $i = n-1$) $t_{min} = 12$,

максимальное время работы цикла (при $i = 1$) $t_{max} = 5 + 7(n-1) = -2 + 7n$,

соответственно среднее время работы цикла $t = 5 + 3.5n$ (тактов).

При $i=n$ сдвиг не выполняется, а только значение n уменьшается на единицу ($t_{++}=1$). Тогда среднее время удаления равно: $t = t_{пр} + (t_{++} + t_{max})/2 = 2 + (1 - 2 + 7n)/2 = 1,5 + 3,5n$ (тактов)

б) для списка:

при удалении элемента достаточно перезаписать адресное поле: $f^{\wedge}.p := r^{\wedge}.p$ (без учета времени освобождения памяти и времени поиска элемента), следовательно, $t = t_{\wedge} + t_{=\wedge} = 3$ (такта).

При выборе структуры учитывают удельный вес операций анализируемых операций.

1.9 Вопросы для самопроверки

- 1.Применим метод последовательного поиска к таблицам?
- 2.Перечислите достоинства и недостатки гнездового метода поиска?
- 3.Перечислите достоинства и недостатки двухсвязного списка?
- 4.Какие количественные оценки применяют к методам сортировки?
- 5.Какие недостатки у дихотомического метода?
6. Какие недостатки у метода шелла?
- 7.Какой метод сортировки лучше?
8. Какие недостатки у метода удаления сдвигом?
- 9.Как построить бинарное дерево?
- 10.Какое бинарное дерево лучше?
- 11.Когда используют структуры данных с явными связями?
- 12.Зачем нужна кэш-таблица?
13. Какие недостатки у метода упорядочения обменом?
14. Какие недостатки у метода вычисления адреса?
15. Какие ограничения на количество гнезд в гнездовом методе?
16. В каком дереве удаление элемента выполняется быстрее?

Глава 2. Оценка эффективности и качества программы

Освоение способов повышения эффективности и качества программных продуктов является важной задачей. Только эффективные, технологичные и качественные программы могут эксплуатироваться в течение долгого времени. На данный момент накоплен некоторый опыт, который может быть использован разработчиками, но всегда приходится учитывать специфику конкретной задачи, следовательно, создать эффективную и качественную программу непросто.

2.1 Основные понятия

Ниже приведены основные понятия и характеристики имеющие важное значения при оценке качества программных продуктов.

Технологичность определяет качество проекта и, как правило, имеет важное значение при разработке сложных программных систем. От технологичности зависят трудовые и материальные затраты на реализацию и последующую модификацию программного продукта. Факторами, которые определяют технологичность являются: *проработанность модели; уровень независимости модулей; стиль программирования и степень повторного использования кодов*. В данной лабораторной работе из-за ограниченного объема времени оценка этого качества не выполняется.

Эффективность программы можно оценить отдельно по каждому ресурсу вычислительной машины. Критериями оценки являются:

- *Время ответа или выполнения*. Оценивается для программ, работающих в интерактивном режиме или в режиме реального времени.
- *Объем оперативной памяти*. Является важным для вычислительных систем, где оперативная память является дефицитным ресурсом.
- *Объем внешней памяти*. Оценивается, если внешняя память дефицитный ресурс или при интенсивной работе с данными.
- *Количество обслуживаемых терминалов и др.*

Выбор критерия оценки эффективности программы сильно зависит от выполняемых ею функций. Например, если основной функцией является поиск данных, то уменьшить время выполнения можно за счет избыточного объема оперативной памяти. Другими словами, оценка памяти будет уже второстепенной задачей.

Ниже описаны способы повышения эффективности программ.

2.2 Способы уменьшения времени выполнения

Время выполнения программы в первую очередь зависит от используемых в ней методов. Однако, важную роль играют циклические фрагменты с большим количеством повторений. Поэтому по возможности необходимо минимизировать тело цикла.

При написании циклов рекомендуется:

- выносить из тела цикла выражения, не зависящие от параметров цикла;
- заменять «длинные» операции умножения и деления вычитанием и сдвигами;
- уменьшить количество преобразования типов в выражениях;
- исключать лишние проверки;
- исключать многократные обращения к элементам массивов по индексам (особенно многомерных, так как при вычислении адреса элемента используются операции умножения на значение индексов).

Пример . Пусть имеется цикл следующей структуры (Pascal):

```
for y:=0 to 99
  for x:=0 to 99 do
    begin
      a[320*x+y]:=S[k,l];
    end; ...
```

В этом цикле операции умножения и обращения к элементу $S[k,l]$ выполняются 10000 раз. Оптимизируем цикл:

```

skl:=S[k,l];      {выносим обращение к элементу массива из цикла}
for x:=0 to 99 do {меняем циклы местами}
  begin
    i:=x shl 8+x shl 6; {умножение заменяем на сдвиги и выносим}
    for y:=0 to 99 do
      a[i+y]:=skl;
    end; ...
  
```

(Сдвиг: Пусть $x=11$ в двоичной системе, тогда 11 сдвигаем на 8 -- 1100000000 и 11 сдвигаем на 6 --11000000 получаем 1111000000, т.е. $512+256+128+64+0+0+0+0+0+0=960$)

В результате вместо 10000 операций умножения будут выполняться 200 операций сдвига, а их время приблизительно сравнимо со временем выполнения операции сложения. Обращение к элементу массива $S[k,l]$ будет выполнено один раз.

2.3 Способы экономии памяти.

Ниже приведен перечень рекомендаций, которые применяются для экономии памяти.

- Следует выбирать алгоритмы обработки, *не требующие дублирования исходных данных структурных типов*. Например, метод «вставки» - не требует дополнительных массивов.
- По возможности использовать динамическую память. При необходимости выделять память, а потом освобождать.

• При передаче структурных данных в подпрограмму по значению, копии этих данных размещаются в стеке. Избежать копирования можно, если передавать данные не по значению, а как неизменяемые (описанные `const`). В последнем случае в стеке размещается только адрес данных, например:

```
Type Massiv=array[1..100] of real;  
function Summa(Const a:Massiv;
```

2.4 Критерии качества программных продуктов.

Качество программных продуктов может оцениваться следующими критериями:

- правильность – функционирование в соответствии с заданием;
- универсальность – обеспечение правильной работы при любых допустимых данных и содержание средств защиты от неправильных данных;
- надежность (помехозащищенность) – обеспечение полной повторяемости результатов, т.е. обеспечение их правильности при наличии различного рода сбоев;
- проверяемость – возможность проверки получаемых результатов;
- точность результатов – обеспечение погрешности результатов не выше заданной;
- защищенность – сохранение конфиденциальности информации;
- эффективность – использование минимально возможного объема ресурсов технических средств;
- адаптируемость – возможность быстрой модификации с целью приспособления к изменяющимся условиям функционирования;
- повторная входимость – возможность повторного выполнения без перезагрузки с диска (для резидентных программ);
- реентерабельность – возможность «параллельного» использования несколькими процессами.

Ниже рассмотрим пример программы, в которой попытаемся сделать улучшения.

Задача. Найти корни функции методом "хорд" с точностью 0.000001

для $f(x) = x^3 / 100 - 5$.

```

Program hrd; {исходный текст программы}
  var x,p,z,b,eps:real;
  function f(x:real):real; {исследуемая функция}
    begin f:=x*x*x/100-5 end;
  Begin
x:=-10; b:=10; eps:=0.000001; {исследуемый интервал и погрешность}
  Repeat {основной цикл}
    p:=f(x); {сохранение предыдущего значения}
    x:=x-f(x)*(b-x)/(f(b)-f(x)); {определение корня}
  Until abs(f(x)-p)<=eps;
writeln(' x = ',x:10:5,' y = ',f(x):10:5); {вывод результатов}
End.

```

В результате анализа данной программы получаем следующие выводы:

1. В теле цикла многократно (5 раз) повторяются одни и те же действия, а именно, вычисление значения функции. Если сократить количество вычислений значения функции, то можно уменьшить время выполнения программы, а следовательно повысить эффективность.

2. Не корректно задан формат вывода вычисленного значения, т.к. он не учитывает порядок погрешности.

Экспериментально подтвердить сделанные выводы можно следующим образом:

1. Ввести в программу контрольные точки, например, фиксировать время начала выполнения цикла и время окончания с помощью процедуры Gettime, а затем определить время выполнения.

2. Чтобы время выполнения было легче зарегистрировать или увидеть, можно воспользоваться, образно говоря, «увеличительной линзой». Другими словами, можно ввести дополнительный цикл, телом которого является исследуемый фрагмент. В результате с помощью процедуры Gettime можно определить полученный эффект.

*Примечание. При внесении в программу контрольных точек, необходимо учитывать их влияние.

В рассматриваемом примере тело цикла можно модифицировать так:

```
Repeat  p:=z; x:=x-p*(b-x)/(f(b)-p); z:=f(x); Until abs(z-p)<=eps;
```

Ниже приведены две программы (программа на языке Turbo Pascal и программа консольного приложения Delphi) , в которых отражены контрольные точки и исследуемые фрагменты.

```
Program hrd; { программа на языке Turbo Pascal}
```

```
Uses dos,crt;
```

```
var x,p,z,b,eps:real; {z - дополнительная переменная}
```

```
i,t1,t2,t3,t4,t11,t22,t33,t44:word; {вспомогательные переменные}
```

```
function f(x:real):real;
```

```
begin f:=x*x*x/100-5 end;
```

```
Begin
```

```
clrscr;
```

```
Gettime(t1,t2,t3,t4); {фиксация времени начала}
```

```
x:=-10; b:=10; eps:=0.000001; z:=f(x);
```

```
{for i:=1 to 50000 do
```

```
{ Исходный цикл}
```

```
Repeat  p:=f(x); x:=x-f(x)*(b-x)/(f(b)-f(x)); Until abs(f(x)-p)<=eps;
```

```
}
```

```
for i:=1 to 50000 do
```

```

{Модифицированный цикл}
Repeat  p:=z;  x:=x-p*(b-x)/(f(b)-p);  z:=f(x);  Until abs(z-p)<=eps;

Gettime(t11,t22,t33,t44); {фиксация времени окончания}
writeln(' x = ',x:10:5); {результаты }
writeln(t1,' ',t2,' ',t3,' ',t4); {время начала выполнения}
writeln(t11,' ',t22,' ',t33,' ',t44); { время окончания}
End.

```

В результате получаем, что время выполнения исходного цикла на языке Turbo Pascal, например, на Pentium 120 при $i_{\max}=50000$ равно 77 ms, а время выполнения модифицированного цикла 44 ms, т.е. повысили эффективность в 1.75 раза.

```

Program hrd; {программа консольного приложения Delphi}
{$APPTYPE CONSOLE}

uses

  SysUtils;

var x,p,z,b,eps:real; // z - дополнительная переменная
    P1,P2: TDateTime; // вспомогательные переменные
    H1,M1,S1,Ms1,H2,M2,S2,Ms2:word;
    i:longint;

function f(x:real):real;
begin f:=x*x*x/100-5 end;

Begin
P1:= Now; // фиксация времени начала
DecodeTime(P1, H1, M1, S1, Ms1);

x:=-10; b:=10; eps:=0.000001; z:=f(x);

{ for i:=1 to 50000000 do

```

```

Repeat // Исходный цикл
  p:=f(x); x:=x-f(x)*(b-x)/(f(b)-f(x));
Until abs(f(x)-p)<=eps;
}

for i:=1 to 50000000 do
Repeat // Модифицированный цикл
  p:=z; x:=x-p*(b-x)/(f(b)-p); z:=f(x);
Until abs(z-p)<=eps;

P2:= Now; // фиксация времени окончания
DecodeTime(P2, H2, M2, S2, Ms2);

writeln(' x = ',x:10:5); // результаты
Writeln(H1,':',M1,':',S1,':',Ms1); // время начала выполнения
Writeln(H2,':',M2,':',S2,':',Ms2); // время окончания
readln;
End.

```

В результате получаем, что время выполнения исходного цикла программы консольного приложения Delphi, например, на Intel(R) Core™ i3-2100 при $i_{\max}=50000000$ равно 6031 ms, а время выполнения модифицированного цикла 2938 ms, т.е. повысили эффективность в 2 раза.

2.5 Вопросы для самопроверки

1. Что такое технологичность?
2. Что значит эффективная программа?
3. Перечислите критерии оценки эффективности?
4. Перечислите способы уменьшения времени выполнения?
5. Как замерить время выполнения программы?
6. Какие факторы искажают результаты замеров?
7. Какой способ наиболее существенно может повысить эффективность?
8. Что значит проверить на правильность?
9. Что значит проверить на универсальность?
10. Что значит проверить на точность результатов?
11. Как оценить проверяемость?
12. Какие имеются способы экономии памяти?
13. Что такое контрольные точки?
14. Как влияет конфигурация оборудования на время выполнения?

Глава 3. Тестирование программного обеспечения

Одним из наиболее трудоемких этапов (от 30 до 60% общей трудоемкости) создания программного продукта является тестирование. Причем доля стоимости тестирования в общей стоимости разработки имеет тенденцию возрастать при увеличении сложности комплексов программ и повышении требований к их качеству. В связи с этим большое внимание уделяется выбору стратегии и методов тестирования, что не является тривиальной задачей.

Таким образом, при подготовке к тестированию необходимо ответить на следующие вопросы:

- Какую стратегию тестирования выбрать и почему?
- Как реализовать стратегию тестирования?
- Какой из методов стратегии тестирования выбрать и почему?
- Как грамотно подготовить тестовый набор данных и сколько тестов необходимо разработать?
- Как выполнить эффективное тестирование?

3.1 Основные принципы тестирования.

Тестированием называется процесс выполнения программы **с целью обнаружения ошибки(!)**. Никакое тестирование не может доказать отсутствие ошибок в программе.

Исходными данными для тестирования являются техническое задание, спецификации, а для некоторых методов тестирования - алгоритм тестирования.

При тестировании рекомендуется соблюдать следующие *основные принципы*:

1. Предполагаемые результаты должны быть известны до тестирования.
2. Следует избегать тестирования программы автором.
3. Необходимо досконально изучать результаты каждого теста.
4. Необходимо проверять действия программы на неверных данных.

5. Необходимо проверять программу на неожиданные побочные эффекты.
6. Удачным считается тест, который обнаруживает хотя бы одну еще не обнаруженную ошибку.
7. Вероятность наличия ошибки в части программы пропорциональна количеству ошибок, уже обнаруженных в этой части.

3.2 Ручное тестирование программных продуктов

Эксперименты показали, что с точки зрения нахождения ошибок, достаточно эффективными являются методы ручного контроля. Поэтому один или несколько из них должны использоваться в каждом программном проекте. Методы ручного контроля предназначены для периода разработки, когда программа закодирована, но тестирование на машине еще не началось. Доказано, что эти методы способствуют существенному увеличению производительности и повышению надежности программ и с их помощью можно находить от 30 до 70% ошибок логического проектирования и кодирования.

Основными методами ручного тестирования являются:

- ◆ инспекции исходного текста;
- ◆ сквозные просмотры;
- ◆ просмотры за столом;
- ◆ обзоры программ.

3.2.1 Инспекции исходного текста.

Инспекции исходного текста (структурный контроль) представляют собой набор процедур и приемов обнаружения ошибок при изучении текста группой специалистов, в которую входят автор программы, проектировщик, специалист по тестированию и координатор (компетентный программист, но не автор программы). Общая процедура инспекции состоит из следующих этапов:

1. участникам группы заранее выдается листинг программы и спецификация на нее;

2. программист рассказывает о логике работы программы и отвечает на вопросы инспекторов;

3. программа анализируется по списку вопросов для выявления исторически сложившихся общих ошибок программирования.

Кроме нахождения ошибок, результаты инспекции позволяют программисту увидеть сделанные им ошибки, получить возможность оценить свой стиль программирования, выбор алгоритмов и методов тестирования. Инспекция является способом раннего выявления частей программы, с большей вероятностью содержащих ошибки, что позволяет при тестировании уделить внимание именно этим частям.

3.2.2 Сквозные просмотры

Сквозной просмотр, как и инспекция, представляет собой набор способов обнаружения ошибок, осуществляемых группой лиц, просматривающих текст программы. Такой просмотр имеет много общего с процессом инспектирования, но отличается процедурой и методами обнаружения ошибок. Группа по выполнению сквозного контроля состоит из 3-5 человек (председатель или координатор, секретарь, фиксирующий все ошибки, специалист по тестированию, программист и независимый эксперт). Этапы процедуры сквозного контроля:

- 1) участникам группы заранее выдается листинг программы и спецификация на нее;
- 2) участникам заседания предлагается несколько тестов, написанных на бумаге, и тестовые данные подвергаются обработке в соответствии с логикой программы (каждый тест мысленно выполняется);
- 3) программисту задаются вопросы о логике проектирования и принятых допущениях;
- 4) состояние программы (значения переменных) отслеживается на бумаге или доске.

В большинстве сквозных просмотров при выполнении самих тестов находят меньше ошибок, чем при опросе программиста.

3.2.3 Проверка за столом

Третьим методом ручного обнаружения ошибок является применявшаяся ранее других методов «проверка за столом». Это проверка исходного текста или сквозные просмотры, выполняемые одним человеком, который читает текст программы, проверяет его по списку и пропускает через программу тестовые данные. Исходя из принципов тестирования, проверку за столом должна проводиться человеком, не являющимся автором программы.

Недостатки метода:

- ◇ проверка представляет собой полностью неупорядоченный процесс;
- ◇ отсутствие обмена мнениями и здоровой конкуренции;
- ◇ меньшая эффективность по сравнению с другими методами.

3.2.4 Оценка посредством просмотра

Этот метод непосредственно не связан с тестированием. Он является методом оценки анонимной программы в терминах ее общего качества, простоты эксплуатации и ясности. Цель этого метода - обеспечить сравнительно объективную оценку и самооценку программистов. Выбирается программист, который должен выполнять обязанности администратора процесса. Администратор набирает группу от 6 до 20 участников, которые должны быть одного профиля. Каждому участнику предлагается представить для рассмотрения две программы с его точки зрения наилучшую и наихудшую. Отобранные программы случайным образом распределяются между участниками. Им дается по 4 программы - две наилучшие и две наихудшие, но программист не знает, какая из них плохая, а какая хорошая. Программист просматривает их и заполняет анкету, в которой предлагается оценить их относительное качество по шкале из семи баллов. Кроме того, проверяющий дает общий комментарий и рекомендации по улучшению программы.

3.2.5 Перечень вопросов для структурного контроля текста

1. *Обращения к данным.*

1) Все ли переменные инициализированы?

- 2) Не превышены ли максимальные (или реальные) размеры массивов и строк?
- 3) Не перепутаны ли строки со столбцами при работе с матрицами?
- 4) Присутствуют ли переменные со сходными именами?
- 5) Используются ли файлы? Если да, то
 - При вводе из файла проверяется ли завершение файла?
 - Соответствуют ли типы записываемых и читаемых значений?
- 6) Использованы ли нетипизированные переменные, открытые массивы, динамическая память? Если да, то
 - Соответствуют ли типы переменных при "наложении" формата?
 - Не выходят ли индексы за границы массивов?

2. *Вычисления.*

- 1) Правильно ли записаны выражения (порядок следования операторов)?
- 2) Корректно ли производятся вычисления неарифметических переменных?
- 3) Корректно ли выполнены вычисления с переменными различных типов (в том числе с использованием целочисленной арифметики)?
- 4) Возможно ли переполнение разрядной сетки или ситуация машинного нуля?
- 5) Соответствуют ли вычисления заданным требованиям точности?
- 6) Присутствуют ли сравнения переменных различных типов?

3. *Передачи управления.*

- 1) Будут ли корректно завершены циклы?
- 2) Будет ли завершена программа?
- 3) Существуют ли циклы, которые не будут выполняться из-за нарушения условия входа? Корректно ли продолжатся вычисления?
- 4) Существуют ли поисковые циклы? Корректно ли отрабатываются ситуации "элемент найден" и "элемент не найден"?

4. *Интерфейс.*

- 1) Соответствуют ли списки параметров и аргументов по порядку, типу, единицам измерения?
- 2) Не изменяет ли подпрограмма аргументов, которые не должны изменяться?
- 3) Не происходит ли нарушения области действия глобальных и локальных переменных с одинаковыми именами?

3.3 Тестирование по принципу «белого ящика»

Стратегия тестирования по принципу «белого ящика», или стратегия тестирования, управляемая логикой программы (с учетом алгоритма), позволяет проверить внутреннюю структуру программы. В этом случае тестирующий получает тестовые данные путем анализа логики программы.

Исчерпывающему входному тестированию стратегии "черного ящика" здесь можно сопоставить исчерпывающее тестирование маршрутов (*критерий покрытия маршрутов*). Подразумевается, что программа проверена полностью, если с помощью тестов удастся осуществить выполнение программы по всем возможным маршрутам передач управления.

Однако, нетрудно видеть, что даже в программе среднего уровня сложности число неповторяющихся маршрутов астрономическое и, следовательно, исчерпывающее тестирование маршрутов невозможно.

Кроме того, метод исчерпывающего тестирования маршрутов имеет ряд недостатков:

- ◆ метод не обнаруживает пропущенные маршруты;
- ◆ не обнаруживает ошибок, появление которых зависит от обрабатываемых данных (например, $\text{if } (a-b) < \text{eps}$ - пропуск функции abs проявится только, если $a < b$);
- ◆ не дает гарантии, что программа соответствует описанию (например, если вместо сортировки по убыванию написана сортировка по возрастанию);

Стратегия «белого ящика» включает в себя следующие методы тестирования:

- Покрытие операторов/
- Покрытие решений.
- Покрытие условий.
- Покрытие решений/условий.
- Комбинаторное покрытие условий.

3.3.1 Покрытие операторов

Критерий покрытия операторов подразумевает выполнение каждого оператора программы, по крайней мере, один раз. Это необходимое, но недостаточное условие для приемлемого тестирования. Рассмотрим пример (рис.3.1):

```

Procedure m(a,b:real;var x:real);
begin
    if(a>1)and(b=0) then x:=x/a;
    if (a=2)and(x>1) then x:=x+1;
end;
  
```

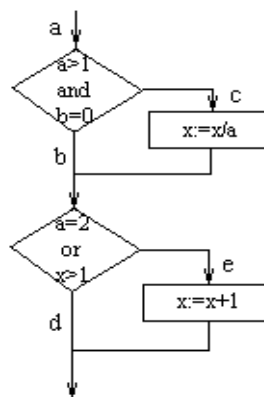


Рисунок 3.1 - Процедура и соответствующий ей алгоритм.

Для приведенного фрагмента можно было бы выполнить каждый оператор один раз, задав в качестве входных данных $a=2$, $b=0$, $x=3$. Но при этом, из второго условия следует, что x может принимать любое значение и оно не проверяется. Кроме того:

- если при написании программы в первом условии описать $(a>1) \text{ or } (b=0)$, то ошибка обнаружена не будет;
- если во втором условии вместо $x>1$ записано $x>0$, то эта ошибка тоже не будет обнаружена;
- существует путь abd (см. Рис.1), в котором x вообще не меняется и, если здесь есть ошибка, она не будет обнаружена.

3.3.2 Покрытие решений (переходов).

Для реализации этого критерия необходимо достаточное число тестов, такое, что каждое решение на этих тестах принимает значение «истина» или «ложь», по крайней мере, один раз.

Нетрудно доказать, что критерий покрытия решений удовлетворяет критерию покрытие операторов, но является более сильным.

Программа, представленная на рис.1, может быть протестирована по методу покрытия решений двумя тестами, покрывающими либо пути ace abd, либо пути acd abe.

Входные условия для первого теста: $a=3, b=0, x=3$.

Входные условия для второго теста: $a=2, b=1, x=1$.

Однако путь, где x не меняется, будет проверен с вероятностью 50%: если во втором условии вместо условия $x>1$ записано $x<1$, то ошибка не будет обнаружена двумя тестами.

3.3.3 Покрытие условий

Критерий покрытия условий более сильный по сравнению с предыдущим. В этом случае записывается число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении были выполнены, по крайней мере, один раз.

Однако, как и в случае покрытия решений, это покрытие не всегда приводит к выполнению каждого оператора, по крайней мере, один раз. К критерию требуется дополнение, заключающееся в том, что каждой точке входа управление должно быть передано, по крайней мере, один раз.

Программа на рис.1 имеет четыре условия: $a>1, b=0, f=2, x>1$. Необходимо реализовать ситуации, где $a>1, a\leq 1, b=0, b\neq 0, a=2, a\neq 2, x>1, x\leq 1$.

Тесты, удовлетворяющие этому условию:

1. $a=2, b=0, x=4$ (путь ace);
2. $a=1, b=1, x=1$ (путь abd).

Хотя количество тестов такое же, как в случае покрытия решений, этот критерий может (но не всегда) вызвать выполнение решений в условиях, не реализуемых при покрытии решений. То есть, этот критерий в основном удовлетворяет критерию покрытия решений, но не всегда. Тесты критерия покрытия условий для ранее рассмотренных примеров покрывают результаты всех решений, но

это случайное совпадение. Например, тесты $a=1, b=0, x=3$ и $a=2, b=1, x=1$ покрывают результаты всех условий, но только два из четырех результатов решений (не выполняется результат «истина» первого решения и результат «ложь» второго).

3.3.4 Покрывание решений/условий

Этот метод требует составить тесты так, чтобы все возможные результаты каждого условия выполнились, по крайней мере, один раз, все результаты каждого решения выполнились, по крайней мере, один раз и каждой точке входа управление передается, по крайней мере, один раз.

Недостатки метода:

- ◇ не всегда можно проверить все условия;
- ◇ невозможно проверить условия, которые скрыты другими условиями;
- ◇ недостаточная чувствительность к ошибкам в логических выражениях.

3.3.5 Комбинаторное покрытие условий

Этот критерий требует создания такого числа тестов, чтобы все возможные комбинации результатов условий в каждом решении и все точки входа выполнялись, по крайней мере, один раз.

Для примера на рис.1 необходимо покрыть тестами восемь комбинаций:

- | | |
|-------------------------|-------------------------|
| 1. $a>1, b=0$; | 5. $a=2, x>1$; |
| 2. $a>1, b\neq 0$; | 6. $a=2, x\leq 1$ |
| 3. $a\leq 1, b=0$; | 7. $a\neq 2, x>1$; |
| 4. $a\leq 1, b\neq 0$; | 8. $a\neq 2, x\leq 1$. |

Эти комбинации можно проверить четырьмя тестами:

- | | |
|--------------------|---------------------------|
| 1. $a=2, b=0, x=4$ | проверяет комбинации 1,5; |
| 2. $a=2, b=1, x=1$ | проверяет комбинации 2,6; |
| 3. $a=1, b=0, x=2$ | проверяет комбинации 3,7; |
| 4. $a=1, b=1, x=1$ | проверяет комбинации 4,8; |

В данном случае то, что четырем тестам соответствует четыре пути, является совпадением. Представленные тесты не покрывают всех путей, например, `acd`. Поэтому иногда необходима реализация восьми тестов.

Таким образом, для программ, содержащих только одно условие на каждое решение, минимальным является критерий, набор тестов которого:

- вызывает выполнение всех результатов каждого решения, по крайней мере, один раз;
- передает управление каждой точке входа, по крайней мере, один раз (чтобы обеспечить выполнение каждого оператора, по крайней мере, один раз).

Для программ, содержащих решения, каждое из которых имеет более одного условия, минимальный критерий состоит из набора тестов, вызывающих выполнение всех возможных комбинаций результатов условий в каждом решении и передающих управление каждой точке входа, по крайней мере, один раз. Термин «возможных» употреблен здесь потому, что некоторые комбинации условий могут быть нереализуемы. Например, для комбинации $k < 0$ и $k > 40$ задать k невозможно.

3.4 Тестирование по принципу «черного ящика»

Одним из способов проверки программ является стратегия тестирования, называемая стратегией "черного ящика" или тестированием с управлением по данным. В этом случае программа рассматривается как "черный ящик" и такое тестирование имеет целью выяснения обстоятельств, в которых поведение программы не соответствует спецификации.

Для обнаружения всех ошибок в программе необходимо выполнить *исчерпывающее тестирование*, т.е. тестирование на всех возможных наборах данных. Для тех же программ, где исполнение команды зависит от предшествующих ей событий, необходимо проверить и все возможные последовательности.

Очевидно, что построение исчерпывающего входного теста для большинства случаев невозможно. Поэтому, обычно выполняется "*разумное*" тестирование, при котором тестирование программы ограничивается прогонами на не-

большом подмножестве всех возможных входных данных. Естественно при этом целесообразно выбрать наиболее подходящее подмножество (подмножество с наивысшей вероятностью обнаружения ошибок).

Правильно выбранный тест подмножества должен обладать следующими свойствами:

1) уменьшать, причем более чем на единицу число других тестов, которые должны быть разработаны для достижения заранее определенной цели «приемлемого» тестирования:

2) покрывать значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибок до и после применения этого ограниченного множества значений входных данных.

Стратегия "черного ящика" включает в себя следующие методы формирования тестовых наборов:

- ◆ эквивалентное разбиение;
- ◆ анализ граничных значений;
- ◆ анализ причинно-следственных связей;
- ◆ предположение об ошибке.

3.4.1 Эквивалентное разбиение

Основу метода составляют два положения:

1. Исходные данные программы необходимо разбить на конечное число классов эквивалентности, так чтобы можно было предположить, что каждый тест, являющийся представителем некоторого класса, эквивалентен любому другому тесту этого класса. Иными словами, если тест какого-либо класса обнаруживает ошибку, то предполагается, что все другие тесты этого класса эквивалентности тоже обнаружат эту ошибку и наоборот

2. Каждый тест должен включать по возможности максимальное количество различных входных условий, что позволяет минимизировать общее число необходимых тестов.

Первое положение используется для разработки набора "интересных" условий, которые должны быть протестированы, а второе - для разработки минимального набора тестов.

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа:

- ◆ выделение классов эквивалентности;
- ◆ построение тестов.

Выделение классов эквивалентности

Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза из спецификации) и разбиением его на две или более групп. Для этого используется таблица следующего вида:

Входное условие	Правильные классы эквивалентности	Неправильные классы эквивалентности

Правильные классы включают правильные данные, неправильные классы - неправильные данные.

Выделение классов эквивалентности является эвристическим процессом, однако при этом существует ряд правил:

- Если входные условия описывают *область* значений (например «целое данное может принимать значения от 1 до 999»), то выделяют один правильный класс $1 \leq X \leq 999$ и два неправильных $X < 1$ и $X > 999$.
- Если входное условие описывает *число* значений (например, «в автомобиле могут ехать от одного до шести человек»), то определяется один правильный класс эквивалентности и два неправильных (ни одного и более шести человек).
- Если входное условие описывает множество входных значений и есть основания полагать, что каждое значение программист трактует особо (например, «известные способы передвижения на АВТОБУСЕ, ГРУЗОВИКЕ, ТАКСИ, МОТОЦИКЛЕ или ПЕШКОМ»), то определяется правильный класс эквивалент-

ности для каждого значения и один неправильный класс (например «на ПРИЦЕПЕ»).

- Если входное условие описывает ситуацию «должно быть» (например, «первым символом идентификатора должна быть буква»), то определяется один правильный класс эквивалентности (первый символ - буква) и один неправильный (первый символ - не буква).

- Если есть любое основание считать, что различные элементы класса эквивалентности трактуются программой неодинаково, то данный класс разбивается на меньшие классы эквивалентности.

Построение тестов.

Этот шаг заключается в использовании классов эквивалентности для построения тестов. Этот процесс включает в себя:

- Назначение каждому классу эквивалентности уникального номера.
- Проектирование новых тестов, каждый из которых покрывает как можно большее число непокрытых классов эквивалентности, до тех пор, пока все правильные классы не будут покрыты (только не общими) тестами.
- Запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов эквивалентности, до тех пор, пока все неправильные классы не будут покрыты тестами.

Разработка индивидуальных тестов для неправильных классов эквивалентности обусловлено тем, что определенные проверки с ошибочными входами скрывают или заменяют другие проверки с ошибочными входами.

Недостатком метода эквивалентных разбиения в том, что он не исследует комбинации входных условий.

3.4.2 Анализ граничных значений

Граничные условия - это ситуации, возникающие на, выше или ниже границ входных классов эквивалентности. Анализ граничных значений отличается от эквивалентного разбиения следующим:

♦ Выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных условий осуществляется таким образом, чтобы проверить тестом каждую границу этого класса.

♦ При разработке тестов рассматриваются не только входные условия (*пространство входов*), но и *пространство результатов*.

Применение метода анализа граничных условий требует определенной степени творчества и специализации в рассматриваемой проблеме. Тем не менее, существует несколько общих правил этого метода:

- Построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, если входное условие описывает область значений (например, для области входных значений от -1.0 до $+1.0$ необходимо написать тесты для ситуаций -1.0 , $+1.0$, -1.001 и $+1.001$).

- Построить тесты для минимального и максимального значений условий и тесты, большие и меньшие этих двух значений, если входное условие удовлетворяет дискретному ряду значений. Например, если входной файл может содержать от 1 до 255 записей, то проверить 0, 1, 255 и 256 записей.

- Использовать правило 1 для каждого выходного условия. Причем, важно проверить границы пространства результатов, поскольку не всегда границы входных областей представляют такой же набор условий, как и границы выходных областей. Не всегда также можно получить результат вне выходной области, но, тем не менее, стоит рассмотреть эту возможность.

- Использовать правило 2 для каждого выходного условия.

- Если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества.

- Попробовать свои силы в поиске других граничных условий.

Анализ граничных условий, если он применен правильно, является одним из наиболее полезных методов проектирования тестов. Однако следует помнить, что граничные условия могут быть едва уловимы и определение их связано с

большими трудностями, что является недостатком этого метода. Второй недостаток связан с тем, что метод анализа граничных условий не позволяет проверять различные сочетания исходных данных.

3.4.3 Анализ причинно-следственных связей

Метод анализа причинно-следственных связей помогает системно выбирать тесты с высокой результативностью. Он дает полезный побочный эффект, позволяя обнаруживать неполноту и неоднозначность исходных спецификаций.

Для использования метода необходимо понимание булевой логики (логических операторов - и, или, не). Построение тестов осуществляется в несколько этапов.

1) Спецификация разбивается на «рабочие» участки, так как таблицы причинно-следственных связей становятся громоздкими при применении метода к большим спецификациям. Например, при тестировании компилятора в качестве рабочего участка можно рассматривать отдельный оператор языка.

2) В спецификации определяются множество причин и множество следствий. *Причина* есть отдельное входное условие или класс эквивалентности входных условий. *Следствие* есть выходное условие или преобразование системы. Каждому причине и следствию приписывается отдельный номер.

3) На основе анализа семантического (смыслового) содержания спецификации строится таблица истинности, в которой последовательно перебираются все возможные комбинации причин и определяются следствия каждой комбинации причин. Таблица снабжается примечаниями, задающими ограничения и описывающими комбинации причин и/или следствий, которые являются невозможными из-за синтаксических или внешних ограничений. Аналогично, при необходимости строится таблица истинности для класса эквивалентности.

Примечание. При этом можно использовать следующие приемы:

- По возможности выделять независимые группы причинно-следственных связей в отдельные таблицы.

- Истина обозначается "1". Ложь обозначается "0". Для обозначения безразличных состояний условий применять обозначение "X", которое предполагает произвольное значение условия (0 или 1).

4) Каждая строка таблицы истинности преобразуется в тест. При этом:

- по возможности следует совмещать тесты из независимых таблиц;
 - для классов эквивалентности входных условий дополнительно необходимо
- Недостаток метода - неадекватно исследует граничные условия.

3.4.4 Предположение об ошибке

Часто программист с большим опытом выискивает ошибки "без всяких методов". При этом он подсознательно использует метод "предположение об ошибке". Процедура метода предположения об ошибке в значительной степени основана на интуиции. Основная идея метода состоит в том, чтобы перечислить в некотором списке возможные ошибки или ситуации, в которых они могут появиться, а затем на основе этого списка составить тесты. Другими словами, требуется перечислить те специальные случаи, которые могут быть не учтены при проектировании.

3.4.5 Примеры тестирования

Пусть необходимо выполнить тестирование программы, определяющей точку пересечения двух прямых на плоскости. Попутно, она должна определять параллельность прямой одной из осей координат.

В основе программы лежит решение системы линейных уравнений:

$$Ax + By = C \text{ и } Dx + Ey = F.$$

1. Используя **метод эквивалентных разбиений**, получаем для всех коэффициентов один правильный класс эквивалентности (коэффициент - вещественное число) и один неправильный (коэффициент - не вещественное число). Откуда можно предложить 7 тестов:

- 1) все коэффициенты - вещественные числа;
- 2) -7) поочередно каждый из коэффициентов - не вещественное число.

2. По методу граничных условий:

можно считать, что для исходных данных граничные условия отсутствуют (коэффициенты - "любые" вещественные числа);

для результатов - получаем, что возможны варианты: единственное решение, прямые сливаются (множество решений), прямые параллельны (отсутствие решений). Следовательно, можно предложить тесты, с результатами внутри области:

- 1) результат - единственное решение ($\delta \neq 0$);
- 2) результат - множество решений ($\delta = 0$ и $\delta_x = \delta_y = 0$);
- 3) результат - отсутствие решений ($\delta = 0$, но $\delta_x \neq 0$ или $\delta_y \neq 0$);

и с результатами на границе:

- 1) $\delta = 0,01$;
- 2) $\delta = -0,01$;
- 3) $\delta = 0$, $\delta_x = 0,01$, $\delta_y = 0$;
- 4) $\delta = 0$, $\delta_y = -0,01$, $\delta_x = 0$.

3. По методу анализа причинно-следственных связей:

Определяем множество условий.

а) для определения типа прямой:

$$\left. \begin{array}{l} a=0 \\ b=0 \\ c=0 \\ d=0 \\ e=0 \\ f=0 \end{array} \right\} \begin{array}{l} \text{ - для определения типа и существования первой прямой} \\ \text{ - для определения типа и существования второй прямой} \end{array}$$

б) для определения точки пересечения:

$$\begin{array}{l} \delta = 0 \\ \delta_x = 0 \\ \delta_y = 0 \end{array}$$

Выделяем три группы причинно-следственных связей (определение типа и существования первой линии, определение типа и существования второй линии, определение точки пересечения) и строим таблицы истинности.

A=	B=0	C=	Результат
0		0	
0	0	X	прямая общего положения
0	1	0	прямая, параллельная оси ОХ
0	1	1	ось ОХ
1	0	0	прямая, параллельная оси ОУ
1	0	1	ось ОУ
1	1	X	множество точек плоскости

Такая же таблица строится для второй прямой.

Каждая строка этих таблиц преобразуется в тест. При возможности (с учетом независимости групп) берутся данные, соответствующие

$\delta =$	$\delta_x =$	$\delta_y =$	Ед. реш.	Мн.реш.	Реш. нет
0	0	0			
0	X	X	1	0	0
1	0	X	0	0	1
1	X	0	0	0	1
1	1	1	0	1	0

строкам сразу двух или всех трех таблиц.

В результате к уже имеющимся тестам добавляются

1) проверки всех случаев расположения обеих прямых -

6 тестов по первой прямой вкладываются в 6 тестов по второй прямой так, чтобы варианты не совпадали, - 6 тестов;

2) выполняется отдельная проверка несовпадения условия $\delta_x = 0$ или $\delta_y = 0$ (в зависимости от того, какой тест был выбран по методу граничных условий) - тест также можно совместить с предыдущими 6 тестами;

4. По методу **предположения об ошибке** добавим тест:

Все коэффициенты - нули.

Всего получили 20 тестов по всем четырем методикам. Если еще попробовать вложить независимые проверки, то возможно число тестов можно еще сократить. **(Не забудьте для каждого теста заранее указывать результат!)**.

3.5 Общая стратегия тестирования

Все методологии проектирования тестов могут быть объединены в общую стратегию. Это оправдано тем, что каждый метод обеспечивает создание определенного набора тестов, но ни один из них сам по себе не может дать полный набор тестов. Приемлемая стратегия состоит в следующем:

1. Если спецификация состоит из комбинации входных условий, то начать рекомендуется с применения метода функциональных диаграмм.
2. В любом случае необходимо использовать анализ граничных значений.
3. Определить правильные и неправильные классы эквивалентности для входных и выходных данных и дополнить, если это необходимо, тесты, построенные на предыдущих шагах.
4. Для получения дополнительных тестов рекомендуется использовать метод предположения об ошибке.
5. Проверить логику программы на полученном наборе тестов. Для этого воспользоваться критериями покрытия решений, покрытия условий, покрытия решений / условий либо комбинаторного покрытия условий. Если необходимость выполнения критерия покрытия требует построения тестов, не встречающихся среди построенных на предыдущих шагах, то следует дополнить уже построенный набор.

3.6 Вопросы для самопроверки

1. Перечислите методы ручного тестирования?
2. Какими возможностями обладают методы ручного тестирования?
3. Перечислите методы «белого ящика»?
4. Какие методы «белого ящика» дают наибольшее количество тестов?
5. Приведите примеры перекрытий методов «белого ящика»?
6. Какие методы «черного ящика» существуют?
7. Дайте анализ методов «черного ящика» применительно к различным видам данных?
8. Приведите пример возможного плана тестирования применительно к большим программным системам?
9. Какие существуют общие принципы тестирования программ?
10. Какие существуют виды ошибок?

Заключение

После выполнения лабораторных работ студенты должны приобрести практические навыки:

- оценки структур данных;
- разработки структур данных;
- оценки методов обработки данных;
- оценки эффективности программ;
- оценки качества программ;
- тестирования программ.

Полученные результаты оценки основных используемых структур данных, алгоритмов работы, а также фрагменты кодов программ, оценочных таблиц и таблиц результатов тестирования должны быть представлены в отчете.

Уровень освоения теоретического материала и правильность предлагаемых решений определяется во время защиты лабораторных работ при наличии отчета.

Список литературы

1. Гагарина Л. Г., Колдаев В. Д., Алгоритмы и структуры данных. - М: Финансы и статистика, 2009 - : 303с.
2. Ахо А.В. , Хопкрофт Д.Э., Ульман Д.Д., Структуры данных и алгоритмы. – М: Вильямс, 2000 - 384 с.
3. Кинг Д., Создание эффективного программного обеспечения. - М: Мир, 1991 – 288с.
3. Вирт Н. Алгоритмы и структуры данных. -М: ДМК Пресс, 2010 -274с.
4. Иванова Г.С. Технология программирования: Учебник. - М: Кнорус, 2011 – 333 с.
5. Иванова Г.С. Программирование. -М.: Кнорус, 2013 - 426с.
6. Майерс Г. Искусство тестирования программ / Пер. с англ. под ред. Б. А. Позина. - М.: Финансы и статистика, 1982.-176с.,
7. Иванова Г.С. Технология программирования. -М.: КноРус – 2011- 333с.