УЗБЕКСКОЕ АГЕНТСТВО СВЯЗИ И ИНФОРМАТИЗАЦИИ

ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра «Технологии программирования»

МЕТОДИЧЕСКОЕ ПОСОБИЕ

ДЛЯ ЛАБОРАТОРНЫХ ЗАНЯТИЙ ПО ДИСЦИПЛИНЕ

«Структуры данных и алгоритмов»

для студентов направлений образования

5521900 – Информатика и информационные технологии

Ташкент 2008

Авторы: Ташкентский университет информационных технологий, заведующий кафедрой «Технологии программирования» д.ф.-м.н.,, проф. Назиров Ш.А., старший преподаватель кафедры «Технологии программирования» И.Х.Бабакулов, Н.А.Арипова, магистрант кафедры «Технологии программирования» Л.Х.Миндулина.

Данное методическое пособие посвящено выполнению лабораторных работ (занятий) по предмету "Структуры данных и алгоритмов", где отражены такие понятия как вычисление времени выполнения программ, типы и структуры данных, абстрактные типы данных (АТД), списки, стек, массив, очередь, АТД основанных на множествах и их программные реализации, ориентированные и неориентированные графы и их программные реализации, внутренняя и внешняя сортировка и их программные реализации, рекурсивные процедуры и рекуррентные соотношения и их программные реализации, алгоритмы «разделяй и властвуй», динамического программирования, «жадные» алгоритмы, алгоритмы поиска и их программные реализации, а также алгоритмы управления внешней памятью и их программные реализации.

Методическое пособие рассчитано на студентов, аспирантов и преподавателей соответствующего профиля.

Рецензенты:

доцент УзНУ
кандидат физ.-мат.наук, А.Хайдаров

Старший научный сотрудник
Института Математики и
Информационных технологий
кандидат физ.-мат.наук, Ф.М.Нуралиев

Введение

Во многих случаях значительные объемы информации, подлежащие обработке, в некотором смысле представляют абстракцию некоторого фрагмента реального мира. Информация, поступающая в машину, состоит из определенного множества данных, относящихся к какой-то проблеме – это именно те данные, которые считаются относящимися к данной конкретной задаче и из которых, как мы надеемся, можно получить (вывести) желанный ответ. Мы говорим о данных как об абстрактном представлении реалий, поскольку некоторые свойства и характеристики объектов при этом игнорируются. Считается, что для конкретной задачи они не являются существенными, определяющими. Поэтому абстрагирование – это упрощение фактов.

Решая любую задачу с помощью машины или без нее, необходимо выбрать уровень абстрагирования, т.е. определить множество данных, представляющих реальную ситуацию. При выборе следует руководствоваться той задачей, которую необходимо решить. Затем надлежит выбрать способ представления этой информации. Здесь уже необходимо ориентироваться на те средства, с помощью которых решается задача, т.е. учитывать возможности, представляемые вычислительной машиной. В большинстве случаев эти два этапа не бывают полностью независимыми.

Выбор представления данных часто является довольно трудной проблемой, ибо не определяется доступными средствами. Всегда нужно принимать во внимание и операции, которые выполняются над этими данными.

Поэтому проектирование любой вычислительной системы вне зависимости от уровня ее сложности ставит разработчик этой системы перед необходимостью выбора тех или иных структур данных, в рамках которых осуществляется отображение соответствующей предметной области. Появившиеся в последние годы современные языки программирования, представляют программисту возможность определять необходимые типы данных, наиболее полно соответствующие решаемой задаче. Это позволяет осуществлять ее решение на том уровне абстракции, который наиболее полно соответствует специфике задачи. Содержание понятий типа, абстракции, и в частности так называемого абстрактного типа данных, играет все возрастающую роль в современных представлениях о разработке языков программирования, и стало объектом интенсивного изучения и обсуждения.

Информатику можно разделить на два направления, одно из которых связано с изучением алгоритмов, другое – с выявлением алгоритмов в реальных явлениях. Второе направление связано с решением сложных проблем. Обрабатываемая с помощью современных ЭВМ информация не подчиняется физическим законам, субъективна, часто является недостаточно ясной и может отличаться в зависимости от принятой точки зрения и способа мышления. Однако необходимо уметь оперировать даже такими данными.

Выяснение структуры данных связано, во-первых, с определением основных образующих структуру элементов и связей между ними, а во-вторых, с выяснением динамики данных.

Можно считать, что определение семантики данных состоит в определении упомянутых выше двух факторов, касающихся статистических и динамических аспектов информации.

Конкретизации в компьютерах структуры данных является одной из задач информатики. Автор языка программирования Паскаль Н. Вирт в книге «Algorithms + Data Structures = Programs» утверждал, что при программировании необходимо учитывать факторы, связанные с отображением как абстрактных структур самого алгоритма в структуру управления программой.

Постепенно языки освобождались от влияния конкретного компьютера, производилось их абстрагирование и обобщение. Наряду с этим в последнее время стала активно развиваться концепция абстрактных типов данных. Необходимо иметь возможность оперировать не просто численными или символьными данными, но и другими типами данных и более сложными структурами данных. Поэтому существует необходимость в дальнейших исследованиях методов абстракции данных и их реализации.

Методическое пособие состоит из восьми лабораторных работ.

Первая лабораторная работа посвящена способам вычисления времени выполнения программ и пошаговой „кристаллизации" алгоритмов

Вторая лабораторная работа посвящена программной реализации абстрактных типов данных (АТД) «Список», «Стек», «Очередь», «Отображение», рекурсивных процедур и программной реализации деревьев

Третья лабораторная работа посвящена программной реализации АТД, основанных на множествах, также программной реализации специальных методов представления множеств.

Четвертая лабораторная работа посвящена программной реализации представления ориентированных и неориентированных графов

Пятая лабораторная работа посвящена программной реализации алгоритмов внутренней и внешней сортировки.

Шестая лабораторная работа посвящена программной реализации рекурсивных процедур и рекуррентных соотношений.

Седьмая лабораторная работа посвящена программной реализации алгоритмов «Разделяй и властвуй», динамическое программирование, «жадных» алгоритмов и алгоритмов поиска

Восьмая лабораторная работа посвящена описанию структуры данных и алгоритмов внешней памяти, а также алгоритмы управления памятью.

В каждой лабораторной работе приведены примеры и задание для самостоятельного выполнения.

При написании данного методического пособия использованы литературные источники [1-21] и многолетний опыт преподавания авторов по дисциплине «Структуры данных и алгоритмов» в Ташкентском университете информационных технологий.

Лабораторная работа - 1.

Тема: Вычисление времени выполнения программ.

Цель работы: изучение методов и способов вычисление времени выполнения программ технологии разработки алгоритмов на примере технологии «пошаговой кристаллизации» и методов анализа временной сложности алгоритмов.

В результате выполнения лабораторной работы студенты должны:

- *знать* методы и способы вычисление времени выполнения программ ,этапы процесса разработки программы; метод «пошаговой кристаллизации»; правила, применяемые при оценке сложности программ;
- *уметь* вычислит времени выполнения программ ,разрабатывать и отлаживать программы при помощи метода «пошаговой кристаллизации»; оценивать временную эффективность программ.

1.1. Время выполнения программ

В процессе решения прикладных задач выбор подходящего алгоритма вызывает определенные трудности. В самом деле, на чем основывать свой выбор, если алгоритм должен удовлетворять следующим противоречащим друг другу требованиям.

1. Быть простым для понимания, перевода в программный код и отладки.
2. Эффективно использовать компьютерные ресурсы и выполняться по возможности быстро.

Если написанная программа должна выполняться только несколько раз, то первое требование наиболее важно. Стоимость рабочего времени программиста обычно значительно превышает стоимость машинного времени выполнения программы, поэтому стоимость программы оптимизируется по стоимости написания (а не выполнения) программы. Если мы имеем дело с задачей, решение которой требует значительных вычислительных затрат, то стоимость выполнения программы может превысить стоимость написания программы, особенно если программа должна выполняться многократно. Поэтому, с финансовой точки зрения, более предпочтительным может стать сложный комплексный алгоритм (в надежде, что результирующая программа будет выполняться существенно быстрее, чем более простая программа). Но и в этой ситуации разумнее сначала реализовать простой алгоритм, чтобы определить, как должна себя вести более сложная программа. При построении сложной программной системы желательно реализовать ее простой прототип, на котором можно провести необходимые измерения и смоделировать ее поведение в целом, прежде чем приступать к разработке окончательного варианта. Таким образом, программисты должны быть осведомлены не только о методах построения быстрых программ, но и знать, когда их следует применить (желательно с минимальными программистскими усилиями).

1.2. Измерение времени выполнения программ

На время выполнения программы влияют следующие факторы.

1. Ввод исходной информации в программу.
2. Качество скомпилированного кода исполняемой программы.
3. Машинные инструкции (естественные и ускоряющие), используемые для выполнения программы.
4. Временная сложность алгоритма соответствующей программы.[1]

Поскольку время выполнения программы зависит от ввода исходных данных, его можно определить как функцию от исходных данных. Но зачастую время выполнения программы зависит не от самих исходных данных, а от их "размера". В этом отношении хорошим примером являются задачи *сортировки*, которые мы подробно рассмотрим в главе 8. В задачах сортировки в качестве входных данных выступает список элементов, подлежащих сортировке, а в качестве выходного результата — те же самые элементы, отсортированные в порядке возрастания или убывания. Например, входной список 2, 1, 3, 1, 5, 8 будет преобразован в выходной список 1, 1, 2, 3, 5, 8 (в данном случае список *отсортирован в порядке возрастания*). Естественной мерой объема входной информации для программы сортировки будет число элементов, подлежащих сортировке, или, другими словами, длина входного списка. В общем случае длина входных данных — подходящая мера объема входной информации, и если не будет оговорено иное, то в качестве меры объема входной информации мы далее будем понимать именно длину входных данных.

Обычно говорят, что время выполнения программы имеет порядок $T(n)$ от входных данных размера n. Например, некая программа имеет время выполнения $T(n) = cn^2$, где c — константа. Единица измерения $T(n)$ точно не определена, но мы будем понимать $T(n)$ как количество инструкций, выполняемых на идеализированном компьютере.

Для многих программ время выполнения действительно является функцией входных данных, а не их размера. В этой ситуации мы определяем $T(n)$ как время выполнения *в наихудшем случае*, т.е. как максимум времени выполнения по всем входным данным размера л. Мы также будем рассматривать $T_{ср}(n)$ как среднее (в статистическом смысле) время выполнения по всем входным данным размера n. Хотя $T_{ср}(n)$ является достаточно объективной мерой времени выполнения, но часто нельзя предполагать (или обосновать) равнозначность всех входных данных. На практике среднее время выполнения найти сложнее, чем наихудшее время выполнения, так как математически это трудноразрешимая задача и, кроме того, зачастую не имеет простого определения понятие "средних" входных данных. Поэтому в основном мы будем использовать наихудшее время выполнения как меру временной сложности алгоритмов, но не будем забывать и о среднем времени выполнения там, где это возможно.

Теперь сделаем замечание о втором и третьем факторах, влияющих на время выполнения программ: о компиляторе, используемом для компиляции программы, и машине, на которой выполняется программа. Эти факторы

влияют на то, что для измерения времени выполнения $T(n)$ мы не можем применить стандартные единицы измерения, такие как секунды или миллисекунды. Поэтому мы можем только делать заключения, подобные "время выполнения такого-то алгоритма пропорционально n^2". Константы пропорциональности также нельзя точно определить, поскольку они зависят от компилятора, компьютера и других факторов.

1.3. Асимптотические соотношения

Для описания скорости роста функций используется O-символика. Например, когда мы говорим, что время выполнения $T(n)$ некоторой программы имеет порядок $O(n^2)$ (читается "о-большое от n в квадрате" или просто "о от n в квадрате"), то подразумевается, что существуют положительные константы c и n_0 такие, что для всех n, больших или равных n_0, выполняется неравенство $T(n) \leq cn^2$.

Пример 1.1. Предположим, что T(0) = 1, T(1) = 4 и в общем случае $T(n) = (n + 1)^2$. Тогда $T(n)$ имеет порядок $O(n^2)$: если положить $n_0 = 1$ и $c = 4$, то легко показать, что для $n \geq 1$ будет выполняться неравенство $(n + 1)^2 \leq 4n^2$. Отметим, что нельзя положить $n_0 = 0$, так как T(0) = 1 и, следовательно, это значение при любой константе c больше $c0^2 = 0$. □

Подчеркнем: мы предполагаем, что все функции времени выполнения определены на множестве неотрицательных целых чисел и их значения также неотрицательны, но необязательно целые. Будем говорить, что $T(n)$ имеет порядок $O(f(n))$, если существуют константы c и n_0 такие, что для всех $n \geq n_0$ выполняется неравенство $T(n) \leq cf(n)$. Для программ, у которых время выполнения имеет порядок $O(f(n))$, говорят, что они имеют *порядок* (или *степень) роста* $f(n)$.

Пример 1.2. Функция $T(n) — 3n^3 + 2n^2$ имеет степень роста $O(n^3)$. Чтобы это показать, надо положить $n_0 = 0$ и $c = 5$, так как легко видеть, что для всех целых $n \geq 0$ выполняется неравенство $3n^3 + 2n^2 \leq 5n^3$. Можно, конечно, сказать, что $T(n)$ имеет порядок $O(n^4)$, но это более слабое утверждение, чем то, что $T(n)$ имеет порядок роста $O(n^3)$.

В качестве следующего примера докажем, что функция 3^n не может иметь порядок $O(2^n)$. Предположим, что существуют константы c и n_0 такие, что для всех $n \geq n_0$ выполняется неравенство $3^n \leq c2^n$. Тогда $c \geq (3/2)^n$ для всех $n \geq n_0$. Но $(3/2)^n$ принимает любое, как угодно большое, значение при достаточно большом n, поэтому не существует такой константы c, которая могла бы мажорировать $(3/2)^n$ для всех n. □

Когда мы говорим, что $T(n)$ имеет степень роста $O(f(n))$, то подразумевается, что $f(n)$ является верхней границей скорости роста $T(n)$. Чтобы указать нижнюю границу скорости роста $T(n)$, используется обозначение: $T(n)$ есть $\Omega(g(n))$ (читается "омега-большое от $g(n)$" или просто "омега от $g(n)$"), это подразумевает существование такой константы c, что

бесконечно часто (для бесконечного числа значений n) выполняется неравенство $T(n) \geq cg(n)$).

Пример 1.3. Для проверки того, что $T(n) = n^3 + 2n^2$ есть $\Omega(n^3)$, достаточно положить $c = 1$. Тогда $T(n) \geq cn^3$ для $n = 0, 1, \ldots$.

Для другого примера положим, что $T(n) = n$ для нечетных $n \geq 1$ и $T(n) = n^2/100$ — для четных $n > 0$. Для доказательства того, что $T(n)$ есть $\Omega(n^2)$, достаточно положить $c = 1/100$ и рассмотреть множество четных чисел n — $0,2,4,6,\ldots$.□

1.4. Ограниченность показателя степени роста

Итак, мы предполагаем, что программы можно оценить с помощью функций времени выполнения, пренебрегая при этом константами пропорциональности. С этой точки зрения программа с временем выполнения $O(n^2)$, например, лучше программы с временем выполнения $O(n^3)$. Константы пропорциональности зависят не только от используемых компилятора и компьютера, но и от свойств самой программы. Пусть при определенной комбинации компилятор-компьютер одна программа выполняется за $100n^2$ миллисекунд, а вторая — за $5n^3$ миллисекунд. Может ли вторая программа быть предпочтительнее, чем первая?

Ответ на этот вопрос зависит от размера входных данных программ. При размере входных данных $n < 20$ программа с временем выполнения $5n^3$ завершится быстрее, чем программа с временем выполнения $100n^2$. Поэтому, если программы в основном выполняются с входными данными небольшого размера, предпочтение необходимо отдать программе с временем выполнения $O(n^3)$.Однако при возрастании n отношение времени выполнения $5n^3/100n^2 = n/20$ также растет. Поэтому при больших n программа с временем выполнения $O(n^2)$ становится предпочтительнее программы с временем выполнения $O(n^3)$. Если даже при сравнительно небольших n, когда время выполнения обеих программ примерно одинаково, выбор лучшей программы представляет определенные затруднения, то естественно для большей надежности сделать выбор в пользу программы с меньшей степенью роста.

Другая причина, заставляющая отдавать предпочтение программам с наименьшей степенью роста времени выполнения, заключается в том, что чем меньше степень роста, тем больше размер задачи, которую можно решить на компьютере. Другими словами, если увеличивается скорость вычислений компьютера, то растет также и размер задач, решаемых на компьютере. Однако незначительное увеличение скорости вычислений компьютера приводит только к небольшому увеличению размера задач, решаемых в течение фиксированного промежутка времени, исключением из этого правила являются программы с низкой степенью роста, как $O(n)$ и $O(n \log n)$.

Пример 1.4. На рис. 1.1 показаны функции времени выполнения (измеренные в секундах) для четырех программ с различной временной сложностью для одного и того же сочетания компилятор-компьютер. Предположим, что можно

использовать 1000 секунд (примерно 17 минут) машинного времени для решения задачи. Какой максимальный размер задачи, решаемой за это время? За 10 секунд каждый из четырех алгоритмов может решить задачи примерно одинакового размера, как показано во втором столбце табл. 1.1.

Предположим, что получен новый компьютер (без дополнительных финансовых затрат), работающий в десять раз быстрее. Теперь за ту же цену можно использовать 10^4 секунд машинного времени — ранее 10^3 секунд. Максимальный размер задачи, которую может решить за это время каждая из четырех программ, показан в третьем столбце табл. 1.1. Отношения значений третьего и второго столбцов приведены в четвертом столбце этой таблицы. Здесь мы видим, что увеличение скорости компьютера на 1 000% приводит к увеличению только на 30% размера задачи, решаемой с помощью программы с временем выполнения $O(2^n)$. Таким образом, 10-кратное увеличение производительности компьютера дает в процентном отношении значительно меньший эффект увеличения размера решаемой задачи. В действительности, независимо от быстродействия компьютера, программа с временем выполнения $O(2^n)$может решать только очень небольшие задачи.

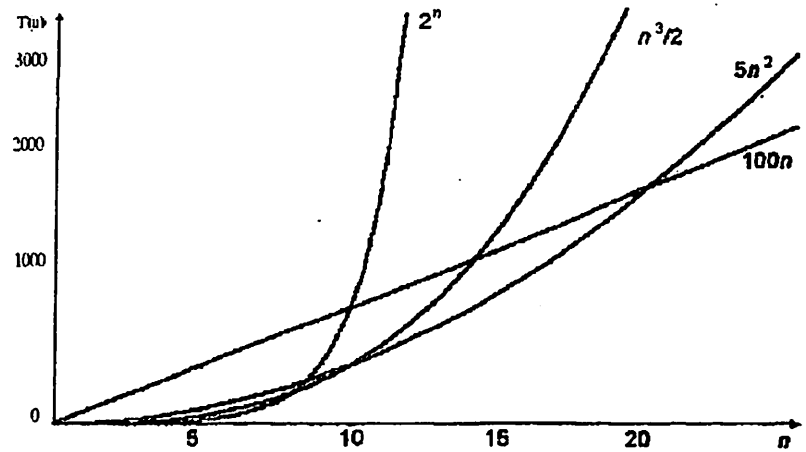


Рис. 1.1. Функции времени выполнения четырех программ

Таблица 1.1. Эффект от 10-кратного увеличения быстродействия компьютера

Время выполнения $T(n)$	Максимальный размер задачи для 10^3 секунд	Максимальный размер задачи для 10^4 секунд	Увеличение максимального размера задачи
$100n$	10	100	10.0
$5n^2$	14	45	3.2
$n^3/2$	12	27	2.3
2^n	10	13	1.3

Из третьего столбца табл. 1.1 ясно видно преимущество программ с временем выполнения $O(n)$: 10-кратное увеличение размера решаемой задачи при 10-кратноМ увеличении производительности компьютера. Программы с временем выполнения $O(n^3)$ и $O(n^2)$ при увеличении быстродействия компьютера на 1 000% дают увеличение размера задачи соответственно на 230% и 320%. Эти соотношения сохранятся и при дальнейшем увеличении производительности компьютера. □

Поскольку существует необходимость решения задач все более увеличивающегося размера, мы приходим к почти парадоксальному выводу. Так как машинное время все время дешевеет, а компьютеры становятся более быстродействующими, мы надеемся, что сможем решать все большие по размеру и более сложные задачи. Но вместе с тем возрастает значимость разработки и использования эффективных алгоритмов именно с низкой степенью роста функции времени выполнения.

Замечание. Мы хотим еще раз подчеркнуть, что степень роста наихудшего времени выполнения — не единственный или самый важный критерий оценки алгоритмов и программ. Приведем несколько соображений, позволяющих посмотреть на критерий времени выполнения с других точек зрения.

1. Если создаваемая программа будет использована только несколько раз, тогда стоимость написания и отладки программы будет доминировать в общей стоимости программы, т.е. фактическое время выполнения не окажет существенного влияния на общую стоимость. В этом случае следует предпочесть алгоритм, наиболее простой для реализации.
2. Если программа будет работать только с "малыми" входными данными, то степень роста времени выполнения будет иметь меньшее значение, чем константа, присутствующая в формуле времени выполнения. Вместе с тем и понятие "малости" входных данных зависит от точного времени выполнения конкурирующих алгоритмов. Существуют алгоритмы, такие как алгоритм целочисленного умножения (см. [96]), асимптотически самые эффективные, но которые никогда не используют на практике даже для больших задач, так как их константы пропорциональности значительно превосходят подобные константы других, более простых и менее "эффективных" алгоритмов.
3. Эффективные, но сложные алгоритмы могут быть нежелательными, если готовые программы будут поддерживать лица, не участвующие в написании

этих программ. Будем надеяться, что принципиальные моменты технологии создания эффективных алгоритмов широко известны, и достаточно сложные алгоритмы свободно применяются на практике. Однако необходимо предусмотреть возможность того, что эффективные, но "хитрые" алгоритмы не будут востребованы из-за их сложности и трудностей, возникающих при попытке в них разобраться.

4. Известно несколько примеров, когда эффективные алгоритмы требуют таких больших объемов машинной памяти (без возможности использования более медленных внешних средств хранения), что этот фактор сводит на нет преимущество "эффективности" алгоритма.

5. В численных алгоритмах точность и устойчивость алгоритмов не менее важны, чем их временная эффективность.

1.5. Вычисление времени выполнения программ

Теоретическое нахождение времени выполнения программ (даже без определения констант пропорциональности) — сложная математическая задача. Однако на практике определение времени выполнения (также без нахождения значения констант) является вполне разрешимой задачей — для этого нужно знать только несколько базовых принципов. Но прежде чем представить эти принципы, рассмотрим, как выполняются операции сложения и умножения с использованием О-символики.Пусть $T_1(n)$ и $T_2(n)$ — время выполнения двух программных фрагментов P_1 и P_2, $T_1(n)$ имеет степень роста $O(f(n))$, а $T_2(n)$ — $O(g(n))$. Тогда $T_1(n) + T_2(n)$, т.е. время последовательного выполнения фрагментов P_1 и P_2, имеет степень роста $O(max(f(n), g(n)))$. Для доказательства этого вспомним, что существуют константы c_1, c_2, n_1 и n_2 такие, что при $n \geq n_1$ выполняется неравенство $T_1(n) \leq c_1 f(n)$, и, аналогично, $T_2(n) \leq c_2 g(n)$, если $n \geq n_2$. Пусть $n_0 = \max(n_1, n_2)$. Если $n \geq n_0$, то, очевидно, что $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$. Отсюда вытекает, что при $n \geq n_0$ справедливо неравенство $T_1(n) + T_2(n) \leq (c_1 + c_2) \max(f(n), g(n))$. Последнее неравенство и означает, что $T_1(n) + T_2(n)$ имеет порядок роста $O(max(f(n), g(n)))$.

Пример 1.5. *Правило сумм*, данное выше, используется для вычисления времени последовательного выполнения программных фрагментов с циклами и ветвлениями. Пусть есть три фрагмента с временами выполнения соответственно $O(n^2)$, $O(n^3)$ и $O(n \log n)$. Тогда время последовательного выполнения первых двух фрагментов имеет порядок $O(max(n^2, n^3))$, т.е. $O(n^3)$. Время выполнения всех трех фрагментов имеет порядок $0(\max(n^3, n \log n))$, это то же самое, что $O(n^3)$. □

В общем случае время выполнения конечной последовательности программных фрагментов, без учета констант, имеет порядок фрагмента с наибольшим временем выполнения. Иногда возможна ситуация, когда порядки роста времен нескольких фрагментов *несоизмеримы* (ни один из них не больше, чем другой, но они и не равны). Для примера рассмотрим два фрагмента с временем выполнения $0(f(n))$ и $0(g(n))$, где

$$f(n)=\begin{cases} n^4, \text{если } n \text{ четное;} \\ n^1, \text{если } n \text{ нечетное.} \end{cases} \qquad g(n)=\begin{cases} n^2, \text{если } n \text{ четное;} \\ n^3, \text{если } n \text{ нечетное.} \end{cases}$$

В данном случае правило сумм можно применить непосредственно и получить время выполнения $O(max(f(n), g(n)))$, т.е. n^4 при n четном и n^3, если n нечетно.

Из правила сумм также следует, что если $g(n) \leq f(n)$ для всех n, превышающих n_0, то выражение $O(f(n) + g(n))$ эквивалентно $O(f(n))$. Например, $O(n^2 + n)$ то же самое, что $O(n^2)$.

Правило произведений заключается в следующем. Если $T_1(n)$ и $T_2(n)$ имеют степени роста $O(f(n))$ и $O(g(n))$ соответственно, то произведение $T_1(n)T_2(n)$ имеет степень роста $O(f(n)g(n))$. Читатель может самостоятельно доказать это утверждение, используя тот же подход, который применялся при доказательстве правила сумм. Из правила произведений следует, что $O(cf(n))$ эквивалентно $O(f(n))$, если c — положительная константа. Например, $O(n^2/2)$ эквивалентно $O(n^2)$.

Прежде чем переходить к общим правилам анализа времени выполнения программ, рассмотрим простой пример, иллюстрирующий процесс определения времени выполнения.

Пример 1.6. Рассмотрим программу сортировки *bubble* (пузырек), которая упорядочивает массив целых чисел в возрастающем порядке методом "пузырька" (листинг 1.4). За каждый проход внутреннего цикла (операторы (3)-(6)) "пузырек" с наименьшим элементом "всплывает" в начало массива.

Листинг 1.1. Сортировка методом „пузырька"

```
procedure bubble ( var A: array [1..n] of integer );
{ Процедура упорядочивает массив A в возрастающем порядке }
var
i, j, temp: integer; begin
(1)        for i:= 1 to n - 1 do
(2)              for j: = n downto i+1 do
(3)                    if A [j -1] > A[j] then begin
{ перестановка местами A[j-l] и A[j] }
(4)                          temp:= A [j -1];
(5)                               A[j - 1]:= A[j];
(6)                               A[j]:= temp;
end
end; { bubble }
```

Число элементов n, подлежащих сортировке, может служить мерой объема входных данных. Сначала отметим, что все операторы присваивания имеют некоторое постоянное время выполнения, независящее от размера входных данных. Таким образом, операторы (4) - (6) имеют время выполнения порядка $O(1)$. Запись $O(1)$ означает "равнозначно некой константе". В

соответствии с правилом сумм время выполнения этой группы операторов равно $O(\max(1, 1, 1)) = O(1)$.

Теперь мы должны подсчитать время выполнения условных и циклических операторов. Операторы if и for вложены друг в друга, поэтому мы пойдем от внутренних операторов к внешним, последовательно определяя время выполнения условного оператора и каждой итерации цикла. Для оператора if проверка логического выражения занимает время порядка O(1). Мы не знаем, будут ли выполняться операторы в теле условного оператора (строки (4) - (6)), но поскольку мы ищем наихудшее время выполнения, то, естественно, предполагаем, что они выполняются. Таким образом, получаем, что время выполнения группы операторов (3) - (6) имеет порядок O(1).

Далее рассмотрим группу (2) - (6) операторов внутреннего цикла. Общее правило вычисления времени выполнения цикла заключается в суммировании времени выполнения каждой итерации цикла. Для операторов (2) - (6) время выполнения на каждой итерации имеет порядок O(1). Цикл выполняется $n - i$ раз, поэтому по правилу произведений общее время выполнения цикла имеет порядок $O((n - i) \times 1)$, что равно $O(n - i)$.

Теперь перейдем к внешнему циклу, который содержит все исполняемые операторы программы. Оператор (1) выполняется $n - 1$ раз, поэтому суммарное время выполнения программы ограничено сверху выражением

$$\sum_{i=1}^{n-1}(n-i) = n(n \cdot 1)/2 = n^2/2 - n/2,$$

которое имеет порядок $O(n^2)$. Таким образом, программа "пузырька" выполняется за время, пропорциональное квадрату числа элементов, подлежащих упорядочиванию. В главе 8 мы рассмотрим программы с временем выполнения порядка $O(n \log n)$, которое существенно меньше $O(n^2)$, поскольку при больших n $\log n$ значительно меньше n. □

Перед формулировкой общих правил анализа программ позвольте напомнить, что нахождение точной верхней границы времени выполнения программ только в редких случаях так лее просто, как в приведенном выше примере, в общем случае эта задача является интеллектуальным вызовом исследователю. Поэтому не существует исчерпывающего множества правил анализа программ. Мы можем дать только некоторые советы и проиллюстрировать их с разных точек зрения примерами, приведенными в этой книге.

Теперь дадим несколько правил анализа программ. В общем случае время выполнения оператора или группы операторов можно параметризовать с помощью размера входных данных и/или одной или нескольких переменных. Но для времени выполнения программы в целом допустимым параметром может быть только n, размер входных данных.

1. Время выполнения операторов присваивания, чтения и записи обычно имеет порядок $O(1)$. Есть несколько исключений из этого правила, например в языке PL/1, где можно присваивать большие массивы, или в любых других языках, допускающих вызовы функций в операторах присваивания.

2. Время выполнения последовательности операторов определяется с помощью правила сумм. Поэтому степень роста времени выполнения последовательности операторов без определения констант пропорциональности совпадает с наибольшим временем выполнения оператора в данной последовательности.
3. Время выполнения условных операторов состоит из времени выполнения условно исполняемых операторов и времени вычисления самого логического выражения. Время вычисления логического выражения обычно имеет порядок $O(1)$. Время для всей конструкции if-then-else состоит из времени вычисления логического выражения и наибольшего из времени, необходимого для выполнения операторов, исполняемых при значении логического выражения true (истина) и при значении false (ложь).
4. Время выполнения цикла является суммой времени всех исполняемых итераций цикла, в свою очередь состоящих из времени выполнения операторов тела цикла и времени вычисления условия прекращения цикла (обычно последнее имеет порядок $O(1)$). Часто время выполнения цикла вычисляется, пренебрегая определением констант пропорциональности, как произведение количества выполненных итераций цикла на наибольшее возможное время выполнения операторов тела цикла. Время выполнения каждого цикла, если в программе их несколько, должно определяться отдельно.

1.6.Вызовы процедур

Для программ, содержащих несколько процедур (среди которых нет рекурсивных), можно подсчитать общее время выполнения программы путем последовательного нахождения времени выполнения процедур, начиная с той, которая не имеет вызовов других процедур. (Вызов процедур мы определяем по наличию оператора call.) Так как мы предположили, что все процедуры нерекурсивные, то должна существовать хотя бы одна процедура, не имеющая вызовов других процедур. Затем можно определить время выполнения процедур, вызывающих эту процедуру, используя уже вычисленное время выполнения вызываемой процедуры. Продолжая этот процесс, найдем время выполнения всех процедур и, наконец, время выполнения всей программы.

Если есть рекурсивные процедуры, то нельзя упорядочить все процедуры таким образом, чтобы каждая процедура вызывала только процедуры, время выполнения которых подсчитано на предыдущем шаге. В этом случае мы должны с каждой рекурсивной процедурой связать временную функцию $T(n)$, где n определяет объем аргументов процедуры. Затем мы должны получить *рекуррентное соотношение* для $T(n)$, т.е. уравнение (или неравенство) для $T(n)$, где участвуют значения $T(k)$ для различных значений k.

Техника решения рекуррентных соотношений зависит от вида этих соотношений, некоторые приемы их решения будут показаны в главе 9. Сейчас же мы проанализируем простую рекурсивную программу.

Пример 1.7. В листинге 1.2 представлена программа вычисления факториала $n!$, т.е. вычисления произведения целых чисел от 1 до n включительно.

Листинг 1.2. Рекурсивная программа вычисления факториала

```
function fact ( n: integer ):integer;
{ fact(n) вычисляет n!}
    begin
(1)     if n <= 1 then
(2)       fact:= 1
        else
(3)         fact:= n * fact(n - 1)
    end; {fact}
```

Естественной мерой объема входных данных для функции *fact* является значение *n*. Обозначим через *T(n)* время выполнения программы. Время выполнения для строк (1) и (2) имеет порядок $O(1)$, а для строки (3) — $O(1) + T(n - 1)$. Таким образом, для некоторых констант c и d имеем

$$T(n) = \begin{cases} c + T(n-1), & \text{если } n > 1, \\ d, & \text{если } n \le 1. \end{cases} \qquad (1.1)$$

Полагая, что $n > 2$, и раскрывая в соответствии с соотношением (1.1) выражение $T(n - 1)$ (т.е. подставляя в (1.1) $n - 1$ вместо n), получим $T(n) = 2c + T(n - 2)$. Аналогично, если $n > 3$, раскрывая $T(n - 2)$, получим $T(n) = 3c + T(n - 3)$. Продолжая этот процесс, в общем случае для некоторого i, $n > i$, имеем $T(n) = ic + T(n - i)$. Положив в последнем выражении $i = n - 1$, окончательно получаем

$$T(n) = c(n - 1) + T(1) = c(n - 1) + d. \qquad (1.2)$$

Из (1.2) следует, что $T(n)$ имеет порядок $O(n)$. Отметим, что в этом примере анализа программы мы предполагали, что операция перемножения двух целых чисел имеет порядок $O(1)$. На практике, однако, программу, представленную в листинге 1.5, нельзя использовать для вычисления факториала при больших значениях n, так как размер получаемых в процессе вычисления целых чисел может превышать длину машинного слова. □

Общий метод решения рекуррентных соотношений, подобных соотношению из примера 1.10, состоит в последовательном раскрытии выражений $T(k)$ в правой части уравнения (путем подстановки в исходное соотношение k вместо n) до тех пор, пока не получится формула, у которой в правой части отсутствует T (как в формуле (1.2)). При этом часто приходится находить суммы различных последовательностей; если значения таких сумм нельзя вычислить точно, то для сумм находятся верхние границы, что позволяет, в свою очередь, получить верхние границы для $T(n)$.

1.7. Практика программирования

Приведем несколько рекомендаций и соображений, которые вытекают из нашего практического опыта построения алгоритмов и реализации их в виде программ. Некоторые из этих рекомендаций могут показаться очевидными или даже банальными, но их полезность можно оценить только при решении реальных задач, а не исходя из теоретических предпосылок. Читатель может проследить применение приведенных рекомендаций при разработке программ в

этой книге, а также может проверить их действенность в собственной практике программирования.

1. ***Планируйте этапы разработки программы.*** *Описать* этапы разработки программы: сначала черновой набросок алгоритма в неформальном стиле, затем псевдопрограмма, далее — последовательная формализация псевдопрограммы, т.е. переход к уровню исполняемого кода. Эта стратегия организует и дисциплинирует процесс создания конечной программы, которая будет простой в отладке и в дальнейшей поддержке и сопровождении.

2. ***Применяйте инкапсуляцию.*** Все процедуры, реализующие АТД, поместите в одно место программного листинга. В дальнейшем, если возникнет необходимость изменить реализацию АТД, можно будет корректно и без особых затрат внести какие-либо изменения, так как все необходимые процедуры локализованы в одном месте программы.

3. ***Используйте и модифицируйте уже существующие программы.*** Один из неэффективных подходов к процессу программирования заключается в том, что каждый новый проект рассматривается "с нуля", без учета уже существующих программ. Обычно среди программ, реализованных на момент начала проекта, можно найти такие, которые если решают не всю исходную задачу, то хотя бы ее часть. После создания законченной программы полезно оглянуться вокруг и посмотреть, где еще ее можно применить (возможно, вариант ее применения окажется совсем непредвиденным).

4. ***Станьте "кузнецом" инструментов.*** На языке программистов *инструмент* (tool) — это программа с широким спектром применения. При создании программы подумайте, нельзя ли ее каким-либо образом обобщить, т.е. сделать более универсальной (конечно, с минимальными программистскими усилиями). Например, предположим, что вам необходимо написать программу, составляющую расписание экзаменов. Вместо заказанной программы можно написать программу-инструмент, раскрашивающий вершины обычного графа (по возможности минимальным количеством цветов) таким образом, чтобы любые две вершины, соединенные ребром, были закрашены в разные цвета. В контексте расписания экзаменов вершины графа — это классы, цвета — время проведения экзаменов, а ребра, соединяющие две вершины-класса, обозначают, что в этих классах экзамены принимает одна и та же экзаменационная комиссия. Такая программа раскраски графа вместе с подпрограммой перевода списка классов в множество вершин графа и цветов в заданные временные интервалы проведения экзаменов составит расписание экзаменов. Программу раскраски можно использовать для решения задач, совсем не связанных с составлением расписаний, например для задания режимов работы светофоров на сложном перекрестке

5. ***Программируйте на командном уровне.*** Часто бывает, что в библиотеке программ не удается найти программу, необходимую для выполнения именно нашей задачи, но мы можем адаптировать для этих целей ту или иную программу-инструмент. Развитые операционные системы предоставляют программам, разработанным для различных платформ, возможность совместной работы в сети вовсе без модификации их кода, за

исключением списка команд операционной системы. Чтобы сделать команды компонуемыми, как правило, необходимо, чтобы каждая из них вела себя как *фильтр*, т.е. как программа с одним входным и одним выходным файлом. Отметим, что можно компоновать любое количество фильтров, и, если командный язык операционной системы достаточно интеллектуален, достаточно просто составить список команд в том порядке, в каком они будут востребованы программой.

Пример 1.8. В качестве примера рассмотрим программу *spell*, написанную Джонсоном (S.C. Johnson) с использованием команд UNIX. На вход этой программы поступает файл f_1, состоящий из текста на английском языке, на выходе получаем все слова из f_1, не совпадающие со словами из небольшого словаря[2]. Эта программа воспринимает имена и правильно написанные слова, которых нет в словаре, как орфографические ошибки. Но обычно выходной список ошибок достаточно короткий, поэтому его можно быстро пробежать глазами и определить, какие слова в нем не являются ошибками.

Первый фильтр, используемый программой *spell*, — это команда *translate*, которая имеет соответствующие параметры и заменяет прописные буквы на строчные, пробелы — на начало новых строк, а остальные символы оставляет без изменений. На выходе этой команды мы получаем файл f_2, состоящий из тех же слов, что и файл f_1 , но каждое слово расположено в отдельной строке. Далее выполняется команда *sort*, упорядочивающая строки входного файла в алфавитном порядке. Результатом выполнения этой команды является файл f_3, содержащий отсортированный список (возможно, с повторениями) слов из файла f_2. Затем команда *unique* удаляет повторяющиеся строки в своем входном файле f_3, создавая выходной файл f_4, содержащий слова из исходного файла (без прописных букв и повторений), упорядоченные в алфавитном порядке. Наконец, к файлу f_4 применяется команда *diff*, имеющая параметр, который указывает на файл f_5, содержащий в алфавитном порядке слова из словаря, расположенные по одному в строке. Результатом этой команды будет список слов из файла f_4, которые не совпадают со словами из файла f_5, т. е. те слова из исходного списка, которых нет в словаре. Программа *spell* состоит из следующей последовательности команд:

```
spell·   translate [A-Z] —> [a-z], пробел —> новая строка
         sort
         unique
         diff словарь
```

Командный уровень программирования требует дисциплины от команды программистов. Они должны писать программы как фильтры везде, где это возможно, и создавать программы-инструменты вместо узкоспециализированных программ всегда, когда для этого есть условия. Существенным вознаграждением за это будет значительное повышение вашего коэффициента отношения результата к затраченным усилиям.

1.8. Пошаговая „кристаллизация" алгоритмов

1.8.1.Алгоритмы

Когда построена (подобрана) подходящая модель исходной задачи, то естественно искать решение в терминах этой модели. На этом этапе основная цель заключается в построении решения в форме *алгоритма,* состоящего из конечной последовательности инструкций, каждая из которых имеет четкий смысл и может быть выполнена с конечными вычислительными затратами за конечное время. Целочисленный оператор присваивания $x:=y+z$ — пример инструкции, которая будет выполнена с конечными вычислительными затратами. Инструкции могут выполняться в алгоритме любое число раз, при этом они сами определяют число повторений. Однако мы требуем, чтобы при любых входных данных алгоритм завершился после выполнения конечного числа инструкций. Таким образом, программа, написанная на основе разработанного алгоритма, при любых начальных данных никогда не должна приводить к бесконечным циклическим вычислениям.

Есть еще один аспект определения алгоритмов, о котором необходимо сказать. Выше мы говорили, что алгоритмические инструкции должны иметь "четкий смысл" и выполняться с "конечными вычислительными затратами". Естественно, то, что понятно одному человеку и имеет для него "четкий смысл", может совершенно иначе представляться другому. То же самое можно сказать о понятии "конечных затрат": на практике часто трудно доказать, что при любых исходных данных выполнение последовательности инструкций завершится, даже если мы четко понимаем смысл каждой инструкции. В этой ситуации, учитывая все аргументы за и против, было бы полезно попытаться достигнуть соглашения о "конечных затратах" в отношении последовательности инструкций, составляющих алгоритм. Однако кажущаяся сложность подобных доказательств может быть обманчивой.

Кроме программ на языке Pascal, мы часто будем представлять алгоритмы с помощью *псевдоязыка* программирования, который комбинирует обычные конструкции языков программирования с выражениями на "человеческом" языке. Мы используем Pascal как язык программирования, но практически любой другой язык программирования может быть использован вместо него для представления алгоритмов, рассматриваемых в этой книге.

Следующие примеры иллюстрируют основные этапы создания компьютерной программы.

Пример 1.9. Рассмотрим математическую модель, используемую для управления светофорами на сложном перекрестке дорог. Мы должны создать программу, которая в качестве входных данных использует множество всех допустимых поворотов на перекрестке (продолжение прямой дороги, проходящей через перекресток, также будем считать "поворотом") и разбивает это множество на несколько групп так, чтобы все повороты в группе могли выполняться одновременно, не создавая проблем друг для друга. Затем мы сопоставим с каждой группой поворотов соответствующий режим работы светофоров на перекрестке. Желательно минимизировать число разбиений

исходного множества поворотов, поскольку при этом минимизируется количество режимов работы светофоров на перекрестке.

Для примера на рис. 1.2 показан перекресток возле Принстонского университета, известный сложностью его преодоления. Обратите внимание, что дороги *C* и *E* односторонние, остальные — двухсторонние. Всего на этом перекрестке возможно 13 поворотов. Некоторые из этих поворотов, такие как *AB* (поворот с дороги *A* на дорогу *B)* и *EC,* могут выполняться одновременно. Трассы других поворотов, например *AD* и *EB,* пересекаются, поэтому их нельзя выполнять одновременно. Режимы работы светофоров должны учитывать эти обстоятельства и не допускать одновременного выполнения таких поворотов, как *AD* и *EB,* но могут разрешать совместное выполнение поворотов, подобных *AB* и EC

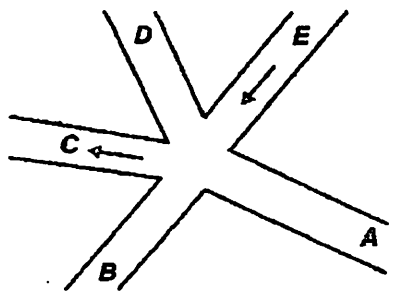


Рис. 1.2. Сложный перекресток

Для построения модели этой задачи можно применить математическую структуру, известную как граф. *Граф* состоит из множества точек, которые называются *вершинами,.* и совокупности линий *(ребер),* соединяющих эти точки. Для решения задачи управления движением по перекрестку можно нарисовать граф, где вершины будут представлять повороты, а ребра соединят ту часть вершин-поворотов, которые нельзя выполнить одновременно. Для нашего перекрестка (рис. 1.2) соответствующий граф показан на рис. 1.3, а в табл. 1.2 дано другое представление графа — в виде, таблицы, где на пересечении строки *i* и столбца j стоит 1 тогда и только тогда, когда существует ребро между вершинами *i* и j.

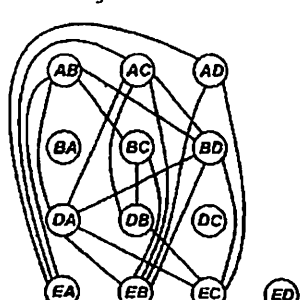


Рис. 1.3. Граф, показывающий несовместимые повороты

Модель в виде графа поможет в решении исходной задачи управления светофорами. В рамках этой модели можно использовать решение, которое дает математическая задача *раскраски графа:* каждой вершине графа надо так задать цвет, чтобы никакие две соединенные ребром вершины не имели одинаковый цвет, и при этом по возможности использовать минимальное количество цветов. Нетрудно видеть, что при такой раскраске графа несовместимым поворотам будут соответствовать вершины, окрашенные в разные цвета.

Проблема раскраски графа изучается математиками уже несколько десятилетий, но теория алгоритмов мало что может сказать о решении этой проблемы. К сожалению, задача раскраски произвольного графа минимальным количеством цветов принадлежит классу задач, которые называются *NP-полными задачами* и для которых все известные решения относятся к типу "проверь все возможности" или "перебери все варианты". Для раскраски графа это означает, что сначала для закраски вершин используется один цвет, затем — два цвета, потом — три и т.д., пока не будет получена подходящая раскраска вершин. В некоторых частных случаях можно предложить более быстрое решение, но в общем случае не существует более эффективного (в значительной степени) алгоритма решения задачи раскраски, чем алгоритм полного перебора возможных вариантов.

Таким образом, поиск оптимального решения задачи раскраски графа требует больших вычислительных затрат. В этой ситуации можно воспользоваться одним из следующих трех подходов. Если граф небольшой, можно попытаться найти оптимальное решение, перебрав все возможные варианты раскраски.

Однако этот подход не приемлем для больших графов*, так как программно трудно организовать эффективный перебор всех вариантов. Второй подход предполагает использование дополнительной информации об исходной задаче. Желательно найти какие-то особые свойства графа, которые исключали бы необходимость полного перебора всех вариантов раскраски для нахождения оптимального решения. В третьем подходе мы немного изменяем постановку задачи и ищем не оптимальное решение, а близкое к оптимальному.

Таблица 1.2. Таблица несовместимых поворотов

	AB	*AC*	*АЛ*	*BA*	*BC*	*BD*	*DA*	*DB*	*DC*	*EA*	*EB*	*EC*	*ED*
AB					1	1	1			1			
AC						1	1	1		1	1		
AD										1	1	1	
BA													
BC	1							1			1		
BD	1	1					1				1	1	
DA	1	1				1					1	1	
DB		1			1							1	
DC													
EA	1	1	1										
EB		1	1		1	1	1						
EC			1			1	1	1					
ED													

Если мы откажемся от требования минимального количества цветов раскраски графа, то можно построить алгоритмы раскраски, которые работают значительно быстрее, чем алгоритмы полного перебора. Алгоритмы, которые быстро находят "подходящее", но не оптимальное решение, называются *эвристическими.*

Примером рационального эвристического алгоритма может служить следующий "жадный" алгоритм раскраски графа. В этом алгоритме сначала мы пытаемся раскрасить как можно больше вершин в один цвет, затем закрашиваем во второй цвет также по возможности максимальное число оставшихся вершин, и т.д. При закраске вершин в новый цвет мы выполняем следующие действия.

1. Выбираем произвольную незакрашенную вершину и назначаем ей новый цвет.

2. Просматриваем список незакрашенных вершин и для каждой из них определяем, соединена ли она ребром с вершиной, уже закрашенной в новый цвет. Если не соединена, то к этой вершине также применяется новый цвет.Этот алгоритм назван "жадным" из-за того, что каждый цвет применяется к максимально большому числу вершин, без возможности пропуска некоторых из них или перекраски ранее закрашенных. Возможны ситуации, когда, будь алгоритм менее "жадным" и пропустил бы некоторые вершины при закраске новым цветом, мы получили бы раскраску графа меньшим количеством цветов. Например, для раскраски графа на рис. 1..4 можно было бы применить два цвета, закрасив вершину 1 в красный цвет, а затем, пропустив вершину 2, закрасить в красный цвет вершины 3 и 4. Но "жадный" алгоритм, основываясь на порядковой очередности вершин, закрасит в красный цвет вершины 1 и 2, для закраски остальных вершин потребуются еще два цвета.

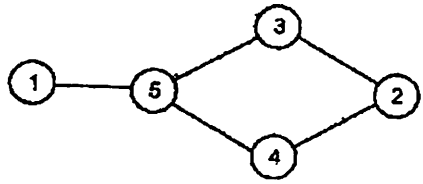


Рис. 1.4. Граф

Применим описанный алгоритм для закраски вершин графа нашей задачи (рис.1.3), при этом первой закрасим вершину *AB* в синий цвет. Также можно закрасить в синий цвет вершины *AC*, *AD* и *BA*, поскольку никакие из этих четырех вершин не имеют общих ребер. Вершину *BC* нельзя закрасить в этот цвет, так как существует ребро между вершинами *AB* и *BC*. По этой же причине мы не можем закрасить в синий цвет вершины *BD*, *DA* и *DB* — все они имеют хотя бы по одному ребру, связывающему их с уже закрашенными в синий цвет вершинами. Продолжая перебор вершин, закрашиваем вершины *DC* и *ED* в синий цвет, а вершины *EA*, *EB* и *EC* оставляем незакрашенными.

Теперь применим второй цвет, например красный. Закраску этим цветом начнем с вершины *BC*. Вершину *BD* можно закрасить красным цветом, вершину *DA* — нельзя, так как есть ребро между вершинами *BD* и *DA*. Продолжаем перебор вершин: вершину *DB* нельзя закрасить красным цветом, вершина *DC* уже закрашена синим цветом, а вершину *EA* можно закрасить красным. Остальные незакрашенные вершины имеют общие ребра с вершинами, окрашенными в красный цвет, поэтому к ним нельзя применить этот цвет.

Рассмотрим незакрашенные вершины *DA*, *DB*, *EB* и *EC*. Если вершину *DA* мы закрасим в зеленый цвет, то в этот цвет также можно закрасить и вершину *DB*, но вершины *EB* и *EC* в зеленый цвет закрасить нельзя. К последним двум вершинам применим четвертый цвет, скажем, желтый. Назначенные вершинам цвета приведены в табл. 1.3. В этой таблице "дополнительные" повороты совместимы с соответствующими поворотами из столбца "Повороты", хотя алгоритмом они окрашены в разные цвета. При задании режимов работы светофоров, основанных на одном из цветов раскраски, вполне безопасно включить в эти режимы и дополнительные повороты.

Данный алгоритм не всегда использует минимально возможное число цветов. Можно использовать результаты из общей теории графов для оценки качества полученного решения. В теории графов *k-кликой* называется множество из *k* вершин, в котором каждая пара вершин соединена ребром. Очевидно, что для закраски *k*-клики необходимо *k* цветов, поскольку в клике никакие две вершины не могут иметь одинаковый цвет.

Цвет	Повороты	Дополнительные повороты
Синий	*AB, AC, AD, BA, DC, ED*	—
Красный	*BC, BD, EA*	*BA, DC, ED*
Зеленый	*DA,DB*	*AD, BA, DC, ED*
Желтый	*EB, EC*	*BA, DC, EA, ED*

В графе рис. 1.3 множество из четырех вершин *AC, DA, BD* и *EB* является 4-кликой. Поэтому не существует раскраски этого графа тремя и менее цветами, и, следовательно, решение, представленное в табл. 1.3, является оптимальным в том смысле, что использует минимально возможное количество цветов для раскраски графа. В терминах исходной задачи это означает, что для управления перекрестком, показанным на рис. 1.1, необходимо не менее четырех режимов работы светофоров.

Итак, для управления перекрестком построены четыре режима работы светофорами, которые соответствуют четырем цветам раскраски графа (табл. 1.3). □

1.8.2. Псевдоязык и пошаговая „кристаллизация" алгоритмов

Поскольку для решения исходной задачи мы применяем некую математическую модель, то тем самым можно формализовать алгоритм решения в терминах этой модели. В начальных версиях алгоритма часто применяются обобщенные операторы, которые затем переопределяются в виде более мелких, четко определенных инструкций. Например, при описании рассмотренного выше алгоритма раскраски графа мы использолали такие выражения, как "выбрать произвольную незакрашенную вершину". Мы надеемся, что читателю совершенно ясно, как надо понимать такие инструкции. Но для преобразования таких неформальных алгоритмов в компьютерные программы необходимо пройти через несколько этапов формализации (этот процесс можно назвать *пошаговой кристаллизацией),* пока мы не получим программу, полностью состоящую из формальных операторов языка программирования.

Пример 1.10. Рассмотрим процесс преобразования "жадного" алгоритма раскраски графа в программу на языке Pascal. Мы предполагаем, что есть граф G, вершины которого необходимо раскрасить. Программа *greedy* (жадный) определяет множество вершин, названное *newclr* (новый цвет), все вершины которого можно окрасить в новый цвет. Эта программа будет вызываться на повторное выполнение столько раз, сколько необходимо для закраски всех вершин исходного графа. В самом первом грубом приближении программа *greedy* на псевдоязыке показана в следующем листинге.

Листинг 1.2. Первое приближение программы *greedy*

```
procedure greedy { var G: GRAPH; var newclr: SET );
{ greedy присваивает переменной newclr множество вершин
      графа G, которые можно окрасить в один цвет}
begin
```

```
(1)   newclr:=∅[1];
(2)   for для каждой незакрашенной вершины v из G do
(3)   if v не соединена с вершинами из newclr then begin
(4)         пометить v цветом;
(5)         добавить v в newclr
      end
end; { greedy }
```

В листинге 1.2 вы легко выделите средства нашего псевдоязыка. Мы применяем полужирное начертание для зарезервированных ключевых слов языка Pascal, которые имеют тот же смысл, что и в стандартном языке Pascal. Написание строчными буквами таких слов, как GRAPH (Граф) и SET[1] (Множество), указывает на имена *абстрактных типов данных*. Их можно определить с помощью объявления типов языка Pascal и операторов, соответствующих абстрактным типам данных, которые задаются посредством процедур языка Pascal (эти процедуры должны входить в окончательный вариант программы). Более детально абстрактные типы данных мы рассмотрим в следующих двух разделах этой главы.

Управляющие конструкции языка Pascal, такие как **if, for** и **while,** могут применяться в операторах псевдоязыка, но условные выражения в них (как в строке (3)) могут быть неформальными, в отличие от строгих логических выражений Pascal. Также и в строке (1) в правой части оператора присваивания применяется неформальный символ. Еще отметим, что оператор цикла **for** в строке (2) проводит повторные вычисления по элементам множества.

Чтобы стать исполняемой, программа на псевдоязыке должна быть преобразована в программу на языке Pascal. В данном случае мы не будем приводить все этапы такого преобразования, а покажем только пример преобразования оператора **if** (строка (3)) в более традиционный код.

Чтобы определить, имеет ли вершина *v* соединение с какой-либо вершиной из *newclr*, мы рассмотрим каждый элемент *w* из *newclr* и по графу *G* проверим, существует ли ребро между вершинами *v* и *w*. Для этого используем новую булеву переменную *found* (поиск), которая будет принимать значение true (истина), если такое ребро существует. Листинг 1.3 показывает частично преобразованную программу листинга 1.2.

Листинг 1.3. Частично преобразованная программа *greedy*

```
procedure greedy  { var G: GRAPH; var newclr:  SET ) ;
begin
(1)   newclr:= ∅;
(2)   for для каждой незакрашенной вершины v из G do begin
(3.1)       found:= false;
(3.2)       for для каждой вершины w из newclr  do
(3.3)             if существует ребро между v и w  then
(3.4)                   found: = true;
(3.5)       if found = false then begin
                  { v не соединена ни с одной вершиной из newclr }
(4)                     пометить  v цветом;
```

```
(5)                    добавить v в newclr
            end
        end
    end;  { greedy }
```

Обратите внимание, что наш алгоритм работает с двумя множествами вершин. Внешний цикл (строки (2) - (5)) выполняется над множеством незакрашенных вершин графа *G*. Внутренний цикл (строки (3.2) - (3.4)) работает с текущим множеством вершин *newclr*. Оператор строки (5) добавляет новые вершины в это множество.

Существуют различные способы представления множеств в языках программирования. В главах 4 и 5 мы изучим несколько таких представлений. В этом примере мы можем представить каждое множество вершин посредством абстрактного типа LIST (Список), который можно выполнить в виде обычного списка целых чисел, ограниченного специальным значением *null* (для обозначения которого мы будем использовать число 0). Эти целые числа могут храниться, например, в массиве, но есть и другие способы представления данных типа LIST, которые мы рассмотрим в главе 2.

Теперь можно записать оператор for в строке (3.2) в виде стандартного цикла по условию, где переменная *w* инициализируется как первый элемент списка *newclr* и затем при каждом выполнении цикла принимает значение следующего элемента из *newclr*. Мы также преобразуем оператор for строки (2) листинга 1.2. Измененная процедура *greedy* представлена в листинге 1.4. В этом листинге есть еще операторы, которые необходимо преобразовать в стандартный код языка Pascal, но мы пока ограничимся сделанным. □

Листинг 1.4. Измененная программа *greedy*

```
procedure greedy ( var G:  GRAPH; var newclr:  LIST );
    { greedy присваивает переменной newclr  множество вершин
            графа G,  которые можно окрасить в один цвет }
var
found:  boolean;
v, w:  integer;
begin
    newclr:= ∅;
    v:= первая незакрашенная вершина из G;
    while v <> null  do begin
        found:= false;
        w:= первая вершина из newclr
        while w <> null  do begin
          if существует ребро между v и w  then
              found:= true;
          w:= следующая вершина из
      newclr;
        end;
        if found = false then begin
          пометить v цветом;
```

```
            добавить v в newclr
        end
        v:= следующая незакрашенная вершина из
    G;
    end;
end; { greedy }
```

Задание

1. Вычислить время выполнения программ
2. Разработать программы для автоматизация вычисления времени выполнения программ
3. Разработать методом пошаговой детализации алгоритм и программу для решения задачи. Оценить временную эффективность полученного алгоритма

1. В состязаниях футбольной лиги участвуют шесть команд: Соколы, Львы, Орлы, Бобры, Тигры и Скунсы. Соколы уже сыграли с Львами и Орлами. Львы также сыграли с Бобрами и Скунсами. Тигры сыграли с Орлами и Скунсами. Каждая команда играет одну игру в неделю. Найдите расписание матчей, чтобы все команды сыграли по одному разу друг с другом в течение минимального количества недель. *Совет.* Создайте граф, где вершины будут соответствовать парам команд, которые еще не играли между собой. Что должны обозначать ребра в таком графе, если при правильной раскраске графа каждый цвет соответствует матчам, сыгранным в течение определенной недели?

2. Рассмотрим руку робота с закрепленным одним концом. Рука состоит из двух "колен", каждое из которых может поворачивать руку на 90 градусов вверх и вниз в вертикальной плоскости. Какую математическую модель следует применить для описания перемещения оконечности руки? Разработайте алгоритм для перемещения оконечности руки из одного возможного положения в другое.

3. Предположим, что необходимо перемножить четыре матрицы действительных чисел $M_1 x M_2 x M_3 x M_4$, где Mi имеет размер 10x20, размер M_2 составляет 20x50, M_3 — 50x1 и M_4 — 1x100. Предположим, что для перемножение двух матриц размером pxq и qxr требуется pqr скалярных операций, это условие выполняется в обычных алгоритмах перемножения матриц. Найдите оптимальный порядок перемножения матриц, который минимизирует общее число скалярных операций. Как найти этот оптимальный порядок в случае произвольного числа матриц?

4. Предположим, что мы хотим разделить множество значений квадратных корней целых чисел от 1 до 100 на два подмножества так, чтобы суммы чисел обоих подмножества были по возможности максимально близки. Если мы имеем всего две минуты машинного времени для решения этой задачи, то какие вычисления необходимо выполнить?

5. Опишите "жадный" алгоритм для игры в шахматы. Каковы, по вашему мнению, его достоинства и недостатки?

6. *Наибольшим общим делителем* двух целых чисел p и q называется наибольшее целое число d, которое делит p и q нацело. Мы хотим создать программу для вычисления наибольшего общего делителя двух целых чисел p и q, используя следующий алгоритм. Обозначим через r остаток от деления p на q. Если r равно 0, то q является наибольшим общим делителем. В противном случае положим p равным q, затем — q равным r и повторим процесс нахождения остатка от деления p на q.

a. покажите, что этот алгоритм правильно находит наибольший общий делитель;

b. запишите алгоритм в виде программы на псевдоязыке;

c. преобразуйте программу на псевдоязыке в программу на языке Pascal.

7. Мы хотим создать программу форматирования текста, которая выравнивала бы текст по ширине строк. Программа должна иметь буфер слов и буфер строки. Первоначально оба буфера пусты. Очередное слово считывается в буфер слов. Если в буфере строки достаточно места, то слово переносится в буфер строки. В противном случае добавляются пробелы между словами до полного заполнения буфера строки, затем после печати строки этот буфер освобождается.

a. запишите алгоритм в виде программы на псевдоязыке;

b. преобразуйте программу на псевдоязыке в программу на языке Pascal.

8. Рассмотрим множество из n городов и таблицу расстояний между ними. Напишите программу на псевдоязыке для нахождения кратчайшего пути, который предусматривает посещение каждого города только один раз и возвращается в тот город, откуда начался путь. На сегодняшний день единственным методом точного решения этой задачи является только метод полного перебора всех возможных вариантов. Попытайтесь построить эффективный алгоритм для решения этой задачи, используя эвристический подход.

9. Рассмотрим следующие функции от n:

$$f_1 = n^2;$$

$$f_2(n) = n^2 + 1000n;$$

$$f_3(n) = \begin{cases} n, & \text{если } n \text{ нечетно,} \\ n^3, & \text{если } n \text{ четно.} \end{cases}$$

$$f_4(n) = \begin{cases} n, & \text{если } n \le 100, \\ n^3, & \text{если } n > 100. \end{cases}$$

Укажите для каждой пары функций, когда $f_i(n)$ имеет порядок $O(f_j(n))$ и когда $fi(n)$ есть $\Omega(f_j(n))$.

10. Рассмотрим следующие функции от n:

$$g_1(n) = \begin{cases} n^2 \text{ для четных } n \geq 0, \\ n^3 \text{ для нечетных } n \geq 1; \end{cases}$$

$$g_2(n) = \begin{cases} n \text{ для } 0 \leq n \leq 100, \\ n^3 \text{ для } n > 100; \end{cases}$$

$$g_3(n) = n^{2.5}.$$

Укажите для каждой пары функций, когда $g_i(n)$ имеет порядок $O(g_i(n))$ и когда $g_i(n)$ есть $\Omega(g_i(n))$.

11. Найдите, используя О-символику, время выполнения в наихудшем случае следующих процедур как функции от n:

а)
```
procedure matmpy ( n: integer);
    var
        i,j, k: integer;
    begin
        for i:= 1 to n do
                for j:= 1 to n do
                    begin
                    C [ i , j ] := 0;
                    for k:=1 to n do
                        C [ i , j ] := C[i,j]+ A[i,k]*B[k,j]
                    end
    end
```

б)
```
procedure mystery (n: integer);
    var
        i, j, k: integer;
    begin
    for i:=1 to n-1 do
            for j:= i+1 to n
      do
                    for k:= 1 to j do
                      {группа операторов с временем выполнения 0(1)}
    end
```

в)
```
procedure veryodd(n:integer);
    var
        i,j,x,y:integer ;
    begin
          for  i:=  1  to  n do
               if нечетное(i) then
                    begin
                  for j :=  1  to  n
                    do
                        x:=x+1;
                  for j:= 1 to i do
```

```
                    y:=y+1
                end
            end
```

*г) **procedure** *recursive* (*n*:integer):
integer;

```
            begin
                if n<=1 then
                    r
                eturn(
                1) else
                    return(recursive(n-1)+recursive(n -1))
            end
```

12. Докажите, что следующие утверждения истинны:

а) 17 имеет порядок $O(1)$;

б) $n(n - 1)/2$ имеет порядок $O(n^2)$;

в) $\max(n^3, 10n^2)$ имеет порядок $O(n^3)$;

г) $\sum_{k=1}^{n} i^k$ есть $O(n^{k+1})$ и $\Omega(n^{k+1})$ для целых k;

д) если $p(x)$ — полином степени κ с положительным старшим коэффициентом,
то $p(x)$ есть $O(n^k)$ и $\Omega(n^k)$.

13. Предположим, что $T_1(n)$ есть $\Omega(f(n))$ и $T_2(n)$ — $\Omega(g(n))$. Какие из следующих утверждений истинны?

а) $T_1(n) + T_2(n)$ есть $\Omega(\max(f(n), g(n)))$;

б) $T_1(n)T_2(n)$ есть $\Omega(f(n)\, g(n))$.

14. Некоторые авторы определяют нижний порядок роста O, следующим образом: $f(n)$ есть $\Omega(g(n))$, если существуют такие неотрицательные константы n_0 и c, что для всех $n \geq n_0$ выполняется неравенство $f(n) \geq cg(n)$.

а) В рамках этого определения будет ли истинным следующее утверждение: $f(n)$ есть $\Omega(g(n))$ тогда и только тогда, когда $g(n)$ имеет порядок $O(f(n))$?

б) Будет ли утверждение а) истинным для определения Ω, из раздела 1.4?

в) Выполните упражнение 1.14 для данного определения Ω.

15. Расположите следующие функции в порядке степени роста:

а) n,

б) $\sqrt{n}$,

в) $\log n$,

г) $\log \log n$,

д) $\log^2 n$,

е) $n / \log n$,

ж) $\sqrt{n} \log^2 n$,

з) $(1/3)^n$, и) $(3/2)^n$,

к) 17.

16. Предположим, что параметр n приведенной ниже процедуры является

положительной целой степенью числа 2, т.е. n = 2, 4, 8, 16, ... Найдите формулу для вычисления значения переменной *count* в зависимости от значения n.

```
procedure mystery(n: integer ); var
        x,count: integer; begin
        count: = 0; x:=2 ;
        while x < n do begin x:=2*x;
              count:=count+1 end;
        writeln(count) end
```

17. Рассмотрим функцию *max(i, n)*, которая возвращает наибольший из элементов, стоящих в позициях от i до $i + n - 1$ в целочисленном массиве A. Также предположим, что n является степенью числа 2.

```
procedure max (i,n: integer): integer;
    var
        ml, m2: integer;
    begin
        if n = 1 then
              return(A[i])
        else begin
              ml:=max(i,n div 2);
              m2:= max(i+n div 2,n div 2 );
              if ml<m2 then
                    return (m2)
              else
              return
    (ml)
              end
    end
```

а) Пусть *T(n)* обозначает время выполнения в наихудшем случае программы *max* в зависимости от второго аргумента n. Таким образом, n — число элементов, среди которых ищется наибольший. Запишите рекуррентное соотношение для T(n), в котором должны присутствовать *T(j)* для одного или нескольких значений j, меньших n, и одна или несколько констант, представляющих время выполнения отдельных операторов программы *max*.

б) Запишите в терминах O-большое и Ω верхнюю и нижнюю границы для *T(n)* в максимально простом виде.

18. Расположите следующие функции в порядке возрастания. Если некоторые из них растут с одинаковой скоростью, то объедините

2^n	$\log_2 \log_2 n$	$n^3 + \log_2 n$
$\log_2 n$	$n - n^2 + 5n^3$	2^{n-1}
n^2	n^3	$n \log_2 n$
$(\log_2 n)^2$	$\sqrt{n}$	6
$n!$	n	$(3/2)^n$

19. Для каждой из приведенных ниже пар функций f и g выполняется одно из равенств: либо $f = O(g)$, либо $g = O(f)$, но не оба сразу. Определите, какой из случаев имеет место.

а) $f(n) = (n^2 - n)/2, \quad g(n) = 6n$

б) $f(n) = n + 2\sqrt{n}, \quad g(n) = n^2$

в) $f(n) = n + n\log_2 n, \quad g(n) = n\sqrt{n}$

г) $f(n) = n^2 + 3n + 4, \quad g(n) = n^3$

д) $f(n) = n\log_2 n, \quad g(n) = n\sqrt{n}/2$

е) $f(n) = n + \log_2 n, \quad g(n) = \sqrt{n}$

ж) $f(n) = 2\log_2 n^2, \quad g(n) = \log_2 n + 1$

з) $f(n) = 4n\log_2 n + n, \quad g(n) = (n^2 - n)/2$

Разработать методом пошаговой детализации алгоритм и программу для решения задачи. Оценить временную эффективность полученного алгоритма.

1. Натуральное число в p-ичной системе счисления задано своими цифрами, хранящимися в массиве *K(n)*. Проверить корректность такого представления и перевести число в q-ичную систему (возможно, число слишком велико, чтобы получить его внутреннее представление; кроме того, $p \le 10$, $q \le 10$).

2. Для натуральных чисел, не превосходящих заданного k, проверить признак делимости на 9 (сумма цифр числа, делящегося на 9, также делится на 9). Распечатать m последних таких чисел ($m << k$).

3. Число делится на 11, если разность между суммой цифр, стоящих на нечетных местах, и суммой цифр, стоящих на четных местах, кратна 11. Проверить этот признак для всех натуральных чисел, не превосходящих заданного m, и вывести числа, кратные 11.

4. В массиве *A(m)* хранятся различные вещественные числа (как большие, так и меньшие единицы). Округлить их, оставив в каждом по 3 значащих цифры.

5. Для заданного m получить таблицу первых m простых чисел.

6. Своими цифрами в массивах *K(m)* и *L(n)* заданы два целых числа в *p*-ичной системе счисления ($p \le 10$). Найти в таком же виде их сумму, не вычисляя самих чисел.

7. Возвести заданное вещественное число *a* в целую степень *k*, не пользуясь операцией возведения в степень и не производя (*k* -1) умножений и многократного сложения (так как *k* велико). **Рекомендация.** Сокращение числа умножений может быть достигнуто применением «индийского алгоритма» — по рекуррентной формуле:

$$x^n = \begin{cases} x, \text{если} \quad n = 1; \\ x^{n \bmod 2} \, (x^{n \operatorname{div} 2})^2, \text{если} \quad n > 1 \end{cases}$$

8. Среди заданных натуральных чисел найти такие, десятичная запись которых не содержит одинаковых цифр.

9. Найти все натуральные числа, не превосходящие заданного *m*, сумма цифр в десятичном представлении каждого из которых равна заданному *k*.

10. Пусть *m* натуральных чисел заданы своими цифрами в q-ичной системе счисления, хранящимися в строках матрицы *K(m,n)*. Найти сумму этих чисел в той же системе, не вычисляя самих чисел ($q \le 10$).

11. Осуществить циклический сдвиг n-разрядного двоичного представления заданного числа *k* на *m* позиций вправо, не находя цифр самого двоичного представления и не пользуясь стандартной процедурой сдвига.

12. В массивах *K(n)* и *L(n)* заданы соответственно числители и знаменатели рациональных чисел вида $x_i = k_i / l_i$, $i = 1, 2, \ldots, n$. Найти наибольшее из этих чисел, не пользуясь операцией деления.

13. Напечатать таблицу сложения одноразрядных чисел в p-ичной системе счисления, $p \le 16$.

14. В заданном вещественном массиве *A(n)* найти все числа, у которых старшая значащая десятичная цифра есть 9 (числа сильно различаются по величине).

15. Найти все натуральные числа, не превосходящие заданного *m* и содержащие хотя бы одну девятку в десятичном представлении.

16. Перевести заданное целое число в систему римского счета.

Указание. Римские цифры обозначаются:

1	I	500	D
5	V	1 000	M
10	X	5 000	$\bar{V}$
50	L	10 000	$\bar{X}$
100	C	...	...

17. Найти минимальное натуральное *m* такое, что: $m = k^3 + l^3 = i^3 + j^3$ (k, l, i, j — различные натуральные числа).

18. Десятичное представление заданного натурального числа напечатать *вразрядку*, то есть вставить пробелы между цифрами.

19. Напечатать столбиком пример на умножение в десятичной системе счисления двух заданных натуральных чисел k и l.

20. Напечатать столбиком пример наделение с остатком двух заданных натуральных чисел k и l.

21. Произведение двух заданных натуральных чисел больше максимально допустимого значения (не вмещается в разрядную сетку машины). Найти это произведение.

22. Найти 20 первых троек *пифагоровых* чисел, то есть целых k, l, m таких, что $k^2 + l^2 = m^2$.

23. Любая целочисленная денежная сумма s > 7 р. может быть выдана без сдачи «трешками» и «пятерками». Найти для заданной суммы s необходимое количество «трешек» и «пятерок».

24. Найти первые m более чем 2-разрядных *чисел-палиндромов,* то есть чисел, десятичная запись которых читается одинаково в прямом и обратном направлениях, например: 373, 426 624.

25. Найти все натуральные числа, не превосходящие заданного n и делящиеся на каждую из своих цифр (в десятичной системе счисления).

26. Найти все натуральные числа, не превосходящие заданного n, десятичная запись которых есть строго возрастающая или строго убывающая последовательность цифр.

27. Каждое из заданных натуральных чисел заменить числом, получающимся при записи его десятичных цифр в обратном порядке.

28. Заданное натуральное число n, не превосходящее 1000, записать прописью, то есть вывести соответствующее количественное числительное, например: 375 — «триста семьдесят пять».

29. Найти все натуральные числа, не превосходящие заданного m, двоичная запись которых представляет собой симметричную последовательность нулей и единиц (начинающуюся с единицы). Показать десятичную и двоичную записи этих чисел.

Дробная часть бесконечной десятичной дроби хранится в массиве цифр $K(l)$ (без округления). Проверить, не является ли эта дробь периодической, и если это так, превратить ее в правильную дробь вида m / n.

Лабораторная работа - 2.

Тема: Программная реализация АТД «Список», «Стек», «Очередь», «Отображение», рекурсивных процедур. Программная реализация деревьев.

Цель работы: изучение абстрактных типов данных и способов их реализации средствами языков программирования.

В результате выполнения лабораторной работы студенты должны:

- *знать* понятия типов данных, структур данных и их классификацию; определение абстрактных типов данных «Список», «Стек», «Очередь», «Отображение», «Дерево» и способы их реализации;
- *уметь* писать и отлаживать программные модули, реализующие вышеперечисленные АТД.

2.1. Классификация типов данных

Типы и структуры данных представляют собой фундамент, на котором строится вся современная технология программирования.

Тип данных можно определить путем задания множества значений данных, принадлежащих данному типу, и операций, определенных для данного множества.

Пусть D множество, состоящее из всех значений данных, разрешенных в языке программирования L. Если каждому элементу d множества D соответствует определенный тип данных, то множество D можно представить как

$$D = D_{t_1} \cup D_{t_2} \cup \ldots$$

причем если $t \neq r$, то $D_t \cap D_r = \varnothing$, где $T = \{t_1, t_2, \ldots\}$, представляет множество имен типов данных, разрешенных в языке L, а

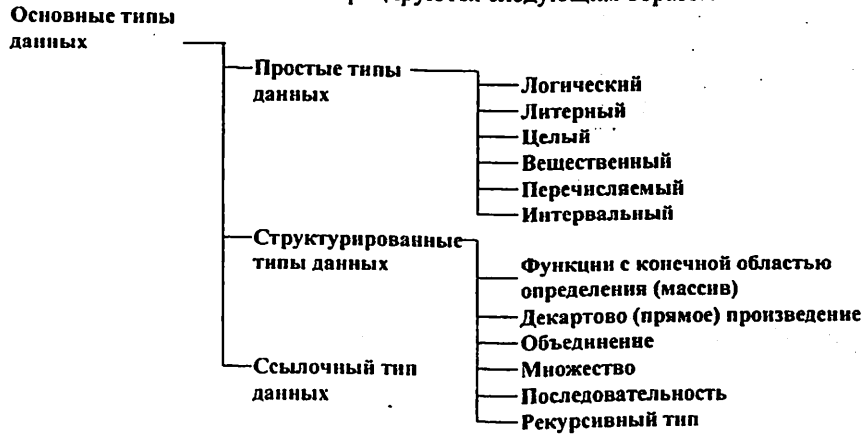
$$D_t = \{d \mid d \in D u typ(d) = t\}$$

есть множество данных типа t.

Пусть F множество операций языка L. Каждый элемент f множества F, как правило, является функцией вида

$$f : D_{t_1} \times D_{t_2} \times \ldots \times D_{t_n} \rightarrow D_{r_1} \times D_{r_2} \times \ldots \times D_{r_m}.$$

Основные типы данных классифицируются следующим образом:



2.2. Простые типы.

Логический тип данных именуется как *boolean*, т. е.

$$D_{boolean} = \{false, true\} \text{ и } false < true .$$

В качестве операций над логическим типом используются

and, or: $D_{boolean} \times D_{boolean} \rightarrow D_{boolea}$

not: $D_{boolean} \rightarrow D_{boolea}$

Литерный тип предназначен для образования текстов, используемых для общения людей, состоит из литер и обозначается *character*. В языке Паскаль существуют две стандартные функции *ord* и *chr*, называемые *функциями преобразования*, которые позволяют отображать множество литер на подмножество натуральных чисел и наоборот:

ord: $D_{character} \rightarrow D_{integer}$

chr: $D_{integer} \rightarrow D_{characte}$

Функция *ord(c)* определяет порядковый номер литеры *c* из упорядоченного набора литер, заданных $D_{character}$, а *chr(i)*, наоборот, определяет литеру, порядковый номер которой равен *i*. Очевидно, что справедливы следующие соотношения:

$$ord(chr(i)) = i,$$
$$chr(ord(c)) = c.$$

Лабораторная работа - 2.

Тема: Программная реализация АТД «Список», «Стек», «Очередь», «Отображение», рекурсивных процедур. Программная реализация деревьев.

Цель работы: изучение абстрактных типов данных и способов их реализации средствами языков программирования.

В результате выполнения лабораторной работы студенты должны:

- *знать* понятия типов данных, структур данных и их классификацию; определение абстрактных типов данных «Список», «Стек», «Очередь», «Отображение», «Дерево» и способы их реализации;
- *уметь* писать и отлаживать программные модули, реализующие вышеперечисленные АТД.

2.1. Классификация типов данных

Типы и структуры данных представляют собой фундамент, на котором строится вся современная технология программирования.

Тип данных можно определить путем задания множества значений данных, принадлежащих данному типу, и операций, определенных для данного множества.

Пусть D множество, состоящее из всех значений данных, разрешенных в языке программирования L. Если каждому элементу d множества D соответствует определенный тип данных, то множество D можно представить как

$$D = D_{t_1} \cup D_{t_2} \cup \ldots$$

причем если $t \neq r$, то $D_t \cap D_r = \varnothing$, где $T = \{t_1, t_2, \ldots\}$, представляет множество имен типов данных, разрешенных в языке L, а

$$D_t = \{d \mid d \in D и typ(d) = t\}$$

есть множество данных типа t.

Пусть F множество операций языка L. Каждый элемент f множества F, как правило, является функцией вида

$$f : D_{t_1} \times D_{t_2} \times \ldots \times D_{t_n} \rightarrow D_{r_1} \times D_{r_2} \times \ldots \times D_{r_m}.$$

Основные типы данных классифицируются следующим образом:

2.2. Простые типы.

Логический тип данных именуется как *boolean*, т. е.

$$D_{boolean} = \{false, true\} \text{ и } false < true .$$

В качестве операций над логическим типом используются

and, or: $D_{boolean} \times D_{boolean} \rightarrow D_{boolea}$

not: $D_{boolean} \rightarrow D_{boolea}$

Литерный тип предназначен для образования текстов, используемых для общения людей, состоит из литер и обозначается *character*. В языке Паскаль существуют две стандартные функции *ord* и *chr*, называемые *функциями преобразования*, которые позволяют отображать множество литер на подмножество натуральных чисел и наоборот:

ord: $D_{character} \rightarrow D_{integer}$

chr: $D_{integer} \rightarrow D_{charact}$

Функция *ord(c)* определяет порядковый номер литеры *c* из упорядоченного набора литер, заданных $D_{character}$, а *chr(i)*, наоборот, определяет литеру, порядковый номер которой равен *i*. Очевидно, что справедливы следующие соотношения:

$$ord(chr(i)) = i,$$
$$chr(ord(c)) = c.$$

Целый и *вещественный* типы данных предназначены для представления числовых значений.

Перечисляемый тип обладает только свойством перечисления принадлежащих ему данных.

Для него можно определить операции:

$$succ, pred : D_T \to D_T$$

$$succ(d_i) = d_{i+1}, 1 \le i \le N-1,$$

$$pred(d_i) = d_{i-1}, 2 \le i \le N.$$

Определение *интервального* типа можно рассматривать как обозначение интервала значений любого заранее определенного типа с возможностью перечисления. Очевидно, что значения элементов должны находиться внутри заданных границ интервала.

2.3. Структурированные типы.

Функция с конечной областью определения представляет собой отображение некоторого конечного множества данных на множество данных другого типа. В традиционных языках программирования этот тип чаще называют *массивом.*

Массив с типом индексов I и типом элементов T_0, например в языке Паскаль, определяется следующим описанием:

type T = **array** [I] **of** T_0

Массив A, элементы которого имеют тип T_0, представляет собой отображение D_I на D_{T_0}:

$$A : D_I \to D_{T_0},$$

поэтому массив типа T есть множество элементов данного отображения. Это множество можно представить в следующем виде:

$$D_T = \left[D_I \to D_{T_0} \right].$$

Прямое (декартово) произведение, как и массив, является одним из основных структурированных типов данных, и его называют также *записью* или *структурой.*

Тип прямого произведения, состоящий из базисных типов $T_1, \ldots, T_n$, определяется следующим образом:

type T= **record** $s_1 : T_1;$
$s_2 : T_2;$
.....
$s_n : T_n;$
end

Здесь $s_1, s_2 \ldots, s_n$ — имена компонент; $T_1, T_2, \ldots, T_n$ — типы данных компонент. Множество данных типа T можно представить с помощью прямого (декартова) произведения множеств $D_{T_1}, D_{T_2}, \ldots D_{T_n}$:

$$D_T = D_{T_1} \times D_{T_2} \times \ldots \times D_{T_n}.$$

Объединение представляет собой множество, отдельные элементы которого классифицируются по категориям.

Объединение типов данных T_1, T_2 можно получить путем объединения соответствующих множеств данных D_{T_1}, D_{T_2} и определения допустимых операций. Если t_1, t_2 — признаки принадлежности к типам T_1 и T_2, то объединение T типов T_1 и T_2 определяется следующим образом:

type T= **union** $t_1 : T_1;$
$t_2 : T_2;$
.....
$t_n: T_n;$

end

Множество, состоящее из перечисления элементов базисного типа T_0, определяется следующим образом:

type T = **set of** T_0

Множество D значений типа T имеет следующий вид:

$$D_T = 2^{D_{T_0}} = \{S \mid S \subset D_{T_0}\}.$$

Тип T, представляющий собой *последовательность* элементов типа T_0, определяется следующим образом:

type T = **sequence of** T_0

Последовательность T определяет множество конечных последовательностей произвольной длины, состоящих из элементов типа T_0, что можно записать следующим образом:

$$D_T = \bigcup_{n=0}^{\infty} D_{T_0} n,$$

где $D_{T_0} n$ — n-й сорт D_{T_0}.

Если x — последовательность, то ее элементы можно записать следующим образом:

а) $x[i]$ — i-й элемент x;
б) $first(x)$ — начальный элемент x;
в) $last(x)$ — конечный элемент x.

Например, если $x = T(e_1, e_2, \ldots, e_n)$, то

$x[i] = e_i$, $first(x) = e_1$, $last(x) = e_n$

и представляют собой функцию следующего вида:

$$[i], first, last : D_T \rightarrow D_{T_0}.$$

Включение — исключение элементов последовательности

а) *tail(x)* — последовательность, в которой из x исключен начальный элемент x;
б) *initial(x)* — последовательность, в которой исключен последний элемент x;
в) *appendl(x,e)* — последовательность, в которую добавили e перед x (слева);
г) *appendr(x,e)* — последовательность, в которую добавили e после x (справа).
Если $x = T(e_1, e_2, ..., e_n)$, то

$$tail(x) = T(e_2, ..., e_n),$$
$$initial(x) = T(e_1, ... e_{n-1}),$$
$$appendl(x, e) = T(e, e_1, ..., e_n),$$
$$appendr(x, e) = T(e_1, ..., e_n, e).$$

Очевидно, что имеют место следующие функции:

$$tail, initial : D_T \to D_T,$$
$$appendl, appendr : D_T \times D_{T_0} \to D_T$$

Между указанными функциями существуют следующие зависимости:

$$first(appendl(x, e)) = e,$$
$$tail(appendl(x, e)) = x,$$
$$appendl(tail(x), first(x)) = x, \quad \text{если} \quad x \neq T(),$$
$$last(appendr(x, e)) = e,$$
$$initial(appendr(x, e)) = x,$$
$$appendr(initial(x), last(x)) = x, \quad \text{если} \quad x \neq T().$$

Определение пустой последовательности

$$empty(x) = \begin{cases} true, если\ x = T(), \\ false, если\ x \neq T(). \end{cases}$$

2.4. Абстрактные типы данных.

Предположим, что создается программа для решения сложной проблемы P_0. Чтобы непосредственно записать программу для решения проблемы P_0, необходимо одновременно учитывать очень большое количество взаимосвязанных факторов, что превышает возможности человека. Однако в большинстве случаев проблему P_0 можно декомпозировать на несколько подпроблем P_1, ..., P_m и сравнительно легко получить решение проблемы P_0 путем решения данных подпроблем. Если проводить дальнейшую декомпозицию каждой подпроблемы P_i до стадии, на которой легко получить ее решение, то это позволит решить основную проблему P_0. Обычно такой способ называют *способом иерархического решения проблем.*

Итак, вопрос состоит в том, каким образом, следуя описанному подходу, создавать программу для решения проблемы P_0. Можно иерархически организовать программу следующим образом. Вначале определяется абстрактный язык программирования L_0 и программа $prog(P_0)$ пишется на языке L_0. Язык L_0 должен быть снабжен типами данных (D_0, F_0) высокого уровня и обладать управляющей структурой C_0. Написанную на языке L_0 программу $prog(P_0)$ будет трудно реализовать, поскольку (D_0, F_0) или C_0

представляют собой идеальные понятия, сформулированные с целью удобства описания проблемы P_0. Подпроблемы P_i являются структурными элементами проблемы P_0. Программу на языке $L_0 = (D_0, F_0, C_0)$ выражают, используя более конкретный язык $L_1 = (D_1, F_1, C_1)$ (в общем случае длявыражения L_0 можно считать, что используются многочисленные языки $L_i = (D_i, F_i, C_i)$, где $i=1, ..., m$). Для этого необходимо, во-первых, задать данные $d_1 \in D_1$ для реализации данных $d_0 \in D_0$ и, во-вторых, разработать на языке L_1 программы *prog(f)*, *prog(c)*, реализующие операции $f \in F_0$ и структуру управления $c \in C_0$. Если повторять данную операцию и дойти до конкретного языка программирования L_e, то благодаря последовательности языков $L_0, L_1, ..., L_e$ абстрактная программа $prog(P_0)$ для решения проблемы P_0 выражается с помощью языка L_e, который обеспечивает возможность ее выполнения.

При таком иерархическом описании необходимо четко определять данные D_i , существующие на каждом L-м уровне иерархии, а также разрешенные при этом операции F_i. В этом смысле D_i и F_i естественно считать единым понятием, что говорит о важности понятия типа данных. Абстрагированием данных называют *абстрактное определение типа данных* (D_i , F_i) , а определенный таким образом тип данных — *абстрактным типом данных*.

Метод абстрагирования данных определяется способом определения типа данных (D,F). Для того чтобы определить тип данных, необходимо:

1. Определить множество значений данных $D = \{D_t \mid t \in T\}$, т. е. для каждого типа данных t определить множество значений данных D_t.

2. Определить функции, соответствующие операциям $f \in F$, т. е. указать для f область определения $D_{T_1} \times D_{T_2} \times ... \times D_{T_n}$ и область значений $D_{r_1} \times D_{r_2} \times ... \times D_{r_m}$, а также для каждого $d \in D_{T_1} \times D_{T_2} \times ... \times D_{T_n}$ задать значение $f(d)$.

Способ абстрагирования должен удовлетворять следующим критериям:

1. Формальность.
2. Конструктивность.
3. Понятность.
4. Минимальность.
5. Широкая область применимости.
6. Расширяемость.

2.5. Абстрактный тип данных „Список"

Списки являются чрезвычайно гибкой структурой, так как их легко сделать большими или меньшими, и их элементы доступны для вставки или удаления в любой позиции списка. Списки также можно объединять или разбивать на меньшие списки. Списки регулярно используются в приложениях, например в программах информационного поиска, трансляторах программных языков или при моделировании различных процессов.

Для формирования абстрактного типа данных на основе математического определения списка мы должны задать множество операторов, выполняемых над объектами типа LIST (Список).

Примем обозначения: *L* — список объектов типа elementtype, *x* — объект этого типа, *p* — позиция элемента в списке. Обычно мы понимаем позиции как множество целых положительных чисел, но на практике могут встретиться другие представления.

1. INSERT(x, p, *L*). Этот оператор вставляет объект *x* в позицию *p* в списке L, перемещая элементы от позиции *p* и далее в следующую, более высокую позицию. Если в списке *L* нет позиции *p*, то результат выполнения этого оператора не определен.

2. LOCATE(x, L). Эта функция возвращает позицию объекта *x* в списке *L*. Если в списке объект *x* встречается несколько раз, то возвращается позиция первого от начала списка объекта *x*. Если объекта *x* нет в списке L, то возвращается END(L).

3. RETRIEVE(p, *L*). Эта функция возвращает элемент, который стоит в позиции p в списке *L*. Результат не определен, если *p* = END(L) или в списке *L* нет позиции *p*. Отметим, что элементы должны быть того типа, который в принципе может возвращать функция. Однако на практике мы всегда можем изменить эту функцию так, что она будет возвращать указатель на объект типа elementtype.

4. DELETE(p, L). Этот оператор удаляет элемент в позиции *p* списка *L*. Результат не определен, если в списке *L* нет позиции *p* или *p* = END(L).

5. NEXT(p, *L*) и PREVIOUS(p, L). Эти функции возвращают соответственно следующую и предыдущую позиции от позиции *p* в списке *L*. Если *p* — последняя позиция в списке *L*, то NEXT(p, *L*) = END(L). Функция NEXT не определена, когда *p* = END(L). Функция PREVIOUS не определена, если *p* = 1. Обе функции не определены, если в списке *L* нет позиции *p*.

6. MAKENULL(L). Эта функция делает список *L* пустым и возвращает позицию END(L).

7. FIRST(L). Эта функция возвращает первую позицию в списке *L*. Если список пустой, то возвращается позиция END(L).

8. PRINTLIST(L). Печатает элементы списка *L* в порядке из расположения.

Нами приведён основной набор операций, осуществляемых над списками. В зависимости от задачи набор функций может варьироваться.

2.6. Стек.

Стек — это последовательность, для которой определены следующие операторы:

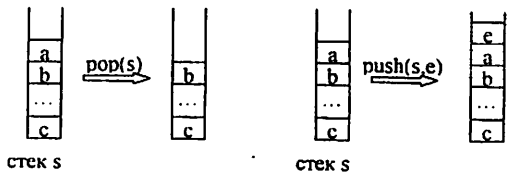
а) образование пустой последовательности *T()*;
б) *first(x)*;
в) *tail(x)*;
г) *appendl(x, e)*;
д) *empty(x)*.

Стек является наиболее широко используемым типом данных и применяется, например, при анализе языковых конструкций. Для стека добавление и извлечение элементов возможно только с одного конца последовательности. Поэтому данное, добавленное к последовательности последним, извлекается из нее первым. В этом смысле стековая память является памятью с дисциплиной обслуживания «последним вошел — первым вышел» (last in first out, LIFO). Это свойство стека эффективно применяется в случае необходимости последовательно обработать ряд элементов.

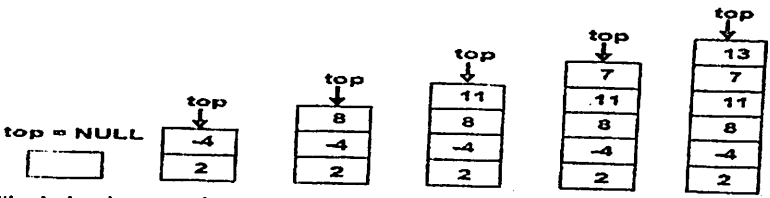
Операции *first, tail, appendl* применительно к стеку обычно называются *top, pop, push* соответственно. В языках процедурного типа стек реализуется с помощью *стековой переменной.* Данные операции предназначены для манипулирования значениями такой переменной. Если задано следующее описание переменной:

var *s*: **stack of** T_0;
e: T_0

то *top(s)* представляет собой операцию указания на самый верхний элемент *s* (обычно стек изображается вертикально, и будем считать, что добавление и извлечение элемента выполняется в его верхней части). *Pop(s)* является операцией извлечения из стека *s* самого верхнего элемента, а *push(s,e)* — операцией добавления элемента *e* к *s*. *First, tail, appendl* для левого края последовательности соответствуют операциям *top, pop, push* для верхней части стека.



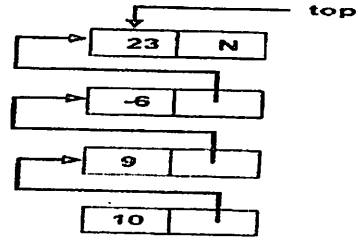
2.7. Реализация стеков с помощью массивов




```
#include <iostream.h>
const int MAX = 10 ;
class stack
{       private :
                int arr[MAX] ;
                int top ;
```

```
        public :
                stack( ) ;
                void push ( int item ) ;
                int pop( ) ;
};
stack :: stack( )
{       top = -1 ;
}
void stack :: push ( int item )
{
        if ( top == MAX - 1 )
        {       cout << endl << "Stack polniy" ;
                return ;
        }
        top++ ;
        arr[top] = item ;
}
int stack :: pop( )
{
        if ( top == -1 )
        {       cout << endl << "Stack pustoy" ;
                return NULL ;
        }
        int data = arr[top] ;
        top-- ;
        return data ;
}
```

2.8. Реализация стека с помощью списка.




```
#include <iostream.h>
class stack
{
        private :
struct node
```

```
            {       int data ;
                    node *link ;
            } *top ;
        public :
            stack( ) ;
            void push ( int item ) ;
            int pop( ) ;
            ~stack( ) ;
} ;
stack :: stack( )
{
        top = NULL ;
}
void stack :: push ( int item )
{       node *temp ;
        temp = new node ;

        if ( temp == NULL )
                cout << endl << "Stack polniy" ;
        temp -> data = item ;
        temp -> link = top ;
        top = temp ; }
int stack :: pop( )
{
        if ( top == NULL )
        {       cout << endl << "Stack pustoy" ;
                return NULL ;
        }
        node *temp ;
        int item ;
        temp = top ;
        item = temp -> data ;
        top = top -> link ;

        delete temp ;
        return item ;
}
stack :: ~stack( )
{       if ( top == NULL )
                return ;
        node *temp ;
        while ( top != NULL )
        {       temp = top ;
                top = top -> link ;
                delete temp ;
```

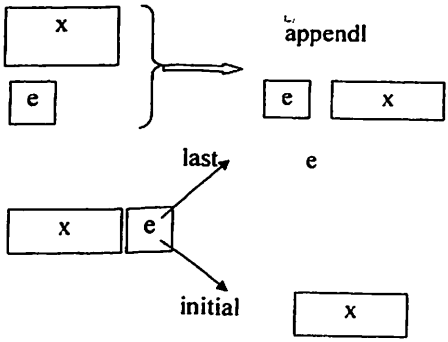
```
    }
}
```

2.9. Очереди

Очередь — это последовательность, для которой определены следующие операции:

а) образование пустой последовательности *T( )*;
б) *last(x);*
в) *initial(x);*
г) *appendl(x,e).*

Часто очередь называют *матрицей ожидания.* Добавление элемента к очереди осуществляется с ее левого конца с помощью оператора *appendl,* а извлечение элемента из очереди производится с правого конца с помощью операторов *last* и *initial.* В связи с этим очередь является памятью типа «первым вошел — первым вышел» (first in first out, FIFO). Очереди используются в случае, когда данные обрабатываются в порядке их поступления или образования.



Объявление очереди в процедурных языках осуществляется аналогично объявлению стека:

var *x* : **queue of** T_0

В качестве операторов управления очередью используются, например,

enter(x, e), leave(x,e):

$enter(x, e) \Leftrightarrow x' = appendl\ (x, e),$

$leave(x, e) \Leftrightarrow x' = initial(x),\ e' = last(x).$

Переменные *x, e* справа от знака $\Leftrightarrow$ выражают значения переменных *x* и *e* перед выполнением оператора, а x', e' выражают значения после выполнения оператора.

2.10. Реализация очередей с помощью указателей

Как и для стеков, любая реализация списков допустима для представления очередей. Однако учитывая особенность очереди (вставка новых

элементов только с одного, заднего, конца), можно реализовать оператор ENQUEUE более эффективно, чем при обычном представлении списков. Вместо перемещения списка от начала к концу каждый раз при пополнении очереди мы можем хранить указатель (или курсор) на последний элемент очереди. Как и в случае со стеками, можно хранить указатель на начало списка — для очередей этот указатель будет полезен при выполнении команд FRONT и DEQUEUE. В языке Pascal в качестве заголовка можно использовать динамическую переменную и поместить в нее указатель на начало очереди. Это позволяет удобно организовать очищение очереди.

Рассмотрим реализацию очередей с использованием указателей языка Pascal. Для начала реализации очередей с помощью массивов необходимо сделать объявление ячеек следующим образом:

```
type
    celltype = record
    element: elementtype;
    next:  ^celltype
end;
```

Теперь можно определить список, содержащий указатели на начало и конец очереди. Первой ячейкой очереди является ячейка заголовка, в которой поле *element* игнорируется. Это позволяет, как указывалось выше, упростить представление для любой очереди. Мы определяем АТД QUEUE

```
type
QUEUE = record
    front, rear:  ^celltype
end;
```

Пример: Реализация операторов очередей

```
procedure MAKENULL (var Q: QUEUE );
    begin
      new(Q.front);   { создание ячейки заголовка }
      Q.front↑.next:=nil;
      Q.rear:=Q.front
    end; { MAKENULL }
function EMPTY ( Q:QUEUE ): boolean;
    begin
        if Q.front = Q.rear  then
            return(true) else
            return(false) end; { EMPTY }
function FRONT ( Q:QUEUE ): elementtype;
    begin
        if EMPTY(Q) then
            error('Очередь пуста') else
            return (Q.front↑.next↑.element) end; { FRONT }
procedure ENQUEUE (x:elementtype; var Q:QUEUE );
    begin
```

```
        new(Q.rear↑.next); Q.rear:=Q.rear↑.next; Q.rear↑.element: = x;
        Q.rear↑.next:= nil
    end; { ENQUEUE }
procedure DEQUEUE ( var Q: QUEUE );
    begin
        if EMPTY (Q) then
            error('Очередь пуста') else
            Q.front:= Q.front↑.next
    end;  { DEQUEUE }
```

Дерево — это совокупность элементов, называемых *узлами* (при этом один из них определен как *корень*), и отношений (родительский–дочерний), образующих иерархическую структуру узлов. Узлы могут являться величинами любого простого или структурированного типа, за исключением файлового. Узлы, которые не имеют ни одного последующего узла, называются *листьями*.

Рассмотрим следующее описание типа данных:

```
type tree = union
        nterm: record
                left, right: tree
            end;
        term: integer
        end
```

Для описания типа данных *tree* применяется в свою очередь тип *tree*. При таком описании не очевиден вкладываемый смысл и наиболее естественным способом толкования является рекурсивное определение *tree*. *Tree* используется при объявлении бинарного дерева, терминальные вершины которого имеют тип *integer*. Если воспользоваться уже имеющимися знаниями об объединении и прямом произведении, то очевидно, что множество данных D_{tree} типа *tree* удовлетворяет следующему рекурсивному выражению:

$$D_{tree} = nterm : (D_{tree} \times D_{tree}) \cup term : D_{integer}$$

где

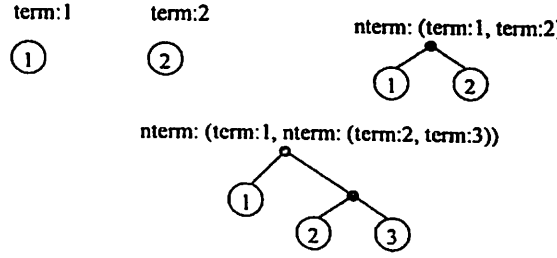
$$a : D = \{(a : d) d \in D\}$$

$$D_1 \times D_2 = \{(d_1, d_2) \mid d_1 \in D_1, d_2 \in D_2\}$$

D_{tree} можно записать следующим образом:

1. $n \in D_{integer}$, *если* $term : n \in D_{tree}$.
2. $n \in D_{tree}$, *если* $nterm : (d_1, d_2) \in D_{tree}$

Таким образом, очевидно, что D_{tree} выражает множество бинарных деревьев, как уже было сказано ранее. Если записать несколько данных, принадлежащих D_{tree}, то получим следующую диаграмму:



Необходимо обратить внимание на то, что *term* и *nterm* являются признаками, указывающими терминальность или нетерминальность узла.

Выше рассмотрен простой пример. Как правило, рекурсивный тип определяется следующим образом:

type $T_1 = \tau_1(T_1, T_2, \ldots. T_n);$

$T_2 = \tau_2(T1, T_2, \ldots, T_n);$

...

$T_n = \tau_n(T1, T_2, \ldots, T_n)$

Из этого примера очевидно, что такое описание может определять любой тип данных. В общем случае с помощью такого описания можно определить сложную древовидную структуру, но для того, чтобы это определение имело смысл, необходимо, чтобы для τ_i был определен оператор конструирования типа прямой суммы, а также необходимо обеспечить конечность древовидной структуры. В качестве примера использования рекурсивного типа данных приведем определение типа *арифметическое выражение:*

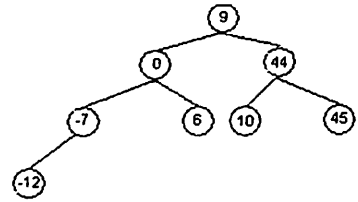
```
type expression  = union
                      a : record
                            addop : (plus, minus);
                            left : expression;
                            right : term
                            end;
                      b: term
                end;
    term  =union
                a: record
                      mulop: (mult,divide);
                      left:term;
                      right:factor
                      end;
                b:factor;
          end;
    factor=union
                a:identifier;
                b:expression
          end
```

В *двоичном (бинарном) дереве* каждый узел может быть связан не более чем двумя другими узлами. Рекурсивно двоичное дерево определяется так: *двоичное дерево* бывает либо пустым (не содержит ни одного узла), либо содержит узел, называемый *корнем*, а также два независимых поддерева — *левое поддерево* и *правое поддерево*.

Двоичное дерево поиска может быть либо пустым, либо оно обладает таким свойством, что корневой элемент имеет большее значение узла, чем любой элемент в левом поддереве, и меньшее или равное, чем элементы в правом поддереве. Указанное свойство называется *характеристическим свойством двоичного дерева* поиска и выполняется для любого узла такого дерева, включая корень. Далее будем рассматривать только двоичные деревья поиска. Такое название двоичные деревья поиска получили по той причине, что скорость поиска в них примерно такая же, что и в отсортированных массивах: $O(n) = C \cdot \log_2 n$ (в худшем случае $O(n) = n$).

Пример. Для набора данных 9, 44, 0, –7, 10, 6, –12, 45 построить двоичное дерево поиска.

Согласно определению двоичного дерева поиска число 9 помещаем в корень, все значения, меньшие его — на левое поддерево, большие или равные — на правое. В каждом поддереве очередной элемент можно рассматривать как корень и действовать по тому же алгоритму. В итоге получаем



Выделим типовые операции над двоичными деревьями поиска:

- добавление элемента в дерево;
- удаление элемента из дерева;
- обход дерева (для печати элементов и т.д.);
- поиск в дереве.

Поскольку определение двоичного дерева рекурсивно, то все указанные типовые операции могут быть реализованы в виде рекурсивных подпрограмм (на практике именно такой вариант чаще всего и применяется). Отметим лишь, что использование рекурсии замедляет работу программы и расходует лишнюю память при её выполнении.

Существует несколько способов обхода (прохождения) всех узлов дерева. Три наиболее часто используемых из них называются *обход в прямом (префиксном) порядке*, *обход в обратном (постфиксном) порядке* и *обход во внутреннем порядке (или симметричный обход)*. Каждый из обходов реализуется с использованием рекурсии.

Ниже приведены подпрограммы печати элементов дерева с использованием обхода двоичного дерева поиска в обратном порядке.

```
{Turbo Pascal}
Procedure PrintTree(T : U);
begin
   if T <> Nil
   then begin PrintTree(T^.L); write(T^.inf : 6); PrintTree(T^.R) end;
end;

// C++
void PrintTree(BinTree *T)
{
        if (T) {PrintTree(T->L); cout << T->inf<<" ";PrintTree(T->R);}
}
```

Реализуем функцию, возвращающую true (1), если элемент присутствует в дереве, и false (0) — в противном случае.

```
{Turbo Pascal}
function find(Tree : U; x : BT) : boolean;
begin
  if Tree=nil then find := false
          else if Tree^.inf=x then Find := True
                       else if x < Tree^.inf
                          then Find:=Find(Tree^.L, x)
                          else Find:=Find(Tree^.R, x)
end;

/* C++ */
int Find(BinTree *Tree, BT x)
{ if (!Tree) return 0;
 else if (Tree->inf==x) return 1;
    else if (x < Tree->inf) return Find(Tree->L, x);
          else return Find(Tree->R, x);
}
```

По сравнению с предыдущими задача удаления узла из дерева реализуется несколько сложнее. Можно выделить два случая удаления элемента *x* (случай отсутствия элемента в дереве является вырожденным):

1) узел, содержащий элемент *x*, имеет степень не более 1 (степень узла — число поддеревьев, выходящих из этого узла);

2) узел, содержащий элемент *x*, имеет степень 2.

Случай 1 не представляет сложности. Предыдущий узел соединяется либо с единственным поддеревом удаляемого узла (если степень удаляемого узла равна 1), либо не будет иметь поддерева совсем (если степень узла равна 0). Намного сложнее, если удаляемый узел имеет два поддерева. В этом случае нужно заменить удаляемый элемент самым правым элементом из его левого поддерева.

```
{Turbo Pascal}
```

```
function Delete(Tree: U; x: BT) : U;
var P, v : U;
begin
  if (Tree=nil)
  then writeln('такого элемента в дереве нет!')
  else if x < Tree^.inf then Tree^.L := Delete(Tree^.L, x){случай 1}
                  else
                   if x > Tree^.inf
                   then Tree^.R :=Delete(Tree^.R,x) {случай 1}
                   else
                   begin {случай 1}
                    P := Tree;
                    if Tree^.R=nil
                    then Tree:=Tree^.L
                    else if Tree^.L=nil
                       then Tree:=Tree^.R
                       else begin
                            v := Tree^.L;
                            while v^.R^.R<>nil do v:= v^.R;
                            Tree^.inf := v^.R^.inf;
                            P := v^.R;
                            v^.R :=v^.R^.L;
                           end;
                    dispose(P);
                   end;
Delete := Tree
end;

{C++}
BinTree * Delete(BinTree *Tree, BT x)
{ BinTree* P, *v;
 if (!Tree) cout << "такого элемента в дереве нет!" << endl;
 else if (x < Tree->inf) Tree->L = Delete(Tree->L, x);
    else if (x > Tree-> inf) Tree->R = Delete(Tree->R, x);
          else {P = Tree;
                 if (!Tree->R) Tree = Tree->L; // случай 1
                 else if (!Tree->L) Tree = Tree->R; // случай 1
                    else { v = Tree->L;
                           while (v->R->R) v = v->R; // случай 2
                           Tree->inf = v->R->inf;
                           P = v->R; v->R = v->R->L;
                          }
                 free(P);
                }
return Tree;
```

```
}
```
Примечание. Если элемент повторяется в дереве несколько раз, то удаляется только первое его вхождение.

Пример.

Реализация двоичного дерева с помощью массива

```
#include <iostream.h>
class btree
{
        private :
                struct node
                {
                        node *left ;
                        char data ;
                        node *right ;
                } *root ;
                char *arr ;
                int *lc ;
                int *rc ;

        public :
                btree ( char *a, int *l, int *r, int size ) ;
        void insert ( int index ) ;
                static node* buildtree ( char *a, int *l, int *r, int index ) ;
                void display( ) ;
                static void inorder ( node *sr ) ;
                ~btree( ) ;
                static void del ( node *sr ) ;
} ;
btree :: btree ( char *a, int *l, int *r, int size )
{
        root = NULL ;
        arr = new char[size] ;
        lc = new int[size] ;
        rc = new int[size] ;
        for ( int i = 0 ; i < size ; i++ )
        {
                * ( arr + i ) = * ( a + i ) ;
                * ( lc + i ) = * ( l + i ) ;
                * ( rc + i ) = * ( r + i ) ;
        }
}
void btree :: insert ( int index )
{
        root = buildtree ( arr, lc, rc, index ) ;
}
```

```
node* btree :: buildtree ( char *a, int *l, int *r, int index )
{
        node *temp = NULL ;
        if ( index != -1 )
        {
                temp = new node ;
                temp -> left = buildtree ( a, l, r, * ( l + index ) ) ;
                temp -> data = * ( a + index ) ;
                temp -> right = buildtree ( a, l, r, * ( r + index ) ) ;
        }
        return temp ;
}
void btree :: display( )
{
        inorder ( root ) ;
}
void btree :: inorder ( node *sr )
{
        if ( sr != NULL )
        {       inorder ( sr -> left ) ;
                cout << sr -> data << "\t" ;
                inorder ( sr -> right ) ;
        }
}
btree :: ~btree( )
{       delete arr ;
        delete lc ;
        delete rc ;
        del ( root ) ;
}
void btree :: del ( node *sr )
{
        if ( sr != NULL )
        {       del ( sr -> left ) ;
                del ( sr -> right ) ;
        }
        delete sr ;
}
```

Задание

Написать программный модуль или класс, реализующий заданный абстрактный тип данных, и программу, демонстрирующую возможности этого типа данных.

1. Односвязный список, реализующий функции вставки, удаления, поиска элемента по ключу, поиска элемента по номеру в списке, очистки списка, а

также возвращающие первый, последний, предшествующий данному и следующий за элементом с заданным номером и вывод элементов списка. Реализовать список при помощи массива.

2. Односвязный список с теми же функциями, реализовать при помощи указателей.

3. Двусвязный список отличается тем, что каждый его элемент ссылается как на следующий за ним, так и на предыдущий элемент. Реализовать при помощи массива двусвязный список, реализующий функции вставки, удаления, поиска, нахождения длины списка, вывода в прямом и обратном порядке, а также возвращающие первый, последний, предшествующий данному и следующий за данным элемент.

4. Реализовать двусвязный список, используя указатели.

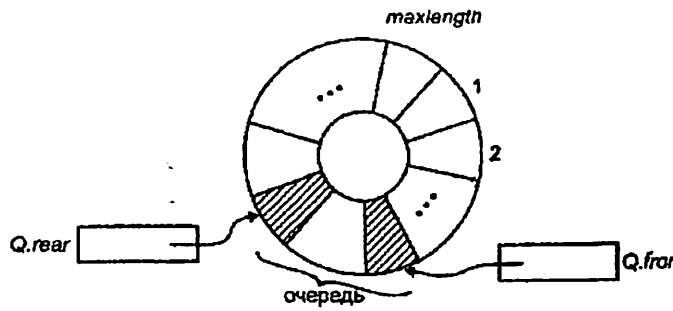
5. Реализовать стек на основе использования массива.

6. Реализовать стек на основе использования указателей.

7. Реализовать стек, используя АТД «Список» (функции АТД «Список» можно реализовать только в том объёме, который будет необходим для реализации стека).

8. Реализовать очередь на основе использования массива.

9. Реализовать очередь на основе использования циклического массива (см. рисунок).



10. Реализовать очередь с использованием указателей.

11. Реализовать очередь на основе односвязного списка.

12. Реализовать очередь на основе двусвязного списка.

13. *Отображение* — это функция, определенная на множестве элементов (области определения) одного типа (будем обозначать его domaintype — тип области определения функции) и принимающая значения из множества элементов (области значений) другого типа, этот тип обозначим rangetype — тип области значений (конечно, типы domaintype и rangetype могут совпадать). Реализовать на основе массива отображение с функциями MAKENULL - очистить, ASSIGN(M,d,r) – присвоить M(d) значение r, COMPUTE(*M, d, r*) – возвращает значение true и присваивает переменной *r* значение *M(d)*, если последнее определено, и возвращает false в противном случае.

14. Реализовать отображение на основе указателей.

15. Реализовать отображение на основе списка.

16.Реализовать на основе массивов дерево со следующими функциями

• PARENT(n, T). Эта функция возвращает родителя (parent) узла n в дереве T. Если n является корнем, который не имеет родителя, то в этом случае возвращается Λ. Здесь Λ обозначает "нулевой узел" и указывает на то, что мы выходим за пределы дерева.

• LEFTMOST_CHILD(n, T). Данная функция возвращает самого левого сына узла n в дереве T. Если n является листом (и поэтому не имеет сына), то возвращается Λ.

• RIGHT_SIBLING(n, T). Эта функция возвращает правого брата узла n в дереве T и значение Λ, если такового не существует. Для этого находится родитель p узла n и все сыновья узла p, затем среди этих сыновей находится узел, расположенный непосредственно справа от узла n. LABEL(n, T). Возвращает метку узла n дерева T. Для выполнения этой функции требуется, чтобы на узлах дерева были определены метки.

• CREATEi(v, T_1 , T_2,..., T_i) — это обширное семейство "созидающих" функций, которые для каждого i = 0, 1, 2, ... создают новый корень r с меткой v и далее для этого корня создает i сыновей, которые становятся корнями поддеревьев T_1, T_2, ..., T_i. Эти функции возвращают дерево с корнем r. Отметим, что если i=0, то возвращается один узел r, который одновременно является и корнем, и листом.

• ROOT(T) возвращает узел, являющимся корнем дерева T. Если T — пустое дерево, то возвращается Λ.

• MAKENULL(T). Этот оператор делает дерево T пустым деревом.

• PRINT(T) – вывод узлов дерева при прямом обходе.

17.Реализовать на основе указателей дерево, как в задаче 1. Функция Print выводит узлы в порядке обратного обхода.

18.Реализовать дерево с использованием списков сыновей. Функция Print должна выводить узлы в порядке симметричного обхода.

19.Реализовать двоичное дерево с использованием массива.

20.Реализовать двоичное дерево с использованием указателей.

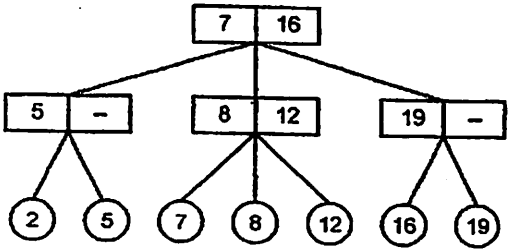
21.Реализовать двоичное дерево с использованием списков.

22.Реализовать двоичное дерево с функцией поиска заданного элемента. Функция должна выводить номера элементов, проходимых при поиске.

23.*2-3 дерево* и имеет следующие свойства.

- Каждый внутренний узел имеет или два, или три сына.
- Все пути от корня до любого листа имеют одинаковую длину.

В каждый внутренний узел записываются ключ наименьшего элемента, являющегося потомком второго сына, и ключ наименьшего элемента — потомка третьего сына, если, конечно, есть третий сын



Реализовать АТД для 2-3 дерева, обеспечивающий добавление, удаление элемента и поиск по ключу, а также вывод элементов-листьев в порядке справа налево.

24. Структура нагруженных деревьев поддерживает операторы множеств, у которых элементы являются словами, т.е. символьными строками. В нагруженном дереве каждый путь от корня к листу соответствует одному слову из множества. При таком подходе узлы дерева соответствуют префиксам слов множества. Мы можем рассматривать узел нагруженного дерева как отображение, где областью определения будет множество {A, B, ..., Z, $} (или другой выбранный алфавит), а множеством значений — множество элементов типа "указатель на узел нагруженного дерева". Более того, так как дерево можно идентифицировать посредством его корня, то АТД TRIE (Нагруженное дерево) и TRIENODE (Узел нагруженного дерева) имеют один и тот же тип данных, хотя операторы, используемые в рамках этих АТД, имеют существенные различия. Для реализации АТД TRIENODE необходимы следующие операторы.

- Процедура ASSIGN(*node, c, p),* которая задает значение *p* (указатель на узел) символу *c* в узле *node,*
- Функция *VALUEOF(node, c)* — возвращает значение (указатель на узел), ассоциированное с символом *c* в узле *node,*
- Процедура *GETNEW(node, c)* делает значение узла *node* с символом *c* указателем на новый узел.
- MAKENULL(node), делающая узел *node* пустым отображением.

Реализовать нагруженное дерево с использованием указателей.
Реализовать нагруженное дерево с использованием списков.

Лабораторная работа - 3.

Тема: Программная реализация АТД, основанных на множествах.

Программная реализация специальных методов представления множеств.

Цель работы: изучение способов представления АТД «Множество» и их реализации средствами языков программирования.

В результате выполнения лабораторной работы студенты должны:

- *знать* методы представления множеств, а также словарей, мультисписков, очередей с приоритетами и других специальных видов множеств;
- *уметь* писать и отлаживать программные модули, реализующие различные виды множеств.

Множество является той базовой структурой, которая лежит в основании всей математики. При разработке алгоритмов множества используются как основа многих важных абстрактных типов данных, и многие технические приемы и методы разработаны для реализации абстрактных типов данных, основанных именно на множествах.

Множеством называется некая совокупность *элементов*, каждый элемент множества или сам является множеством, или является примитивным элементом, называемым *атомом*. Далее будем предполагать, что все элементы любого множества различны, т.е. в любом множестве нет двух копий одного и того же элемента.

Когда множества используются в качестве инструмента при разработке алгоритмов и структур данных, то атомами обычно являются целые числа, символы, строки символов и т.п., и все элементы одного множества, как правило, имеют одинаковый тип данных. Мы часто будем предполагать, что атомы линейно упорядочены с помощью отношения, обычно обозначаемого символом "<" и читаемого как "меньше чем" или "предшествует". *Линейно упорядоченное* множество S удовлетворяет следующим двум условиям.

1. Для любых элементов a и b из множества S может быть справедливым только одно из следующих утверждений: $a < b$, $a = b$ или $b < a$.
2. Для любых элементов a, b и c из множества S таких, что $a < b$ и $b <$ c, следует $a < c$ (свойство транзитивности).

Рассмотрим операторы, выполняемые над множествами, которые часто включаются в реализацию различных абстрактных типов данных. Некоторые из этих операторов имеют общепринятые (на сегодняшний день) названия и эффективные методы их реализации. Следующий список содержит наиболее общие и часто используемые операторы множеств (т.е. выполняемые над множествами).

1-3. Первые три процедуры UNION(A, B, C), INTERSECTION(A, B, C) и DIFFERENCE(A, *B, C*) имеют "входными" аргументами множества A и B, а в

качестве результата — "выходное" множество С, равное соответственно $A \cup B$, $A \cap B$ и $A \setminus B$.

4. Иногда мы будем использовать оператор, который называется *слияние* (merge), или *объединение непересекающихся множеств*. Этот оператор (обозначается MERGE) не отличается от оператора объединения двух множеств, но здесь предполагается, что множества-операнды *не пересекаются* (т.е. не имеют общих элементов). Процедура MERGE(*A, B, C*) присваивает множеству С значение А и В, но результат будет не определен, если $A \cap B \neq 0$, т.е. в случае, когда множества А и В имеют общие элементы.

5. Функция MEMBER(*x, A*) имеет аргументами множество А и объект *x* того же типа, что и элементы множества *A*, и возвращает булево значение true (истина), если $x \in A$, и значение false (ложь), если $x \notin A$.

6. Процедура MAKENULL(A) присваивает множеству А значение пустого множества.

7. Процедура INSERT(*x, A*), где объект *x* имеет тот же тип данных, что и элементы множества А, делает *x* элементом множества А. Другими словами, новым значением множества А будет $A \cup \{x\}$. Отметим, что в случае, когда элемент *x* уже присутствует в множестве *A*, это множество не изменяется в результате выполнения данной процедуры.

4. Процедура DELETE(*x, A*) удаляет элемент *x* из множества *A*, т.е. заменяет множество А множеством $A \setminus \{x\}$. Если элемента *x* нет в множестве А, то это множество не изменяется.

5. Процедура ASSIGN(*A,B*) присваивает множеству А в качестве значения множество В.

6. Функция MIN(*A*) возвращает наименьший элемент множества А. Для применения этой функции необходимо, чтобы множество А было параметризировано и его элементы были линейно упорядочены. Например, MIN({2, 3, 1}) — 1 и MIN({'a', 'b', 'c'}) = 'a' Подобным образом определяется функция MAX.

7. Функция EQUAL(*A, B*) возвращает значение true тогда и только тогда, когда множества А и Б состоят из одних и тех же элементов.

8. Функция FIND(*x*) оперирует в среде, где есть набор непересекающихся множеств. Она возвращает имя (единственное) множества, в котором есть элемент *x*.

3.1. Реализация множеств посредством двоичных векторов

Если все рассматриваемые множества будут подмножествами небольшого *универсального множества* целых чисел 1, ..., *N* для некоторого фиксированного *N*, тогда можно применить реализацию АТД SET посредством двоичного (булева) вектора. В этой реализации множество представляется двоичным вектором, в котором *i*-й бит равен 1 (или true), если *i* является элементом множества. Главное преимущество этой реализации состоит в том, что здесь операторы MEMBER, INSERT и DELETE можно выполнить за фиксированное время (независимо от размера множества) путем прямой адресации к соответствующему биту. Но операторы UNION, INTERSECTION и

DIFFERENCE выполняются за время, пропорциональное размеру универсального множества.

С помощью объявлений языка Pascal АТД SET можно определить следующим образом:

```
const
    N ={подходящее числовое значение}; type
    SET = packed array[1..N] of boolean;
```

Реализация оператора UNION

```
procedure UNION (A, B: SET; var C: SET );
    var
        i: integer; begin
        for i:= 1 to N do
            C[i]:= A[i] or B[i] end;
```

3.2. Реализация множеств посредством связанных списков

Очевиден способ представления множеств посредством связанных списков, когда элементы списка являются элементами множества. В отличие от представления множеств посредством двоичных векторов, в данном представлении занимаемое множеством пространство пропорционально размеру представляемого множества, а не размеру универсального множества. Кроме того, представление посредством связанных списков является более общим, поскольку здесь множества не обязаны быть подмножествами некоторого конечного универсального множества.

Процедура вставки элемента

```
procedure INSERT  {x:elementtype; p:↑celltype);
var
        current, newcell: ↑celltype;
        begin
        current:= p;
        while current↑.next <> nil do begin
            if current↑.next↑.element = x then
                return; { элемент x уже есть в списке }
            if current↑.next↑.element > x then
            goto add; { далее останов процедуры }
                current:= current↑. next
          end;
    add: {здесь current — ячейка, после которой надо вставить x}
          new(newcell);
          newcell↑.element:= x;
          newcell↑.next:= current↑.next;
          current↑.next:=newcell
end; { INSERT }
```

3.3. Словари

Применение множеств при разработке алгоритмов не всегда требует таких мощных операторов, как операторы объединения и пересечения. Часто достаточно только хранить в множестве "текущие" объекты с периодической вставкой или удалением некоторых из них. Время от времени также возникает необходимость узнать, присутствует ли конкретный элемент в данном множестве. Абстрактный тип множеств с операторами INSERT, DELETE и MEMBER называется DICTIONARY (Словарь).

Словари можно представить посредством сортированных или несортированных связанных списков. Другая возможная реализация словарей использует двоичные векторы, предполагая, что элементы данного множества являются целыми числами 1, ..., N для некоторого N или элементы множества можно сопоставить с таким множеством целых чисел.

Третья возможная реализация словарей использует массив фиксированной длины с указателем на последнюю заполненную ячейку этого массива. Эта реализация выполнима, если мы точно знаем, что размер множества не превысит заданную длину массива.

Объявления типов и процедуры реализации словаря посредством массива

```
const
    maxsize = { некое число, максимальный размер массива }
type
    DICTIONARY = record
        last: integer;
        data: array[1..maxsize] of nametype end;
procedure MAKENULL (var A: DICTIONARY);
    begin
        A.last := 0 end; { MAKENULL }
function MEMBER (x: nametype; var A:DICTIONARY): boolean;
    var
        i: integer; begin
        for i:= 1 to A.last do
          if A.data[i] = x then return(true);
          return(false) { элемент x не найден }
    end; { MEMBER }
procedure INSERT (x: nametype; var A: DICTIONARY);
    begin
        if not MEMBER(x,A) then
            if A.last < maxsize then begin
              A.last:= A.last + 1;
              A.data[A. last] := x end
            else error('База данных заполнена')
    end; { INSERT }
procedure DELETE (x: nametype; var A: DICTIONARY);
    var
        i= integer; begin
```

```
        if A.last > 0 then begin
            i:= 1;
            while (A.data[i] <> x) and (i < A. last) do
                i:= i + 1;
            if A.data[i] = x then begin
                    A.data[i] = A.data[A.last];
                      { перемещение последнего элемента на место
                    элемента x; если i = A.last, то удаление x
                          происходит на следующем шаге }
                    A.last:= A.last - 1
            end
        end
    end; { DELETE }
```

3.4. Реализация АТД для отображений

Для этого АТД мы определили такие операторы.

1. MAKENULL(*A*). Инициализирует отображение *A*, где ни одному элементу области определения не соответствует ни один элемент области значений.
2. ASSIGN(*A, d, r)*. Задает для *A(d)* значение *r*.
3. COMPUTE(*A, d, r)*. Возвращает значение true и устанавливает значение *r* для *A(d)*, если *A(d)* определено, в противном случае возвращает значение false.

Отображения можно эффективно реализовать с помощью хеш-таблиц.

```
type
    celltype = record
        domainelement: domaintype;
        range: rangetype;
        next: ↑celltype end
```

Здесь domaintype и rangetype — типы данных для элементов области определения и области значений соответственно. Объявление АТД MAPPING следующее;

```
type
    MAPPING = array[0..B-1] of ↑celltype
```

Последний массив — это массив сегментов для хеш-таблицы.

Процедура ASSIGN для открытой хеш-таблицы

```
procedure ASSIGN (var A: MAPPING; d:domaintype;r:rangetype);
var
    bucket: integer;
    current: ↑cell type;
begin
    bucket:= h(d);
    current:= A[bucket] ;
    while current <> nil do
        if current↑.domainelement = d then begin
            current↑.range:=r;{замена старого значения для d}
            return
```

```
            end
            else
              current:=current↑.next; { d не найден в списке}
        current:= A[bucket];
        {использование current для запоминания первой ячейки}
        new(A[bucket]);
        A[bucket]↑.domainelement:= d;
        A[bucket]↑.range: =r;
        A[bucket]↑.next:= current;
end;  { ASSIGN }
```

3.5. Структуры мультисписков

В общем случае структура мультисписка — это совокупность ячеек, некоторые из которых имеют более одного указателя и поэтому одновременно могут принадлежать нескольким спискам. Для каждого типа ячеек важно различать поля указателей для разных списков, чтобы можно было проследить элементы одного списка, не вступая в противоречие с указателями другого списка.

Пример отношения между множеством студентов и множеством учебных курсов

	CS101	CS202	CS303
Alan			X
Alex	X	X	
Alice			X
Amy	X		
Andy		X	X
Ann	X		X

Регистрация (студентов по учебным курсам)

Пока пусть каждое множество C_s является множеством регистрационных записей, соответствующих конкретному студенту s и некоторому учебному курсу c. Если принять за регистрацию пару (s, c), то множество C_s можно определить следующим образом:

C_s — $\{(s, c) \mid$ студент s записан на курс $c\}$,

Аналогично определяется множество S_c:

S_c — $\{(s, c) \mid$ студент s записан на курс $c\}$.

С этой точки зрения можно задать поля указателя в каждой записи для студента и для курса, которые указывали бы на первые регистрационные записи множеств C_s и S_c соответственно. Каждая регистрационная запись должна иметь два поля указателя, одно из них, назовем его *cnext*, будет указывать на следующую регистрационную запись списка множества C_s, которому она принадлежит, а второе поле, *snext*, будет указывать на

следующую регистрационную запись списка множества S_c, которому она также принадлежит.

Реализация поиска в мультисписке

```
type
    stype = array[1..20] of char;
    ctype = array[1..5] of char;
    recordkinds = (student, course, enrollment);
    recordtype = record
        case kind : recordkinds of
            student: (studentname: stype;
                      firstcourse:↑recordtype); course:(coursename: ctype;
                      firststudent:↑recordtype);
            enrollment: (cnext, snext:↑recordtype)
        end;
procedure printstudents(cname: ctype );
    var
        c, e, f: ↑recordtype;
        begin
        c:=указатель на запись курса,где c↑.coursename=cname;
        { последний оператор зависит от того,
            как реализовано множество записей курсов }
        e:= c↑.firststudent; {e пробегает по кольцу
                              указателей регистрационных записей }
        while e↑.kind = enrollment do begin
            f:=e;
            repeat
                f: = f↑.cnext
            until
            f↑.kind = student;
                      {сейчас f — указатель на студента-
                      собственника регистрации e↑}
            writeln(f↑. studentname) ;
            e: = e↑. snext
        end
    end; { printstudents }
```

3.6. Множества с операторами MERGE и FIND

В этом разделе мы рассмотрим ситуацию, когда есть совокупность объектов, каждый из которых является множеством. Основные действия, выполняемые над такой совокупностью, заключаются в объединении множеств в определенном порядке, а также в проверке принадлежности определенного объекта конкретному множеству. Эти задачи решаются с помощью операторов MERGE (Слить) и FIND (Найти). Оператор MERGE(A, *B, C)* делает множество *C* равным объединению множеств A и B, если эти множества не пересекаются (т.е. не имеют общих элементов); этот оператор не определен, если множества

A и B пересекаются. Функция FIND(*x*) возвращает множество, которому принадлежит элемент *x;* в случае, когда *x* принадлежит нескольким множествам или не принадлежит ни одному, значение этой функции не определено.

Пример. *Отношением эквивалентности* является отношение со свойствами рефлексивности, симметричности и транзитивности. Другими словами, если на множестве *S* определено отношение эквивалентности, обозначаемое символом "е", то для любых элементов *a, b* и *c* из множества *S* (не обязательно различных) справедливы следующие соотношения.

1. $a \equiv a$ (свойство рефлексивности).
2. Если $a \equiv b$, то $b \equiv a$ (свойство симметричности).
3. Если $a \equiv b$ и $b \equiv c$, то $a \equiv c$ (свойство транзитивности).

Отношение равенства (обозначается знаком "=") — это пример отношения эквивалентности на любом множестве *S.* Для любых элементов *a, b* и *c* из множества *S* имеем (1) *a* — *a;* (2) если $a = b$, то $b = a$; (3) если $a = b$ и $b = c$, то $a = c$. Далее мы встретимся с другими примерами отношения эквивалентности.

В общем случае отношение эквивалентности позволяет разбить совокупность объектов на непересекающиеся группы, когда элементы *a* и *b* будут принадлежать одной группе тогда и только тогда, когда $a \equiv b$. Если применить отношение равенства (частный случай отношения эквивалентности), то получим группы, состоящие из одного элемента.

Более формально можно сказать так: если на множестве *S* определено отношение эквивалентности, то в соответствии с этим отношением множество *S* можно разбить на непересекающиеся подмножества S_1, S_2,..., которые называются *классами эквивалентности,* и объединение этих классов совпадает с *S.* Таким образом, $a = b$ для всех *a* и *b* из подмножества S_1, но *a* не эквивалентно *b,* если они принадлежат разным подмножествам. Например, отношение сравнения по модулю n^1 — это отношение эквивалентности на множестве целых чисел. Чтобы показать это, достаточно заметить, что $a - a = 0$ (рефлексивность), если $a - b = dn$, то $b - a = (-d)n$ (симметричность), если $a - b = dn$ и $b - c = en$, то $a - c = (d + e)n$ (транзитивность). В случае сравнения по модулю *n* существует *n* классов эквивалентности, каждое из которых является множеством целых чисел. Первое множество состоит из целых чисел, которые при делении на *n* дают остаток 0, второе множество состоит из целых чисел, которые при делении на *n* дают остаток 1, и т.д., *n*-е множество состоит из целых чисел, которые дают остаток $n - 1$.

Задачу эквивалентности можно сформулировать следующим образом. Есть множество *S* и последовательность утверждений вида *"a* эквивалентно *b".* Надо по представленной последовательности таких утверждений определить, какому классу эквивалентности принадлежит предъявленный элемент. Например, есть множество $S = \{1, 2, ..., 7\}$ и последовательность утверждений

$$1 \equiv 2 \quad 5 \equiv 6 \quad 3 \equiv 4 \quad 1 \equiv 4$$

Необходимо построить классы эквивалентности множества *S.* Сначала полагаем, что все элементы этого множества представляют отдельные классы,

затем, применяя заданную последовательность утверждений, объединяем "индивидуальные" классы. Вот последовательность этих объединений:

1≡2	{1,2} {3} {4} {5} {6} {7}
5≡ 6	{1, 2} {3} {4} {5, 6} {7}
3 ≡ 4	{1,2} {3,4} {5,6} {7}
1 ≡ 4	{1, 2, 3, 4} {5, 6} {7}

Можно "решать" задачу эквивалентности начиная с любого утверждения заданной последовательности утверждений. При "обработке" утверждения *a* = *b* сначала с помощью оператора FIND находятся классы эквивалентности для элементов *a* и *b*, затем к этим классам применяется оператор MERGE.

Задача эквивалентности часто встречается во многих областях компьютерных наук. Например, она возникает при обработке компилятором Fortran "эквивалентных объявлений", таких как

EQUIVALENCE (A(1),B(1,2),C(3)), (A(2),D,E), (F,G)

3.7. Простая реализация АТД MFSET

Начнем с простейшего АТД, реализующего операторы MERGE и FIND. Этот АТД, назовем его MFSET (Множество с операторами MERGE и FIND), можно определить как множество, состоящее из подмножеств, со следующими операторами.

1. MERGE(A, *B)* объединяет компоненты A и B, результат присваивается или A, или *B.*
2. FIND(*x*) — функция, возвращающая имя компонента, которому принадлежит *x.*
3. INITIAL(A, *x)* создает компонент с именем A, содержащим только элемент *x.*

Для корректной реализации АТД MFSET надо разделить исходные типы данных или объявить, что MFSET состоит из данных двух разных типов — типа имен множеств и типа элементов этих множеств. Во многих приложениях можно использовать целые числа для имен множеств. Если общее количество элементов всех компонент равно *n,* то можно использовать целые числа из интервала от 1 до л для идентификации элементов множеств. Если принять это предположение, то в таком случае существенно, что номерами элементов можно индексировать ячейки массива. Тип имен компонентов не так важен, поскольку это тип ячеек массива, а не индексов. Но если мы хотим, чтобы тип элементов множеств был отличным от числового, то необходимо применить отображение, например посредством хеш-функции, ставящее в соответствие элементам множеств уникальные целые числа из заданного интервала. В последнем случае надо знать только общее число элементов всех компонентов.

После всего сказанного не должно вызывать возражений следующее объявление типов:

```
const
    n = { количество всех элементов };
type
    MFSET = array[1..n] of integer;
```

или объявление более общего типа

array[интервал элементов] **of** (тип имен множеств)

Предположим, что мы объявили тип MFSET как массив *components* (компоненты), предполагая, что *components[x]* содержит имя множества, которому принадлежит элемент *x*. При этих предположениях операторы АТД MFSET реализуются легко, что видно из листинга с кодом процедуры MERGE. Процедура INITIAL(A, *x)* просто присваивает *components[x]* значение A, а функция FIND(*x*) возвращает значение *components[x]*.

Процедура MERGE

```
procedure MERGE ( A, B: integer; var C: MFSET );
    var
        x: 1..n;begin
        for x:= 1 to n do
          if C[x] = B then
                C[x]:= A
    end;   { MERGE  }
```

Время выполнения операторов при такой реализации MFSET легко проанализировать. Время выполнения процедуры MERGE имеет порядок $O(n)$, а для INITIAL и FIND время выполнения фиксированно, т.е. не зависит от n.

3.8. АТД с операторами MERGE и SPLIT

Пусть S — множество, элементы которого упорядочены посредством отношения "<". Оператор разбиения SPLIT(S, S_1, S_2, x) разделяет множество S на два множества: $S_1 = \{ a \mid a \in S$ и $a < x \}$ и $S_2 = \{ a \mid a \in S$ и $a \geq x \}$. Множество S после разбиения не определено (если не оговорено, что оно принимает значение S_1 или S_2). Можно привести много различных ситуаций, когда необходимо разделить множество путем сравнения всех его элементов с одним заданным значением.

Задание

Написать программный модуль или класс, реализующий заданный абстрактный тип данных, и программу, демонстрирующую возможности этого типа данных.

1. Множество символов, реализованное с использованием двоичного вектора.
2. Множество записей, реализованное с использованием списков.
3. Словарь с использованием двоичного вектора и отображения.
4. Словарь с использованием односвязного списка.
5. Словарь с использованием двусвязного списка.
6. Словарь с использованием массива.
7. Словарь с использованием циклического массива.
8. Словарь с использованием открытой хеш-таблицы.
9. Словарь с использованием закрытой хеш-таблицы.
10. Отображение с использованием открытого хеширования.
11. Отображение с использованием закрытого хеширования.

12.Очередь с приоритетами — это АТД, основанный на модели множеств с операторами INSERT и DELETEMIN, а также с оператором MAKENULL для инициализации структуры данных. Перед определением нового оператора DELETEMIN сначала объясним, что такое "очередь с приоритетами". Этот термин подразумевает, что на множестве элементов задана функция приоритета (priority), т.е. для каждого элемента *a* множества можно вычислить функцию *p(a), приоритет элемента a,* которая обычно принимает значения из множества действительных чисел, или, в более общем случае, из некоторого линейно упорядоченного множества. Оператор INSERT для очередей с приоритетами понимается в обычном смысле, тогда как DELETEMIN является функцией, которая возвращает элемент с наименьшим приоритетом и в качестве побочного эффекта удаляет его из множества. Реализовать такую структуру данных с использованием списков.

13.Реализовать очередь с приоритетами с использованием бинарных деревьев.

Реализовать мультисписок с функциями добавления и удаления элемента, а также поиска и вывода элементов.

Лабораторная работа - 4.

Тема: Программная реализация представления ориентированных графов.

Программная реализация представления неориентированных графов.

Цель работы: изучение способов представления графов при помощи различных структур данных и реализации алгоритмов, использующих эти структуры данных.

В результате выполнения лабораторной работы студенты должны:

- *знать* основные понятия и виды графов; способы их машинного представления; основные алгоритмы для работы с графами;
- *уметь* писать и отлаживать программные модули, реализующие АТД «Граф».

Граф - это сложная нелинейная многосвязная динамическая структура, отображающая свойства и связи сложного объекта.

Многосвязная структура обладает следующими свойствами:

- на каждый элемент (узел, вершину) может быть произвольное количество ссылок;
- каждый элемент может иметь связь с любым количеством других элементов;
- каждая связка (ребро, дуга) может иметь направление и вес.

В узлах графа содержится информация об элементах объекта. Связи между узлами задаются ребрами графа. Ребра графа могут иметь направленность, показываемую стрелками, тогда они называются ориентированными, ребра без стрелок - неориентированные.

Граф, все связи которого ориентированные, называется ориентированным графом или орграфом; граф со всеми неориентированными связями - неориентированным графом; граф со связями обоих типов - смешанным графом. Обозначение связей: неориентированных - (A,B), ориентированных. Примеры изображений графов даны на рис.4.1. Скобочное представление графов:

а).((A,B),(B,A)) и б).(< A,B >,< B,A >).

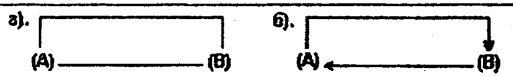


рис. 4.1 Граф неориентированный (а) и ориентированный (б).

Для ориентированного графа число ребер, входящих в узел, называется ***полустепенью захода узла***, выходящих из узла ***-полустепенью исхода.*** Количество входящих и выходящих ребер может быть любым, в том числе и нулевым. Граф без ребер является ***нуль-графом***.

Если ребрам графа соответствуют некоторые значения, то граф и ребра называются *взвешенными*. *Мультиграфом* называется граф, имеющий параллельные (соединяющие одни и те же вершины) ребра, в противном случае граф называется простым.

Путь в графе - это последовательность узлов, связанных ребрами; элементарным называется путь, в котором все ребра различны, простым называется путь, в котором все вершины различны. Путь от узла к самому себе называется циклом, а граф, содержащий такие пути - циклическим.

Два узла графа смежны, если существует путь от одного из них до другого. Узел называется инцидентным к ребру, если он является его вершиной, т.е. ребро направлено к этому узлу.

Логически структура-граф может быть представлена матрицей смежности или матрицей инцидентности.

Матрицей смежности для n узлов называется квадратная матрица adj порядка n. Элемент матрицы a(i,j) равен 1, если узел j смежен с узлом i (есть путь < i,j >), и 0 -в противном случае (рис.4.2).

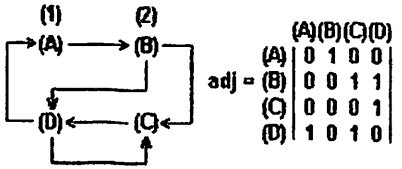


рис 4.2

4.1. Граф и его матрица смежности

Если граф неориентирован, то a(i,j)=a(j,i), т.е. матрица симметрична относительно главной диагонали.

Матрицы смежности используются при построении матриц путей, дающих представление о графе по длине пути: путь длиной в 1 - смежный участок - , путь длиной 2 - (< A,B >,< B,C >), ... в n смежных участков: где n - максимальная длина, равная числу узлов графа. На рис.4.3 даны путевые матрицы пути adj2, adj3, adj4 для графа рис.4.2.

	(A)	(B)	(C)	(D)
(A)	0	0	1	1
(B)	1	0	1	1
(C)	1	0	1	0
(D)	0	1	0	1

adj2

	(A)	(B)	(C)	(D)
(A)	1	0	1	1
(B)	1	1	1	1
(C)	0	1	0	1
(D)	0	0	1	1

adj3

	(A)	(B)	(C)	(D)
(A)	1	0	1	0
(B)	0	1	0	0
(C)	0	0	1	1
(D)	0	0	1	1

adj4

рис.4.3. Матрицы путей

Матрицы инцидентности используются только для орграфов. В каждой строке содержится упорядоченная последовательность имен узлов, с которыми данный узел связан ориентированными (исходящими) ребрами. На рис.4.4 показана матрица инцидентности для графа рис. 4.2.

Узлы	1	2	номера связей
A	B	-	
B	C	D	
C	D	-	
D	A	C	

рис.4.4 Матрицы инцидентности

Существуют два основных метода представления графов в памяти ЭВМ: матричный, т.е. массивами, и связными нелинейными списками. Выбор метода представления зависит от природы данных и операций, выполняемых над ними. Если задача требует большого числа включений и исключений узлов, то целесообразно представлять граф связными списками; в противном случае можно применить и матричное представление.

4.2. Матричное представление орграфов.

При использовании матриц смежности их элементы представляются в памяти ЭВМ элементами массива. При этом, для простого графа матрица состоит из нулей и единиц, для мультиграфа - из нулей и целых чисел, указывающих кратность соответствующих ребер, для взвешенного графа - из нулей и вещественных чисел, задающих вес каждого ребра.

Например, для простого ориентированного графа, изображенного на рис.4.2 массив определяется как:

```
mas:array[1..4,1..4]=((0,1,0,0),(0,0,1,1),(0,0,0,1),(1,0,1,0))
```

Матрицы смежности применяются, когда в графе много связей и матрица хорошо заполнена.

Связное представление орграфов. Орграф представляется связным нелинейным списком, если он часто изменяется или если полустепени захода и исхода его узлов велики. Рассмотрим два варианты представления орграфов связными нелинейными списковыми структурами.

В первом варианте два типа элементов - атомарный и узел связи. На рис.4.5 показана схема такого представления для графа рис.4.2. Скобочная запись связей этого графа:

(< A,B >, < B,C >, < C,D >, < B,D >, < D,C >)

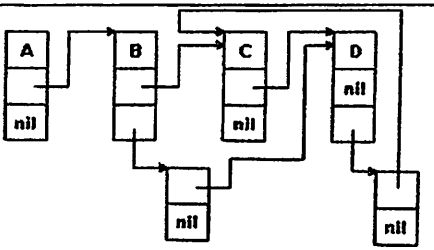


рис 4.5

4.3. Машинное представление графа элементами двух типов

Более рационально представлять граф элементами одного формата, двойными: атом-указатель и указатель-указатель или тройными: указатель-data/down-указатель. На рис.4.6 тот же граф представлен элементами одного формата.

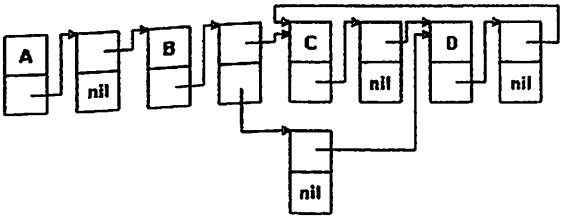


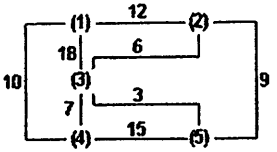
Рис 4.6 Машинное представление графа однотипными элементами

Многосвязная структура - граф - находит широкое применение при организации банков данных, управлении базами данных, в системах программного имитационного моделирования сложных комплексов, в системах исскуственного интеллекта, в задачах планирования и в других сферах.

В качестве примера приведем программу, находящую кратчайший путь между двумя указанными вершинами связного конечного графа.

Пусть дана часть карты дорожной сети и нужно найти наилучший маршрут от города 1 до города 5. Такая задача выглядит достаточно простой, но "наилучший" маршрут могут определять многие факторы. Например: (1) расстояние в километрах; (2) время прохождения маршрута с учетом ограничений скорости; (3) ожидаемая продолжительность поездки с учетом дорожных условий и плотности движения; (4) задержки, вызванные проездом через города или объездом городов; (5) число городов, которое необходимо посетить, например, в целях доставки грузов. Задачи о кратчайших путях относятся к фундаментальным задачам комбинаторной оптимизации.

Среди десятков алгоритмов для отыскания кратчайшего пути один из лучших принадлежит Дейкстре. Алгоритм Дейкстры, определяющий кратчайшее расстояние от данной вершины до конечной, легче пояснить на примере. Рассмотрим граф, задающий связь между городами на карте дорог. Представим граф матрицей смежности A, в которой: A(i,j)-длина ребра между узлами i и j. Используя полученную матрицу и матрицы, отражающие другие факторы, можно определить кратчайший путь.



A =

	(1)	(2)	(3)	(4)	(5)	
	0	12	18	10	.	(1)
	12	0	6	.	9	(2)
	18	6	0	7	3	(3)
	10	.	7	0	15	(4)
	.	9	3	15	0	(5)

"." - ребро отсутствует

Часть дорожной карты, представленная в виде взвешенного графа и его матрицы смежности

4.4. Обход орграфа в глубину.

Предположим, что есть ориентированный граф G, в котором первоначально все вершины помечены меткой *unvisited* (не посещалась). Поиск в глубину начинается с выбора начальной вершины *v* графа *G*, для этой вершины метка *unvisited* меняется на метку *visited* (посещалась). Затем для каждой вершины, смежной с вершиной *v* и которая не посещалась ранее, рекурсивно применяется поиск в глубину. Когда все вершины, которые можно достичь из вершины *v*, будут "удостоены" посещения, поиск заканчивается. Если некоторые вершины остались не посещенными, то выбирается одна из них и поиск повторяется. Этот процесс продолжается до тех пор, пока обходом не будут охвачены все вершины орграфа G.

Этот метод обхода вершин орграфа называется поиском в глубину, поскольку поиск не посещенных вершин идет в направлении вперед (вглубь) до тех пор, пока это возможно.

Для представления вершин, смежных с вершиной *v*, можно использовать список смежности *L[v]*, а для определения вершин, которые ранее посещались, — массив *mark* (метка), чьи элементы будут принимать только два значения: *visited* и *unvisited*.

4.5. Остовные деревья минимальной стоимости

Пусть $G = (V, E)$ — связный граф, в котором каждое ребро (v, w) помечено числом $c(v, w)$, которое называется *стоимостью ребра*. *Остовным деревом* графа *G* называется свободное дерево, содержащее все вершины *V* графа G. *Стоимость* остовного дерева вычисляется как сумма стоимостей всех ребер, входящих в это дерево.

Существуют два популярных метода построения остовного дерева минимальной стоимости для помеченного графа $G = (V, E)$, основанные на свойстве ОДМС. Один такой метод известен как *алгоритм Прима* (Prim). В этом алгоритме строится множество вершин *U*, из которого "вырастает" остовное дерево. Пусть V— {1, 2, ..., *n*}. Сначала $U = \{1\}$. На каждом шаге алгоритма находится ребро наименьшей стоимости (u, v) такое, что $u \in U$ и $v \in V \setminus U$, затем вершина *v* переносится из множества $V \setminus U$ в множество *U*. Этот процесс продолжается до тех пор, пока множество *U* не станет равным множеству *V*.

В *алгоритме Крускала* (Kruskal) построение остовного дерева минимальной стоимости для графа *G* начинается с графа $T = (V, 0)$, состоящего только из *n* вершин графа *G* и не имеющего ребер. Таким образом, каждая вершина является связной (о, самой собой) компонентой. В процессе выполнения алгоритма мы имеем набор связных компонент, постепенно объединяя которые формируем остовное дерево.

При построении связных, постепенно возрастающих компонент поочередно проверяются ребра из множества E в порядке возрастания их стоимости. Если очередное ребро связывает две вершины из разных компонент, тогда оно добавляется в граф T. Если это ребро связывает две вершины из одной компоненты, то оно отбрасывается, так как его добавление в связную компоненту, являющуюся свободным деревом, приведет к образованию цикла. Когда все вершины графа G будут принадлежать одной компоненте, построение остовного дерева минимальной стоимости T для этого графа заканчивается.

Пример: написать программу для поиска кратчайшего пути между вершинами в графе.

```
{========== Программный пример 4.1 ==============}
{ Алгоритм Дейкстры }

Program ShortWay;
Const n=5;  max=10000;
Var a: Array [1..n,1..n] of Integer;
   v0,w,edges: Integer;
   from,tu,length: Array [1..n] of Integer;
Procedure adjinit;
{ Эта процедура задает веса ребер графа  посредством
определения его матрицы смежности A размером N x N }
Var i,j: Integer;
Begin
{ "Обнуление" матрицы (вершины не связаны) }
 For i:=1 to n do
  For j:=1 to n do a[i,j]:=max;
 For i:=1 to n do a[i,i]:=0;
{ Задание длин ребер, соединяющих смежные узлы графа }
 a[1,2]:=12; a[1,3]:=18; a[1,4]:=10;
 a[2,1]:=12; a[2,3]:=6;  a[2,5]:=9;
 a[3,1]:=18; a[3,2]:=6;  a[3,4]:=7;  a[3,5]:=3;
 a[4,1]:=10; a[4,3]:=7;  a[4,5]:=15;
 a[5,2]:=9;  a[5,3]:=3;  a[5,4]:=15;
End;

Procedure printmat;
{ Эта процедура выводит на экран дисплея матрицу
смежности A взвешенного графа }
Var i,j: Integer;
Begin  writeln;
 writeln('Матрица смежности взвешенного графа (',n,'x',n,'):');
 writeln;
 For i:=1 to n do
```

```
  Begin  write ('Ë');
   For j:=1 to n do
    If a[i,j]=max Then write('  ----')  Else write(a[i,j]:6);
   writeln(' Ë')
  End; writeln;
 writeln (' ("----" - ребро отсутствует)')
End;
Procedure dijkst;
```

Эта процедура определяет кратчайшее расстояние от начальной вершины V0 до конечной вершины W в связном графе с неотрицательными весами с помощью алгоритма, принадлежащего Дейкстре.

Результатом работы этой процедуры является дерево кратчайших путей с корнем V0.

---- Входные и выходные переменные ---

A(I,J)	длина ребра, соединяющего вершины I и J. Если ребро отсутствует, то A(I,J)=10000 (произвольному большому числу).
V0	начальная вершина.
W	конечная вершина.
N	вершины в графе пронумерованы 1,...,N.
FROM(I) TU(I)	содержит I-е ребро в дереве кратчайших путей от вершины FROM(I) к вершине TU(I)
LENGTH(I)	длины LENGTH(I).
EDGES	число ребер в дереве кратчайших путей на данный момент.

--- Внутренние переменные ---

DIST(I)	кратчайшее расстояние от UNDET(I) до частичного дерева кратчайших путей.
NEXT	очередная вершина, добавляемая к дереву кратчайших путей.
NUMUN	число неопределенных вершин.
UNDET(I)	список неопределенных вершин.
VERTEX(I)	вершины частичного дерева кратчайших путей, лежащие на кратчайшем пути от UNDET(I) до V0. }

```
Label stpoint;
Var dist,undet,vertex: array[1..n] of Integer;
  next,numun,i,j,k,l,jk: Integer;
```

```
Begin
 edges:=0; next:=v0; numun:=n-1;
 For i:=1 to n do
  Begin  undet[i]:=i; dist[i]:=a[v0,i]; vertex[i]:=v0 End;
 undet[v0]:=n; dist[v0]:=dist[n];
 goto stpoint;
 Repeat
{ Исключение вновь определенной вершины из списка неопределенных}
  dist[k]:=dist[numun]; undet[k]:=undet[numun];
  vertex[k]:=vertex[numun];
{ Остались ли неопределеные вершины ? }
  dec(numun);
{ Обновление кратчайшего расстояния до всех неопределенных вершин }
  For i:=1 to numun do
  Begin   j:=undet[i]; jk:=l+a[next,j];
     If dist[i] > jk Then Begin  vertex[i]:=next; dist[i]:=jk  End
  End;
stpoint:{Запоминание кратчайшего расст.до неопределенной вершины}
  k:=1; l:=dist[1];
  For i:=1 to numun do
   If dist[i] < l Then Begin  l:=dist[i]; k:=i  End;
{ Добавление ребра к дереву кратчайших путей }
  inc(edges);  from[edges]:=vertex[k]; tu[edges]:=undet[k];
  length[edges]:=l; next:=undet[k]
 Until next = w        { Достигли ли мы w }
End;
Procedure showway;
{ Эта процедура выводит на экран дисплея кратчайшее расстояние между
вершинами V0 и W взвешенного графа, определенное процедурой dijkst }
Var i: Integer;
Begin
 writeln;  writeln('Кратчайшее расстояние между');
 writeln('узлами ',v0,' и ',w,' равно ',length[edges])
End;
{ Основная программа }
Begin
 adjinit;  printmat;   v0:=1;w:=5;
 dijkst;  showway;   readln
End.
```

Задание.

Написать программный модуль или класс, реализующий заданный абстрактный тип данных, и программу, демонстрирующую возможности этого типа данных. Можно использовать любой способ реализации (массивы, курсоры, указатели, пройденные ранее АТД).

1. Ориентированный простой граф с реализацией операторов чтения меток вершин и дуг, вставки и удаления вершин и дуг и оператора перемещения по последовательностям дуг, а также вывода данных в вершинах.

2. Ориентированный мультиграф с реализацией операторов чтения меток вершин и дуг, вставки и удаления вершин и дуг и оператора перемещения по последовательностям дуг, а также вывода данных на дугах.

3. Ориентированный взвешенный граф с реализацией функции поиска кратчайшего пути между заданными вершинами.

4. Ориентированный граф, реализующий функцию обхода в глубину.

5. Ориентированный граф с функцией поиска циклов в графе.

6. Простой неорграф с реализацией операторов чтения меток вершин и дуг, вставки и удаления вершин и дуг и оператора перемещения по последовательностям дуг, а также вывода данных в вершинах.

7. Неориентированный взвешенный мультиграф с реализацией операторов чтения меток вершин и дуг, вставки и удаления вершин и дуг и оператора перемещения по последовательностям дуг, а также вывода данных на рёбрах.

8. Неорграф с функцией поиска кратчайшего пути.

9. Неорграф с функцией определения связности.

10. Неорграф с реализацией функции поиска минимального остовного дерева.

11. Неорграф с реализацией обхода в глубину.

12. Неорграф с реализацией обхода в ширину.

Лабораторная работа - 5.

Тема: Программная реализация алгоритмов внутренней и внешней сортировки.

Цель работы: изучение алгоритмов внутренней и внешней сортировки.
В результате выполнения лабораторной работы студенты должны:

- *знать* различные методы сортировки;
- *уметь* писать и отлаживать программы, реализующие алгоритмы сортировки.

В общей постановке задача сортировки информации ставится следующим образом. Имеется последовательность однотипных записей, одно из полей которых выбрано в качестве ключевого (далее мы будем называть его ключом сортировки). Тип данных ключа должен включать операции сравнения ("=", ">", "<", ">=" и "<="). Задачей сортировки является преобразование исходной последовательности в последовательность, содержащую те же записи, но в порядке возрастания (или убывания) значений ключа. Метод сортировки называется устойчивым, если при его применении не изменяется относительное положение записей с равными значениями ключа.

Различают сортировку массивов записей, целиком расположенных в основной памяти (внутреннюю сортировку), и сортировку файлов, хранящихся во внешней памяти и не помещающихся полностью в основной памяти (внешнюю сортировку). Для внутренней и внешней сортировки требуются существенно разные методы. В этой части мы рассмотрим наиболее известные методы внутренней сортировки, начиная с простых и понятных, но не слишком быстрых, и заканчивая не столь просто понимаемыми усложненными методами.

Естественным условием, предъявляемым к любому методу внутренней сортировки является то, что эти методы не должны требовать дополнительной памяти: все перестановки с целью упорядочения элементов массива должны производиться в пределах того же массива. Мерой эффективности алгоритма внутренней сортировки являются число требуемых сравнений значений ключа (C) и число перестановок элементов (M).

Заметим, что поскольку сортировка основана только на значениях ключа и никак не затрагивает оставшиеся поля записей, можно говорить о сортировке массивов ключей. В следующих разделах, чтобы не привязываться к конкретному языку программирования и его синтаксическим особенностям, мы будем описывать алгоритмы словами и иллюстрировать их на простых примерах.

5.1. Сортировка включением

Одним из наиболее простых и естественных методов внутренней сортировки является сортировка с простыми включениями. Идея алгоритма очень проста. Пусть имеется массив ключей a[1], a[2], ..., a[n]. Для каждого

элемента массива, начиная со второго, производится сравнение с элементами с меньшим индексом (элемент a[i] последовательно сравнивается с элементами a[i-1], a[i-2] ...) и до тех пор, пока для очередного элемента a[j] выполняется соотношение a[j] > a[i], a[i] и a[j] меняются местами. Если удается встретить такой элемент a[j], что a[j] <= a[i], или если достигнута нижняя граница массива, производится переход к обработке элемента a[i+1] (пока не будет достигнута верхняя граница массива).

Легко видеть, что в лучшем случае (когда массив уже упорядочен) для выполнения алгоритма с массивом из n элементов потребуется n-1 сравнение и 0 пересылок. В худшем случае (когда массив упорядочен в обратном порядке) потребуется n(n-1)/2 сравнений и столько же пересылок. Таким образом, можно оценивать сложность метода простых включений как $O(n^2)$.

Можно сократить число сравнений, применяемых в методе простых включений, если воспользоваться тем фактом, что при обработке элемента a[i] массива элементы a[1], a[2], ..., a[i-1] уже упорядочены, и воспользоваться для поиска элемента, с которым должна быть произведена перестановка, методом двоичного деления. В этом случае оценка числа требуемых сравнений становится O(n?log n). Заметим, что поскольку при выполнении перестановки требуется сдвижка на один элемент нескольких элементов, то оценка числа пересылок остаётся O(n2).

Таблица 1 Пример сортировки методом простого включения

Начальное состояние массива	8 23 5 65 44 33 1 6
Шаг 1	8 23 5 65 44 33 1 6
Шаг 2	8 5 23 65 44 33 1 6 5 8 23 65 44 33 1 6
Шаг 3	5 8 23 65 44 33 1 6
Шаг 4	5 8 23 44 65 33 1 6
Шаг 5	5 8 23 44 33 65 1 6 5 8 23 33 44 65 1 6
Шаг 6	5 8 23 33 44 1 65 6 5 8 23 33 1 44 65 6 5 8 23 1 33 44 65 6 5 8 1 23 33 44 65 6 5 1 8 23 33 44 65 6 1 5 8 23 33 44 65 6
Шаг 7	1 5 8 23 33 44 6 65 1 5 8 23 33 6 44 65 1 5 8 23 6 33 44 65 1 5 8 6 23 33 44 65 1 5 6 8 23 33 44 65

5.3. Пирамидальная сортировка

В этом разделе мы рассмотрим алгоритм сортировки, называемой *пирамидальной*, его время выполнения в худшем случае такое, как и в среднем, и имеет порядок $O(n \log n)$. Этот алгоритм можно записать в абстрактной (обобщенной) форме, используя операторы множеств INSERT, DELETE, EMPTY и MIN Обозначим через L список элементов, подлежащих сортировке, а S — множество элементов типа recordtype (тип записи), которое будет использоваться для хранения сортируемых элементов. Оператор MIN применяется к ключевому полю записей, т.е. MIN(S) возвращает запись из множества S с минимальным значением ключа.

Абстрактный алгоритм сортировки

```
for x ∈L do
    INSERT(x, S) ;
while not EMPTY(S) do begin
    y: = MIN(S);
    writeln(y) ;
    DELETE(y, S)
end
```

5.4. „Карманная" сортировка

Предположим, что значения ключей являются целыми числами из интервала от 1 до n, они не повторяются и число сортируемых элементов также равно n. Если обозначить через A и B массивы типа array[1.. n] of recordtype, n элементов, подлежащих сортировке, первоначально находятся в массиве A, тогда можно организовать поочередное помещение в массив B записей в порядке возрастания значений ключей следующим образом:

```
for i:= 1 to n do
    B[A[i].key]:= A[i];
```

Этот код вычисляет, где в массиве B должен находиться элемент $A[i]$, и помещает его туда. Весь этот цикл требует времени порядка $O(n)$ и работает корректно только тогда, когда значения всех ключей различны и являются целыми числами из интервала от 1 до n.

Существует другой способ сортировки элементов массива A с временем $O(n)$, но без использования второго массива B. Поочередно посетим элементы A[1], ..., $A[n]$. Если запись в ячейке $A[i]$ имеет ключ j и $j \neq i$, то меняются местами записи в ячейках $A[i]$ и $A[j]$. Если после этой перестановки новая запись в ячейке $A[i]$ имеет ключ k и $k \neq i$, то осуществляется перестановка между $A[i]$ и $A[k]$ и т.д. Каждая перестановка помещает хотя бы одну запись в нужном порядке. Поэтому данный алгоритм сортировки элементов массива A на месте имеет время выполнения порядка $O(n)$.

```
for i:= 1 to n do
    while A[i].key <> i do
        swap(A[i], A[A[i].key]);
```

в общем случае мы должны быть готовы к тому, что в одном "кармане" может храниться несколько записей, а также должны уметь объединять содержимое нескольких "карманов" в один, располагая элементы в объединенном "кармане" в правильном порядке.

Задание.

1. **Сортировка простыми вставками.** Упорядочить массив *R(l)* по невозрастанию, используя следующий подход: для $i = 2,3,...,l$ каждый элемент r_i вставлять в нужное место среди упорядоченных ранее элементов $r_1, r_2, ..., r_{i-1}$, раздвигая их за счет удаления r_i.

2. **Сортировка бинарными вставками.** Как в предыдущей задаче, каждый из элементов r_i вставлять в нужное место ранее упорядоченной цепочки $r_1, r_2, ..., r_{i-1}$ используя для поиска места метод дихотомии (метод деления отрезка пополам).

3. **Сортировка слиянием.** Упорядочить по неубыванию массив A(n), разбивая его на группы по 1,2,4,8,... элементов и проводя попарное слияние соседних групп (число *n* не обязательно равно степени 2). Разрешается выделить дополнительный рабочий массив *B(n)* и проводить слияние поочередно из A в B, затем — из *B* в *A* и так далее.

4. В неупорядоченном массиве *K(m)* есть совпадающие элементы. Из каждой группы одинаковых элементов оставить только один, удалив остальные и поджав массив к его началу.

5. Турнирная таблица представлена квадратной матрицей *A(n, n)*, каждый элемент a_{ij} которой есть число голов, забитых *i*-й командой в ворота j- й команды. По диагонали расположить место каждой команды (по числу побед за вычетом числа поражений; в случае равенства — по суммарной разности забитых и пропущенных голов).

6. Упорядоченный по невозрастанию массив *B(n)* преобразовать в упорядоченный по возрастанию, оставив по одному в каждой группе совпадающих элементов.

7. Даны два упорядоченных по возрастанию массива *A(m)* и *B(n)*. Получить из них путем слияния упорядоченный по возрастанию массив *C*; совпадающие элементы вставлять единожды. Подсчитать количество элементов в массиве *C*.

8. Из двух упорядоченных по невозрастанию массивов *A(m)* и *B(n)* получить путем слияния упорядоченный по убыванию массив *C*; удаляемые элементы собрать в массиве *D*. Подсчитать количество элементов в массивах *C* и *D*.

9. Произвести слияние упорядоченного по возрастанию *A(m)* и неупорядоченного *B(n)* массивов (n<<m) в упорядоченный по неубыванию массив *C*.

10. Путем слияния из возрастающего *A(m)* и невозрастающего *B(n)* массивов получить возрастающий массив *C* (с удалением совпадающих элементов). Подсчитать количество элементов в массиве *C*.

11. Упорядочить строки матрицы *A(m, n)* лексикографически по убыванию.

12. Упорядочить строки матрицы *B(m, n)* по неубыванию элементов ее *k-го* столбца.

13. Упорядочить строки матрицы *D(m, n)* лексикографически по неубыванию первых *k* элементов строки.

14. Провести построчное слияние двух матриц *A(m, n)* и *B(k, n)*, строки которых лексикографически упорядочены по неубыванию первых *l* элементов каждой строки.

15. Произвести построчное слияние двух матриц *A(m, n)* и *B(k, n)*, упорядоченных по неубыванию элементов первого столбца.

16. Матрица *K(m, n)* упорядочена по возрастанию элементов первого столбца. С внешнего устройства (с клавиатуры, из файла и так далее) поступают дополнительные строки. Вставлять их в нужное место, раздвигая остальные, с сохранением упорядоченности. При невозможности такой вставки (первые элементы строк совпадают, а остальные — нет) вводимую строку вывести на печать с сообщением о невозможности вставки. Результат - пополненная матрица.

17. Упорядочить по неубыванию каждую строку матрицы *A(m, n)*, а после этого перестановкой строк упорядочить всю матрицу по неубыванию элементов первого столбца.

18. Возрастающий массив *A(n)* хранится в памяти. С устройства ввода (из файла или с клавиатуры) поступают элементы неупорядоченного массива *B*. По мере ввода элементов массива *B* производить слияние этих массивов в массив *A*, не выделяя дополнительного места под массив *B*. При совпадении элементов (невозможности вставки без нарушения признака порядка) выводить их на печать с сообщением. Работу прекратить при исчерпании входных данных или при заполнении всей памяти, отведенной под массив *A*.

19. Строки матрицы *A(m, n)* в произвольном порядке поступают с устройства ввода (из файла или с клавиатуры). Располагать их по мере поступления в выделенном под нее массиве лексикографически по возрастанию; совпадающие строки вставлять единожды.

20. Элементы массива *D(n)* случайным образом перемешаны. Элементы k_i массива *K(n)* указывают номера позиций, которые занимали соответствующие элементы d_i до перемешивания. Восстановить исходное состояние массива *D*.

21. Матрица *L(m, n)* состоит из нулей и единиц. Удалить из нее совпадающие строки, а оставшиеся упорядочить по возрастанию двоичных чисел, образуемых строками. Число *n* больше разрядности компьютера.

22. В начале каждой строки частично заполненной , матрицы *A(m, n)* сгруппированы элементы, упорядоченные по возрастанию. В массиве *K(m)* указано количество элементов в каждой строке. Слить все строки матрицы *A* в одномерный неубывающий массив *B*.

23. Массив $X(n)$ разбит на m фрагментов. В целочисленном массиве $K(m)$ хранятся длины соответствующих фрагментов (все k_i различны, их сумма равна n). Упорядочить массив K по возрастанию, переставив соответствующие фрагменты в массиве X.

24. Строки и столбцы верхнетреугольной матрицы, не содержащей нулей на диагонали и над ней, случайным образом перемешаны, первоначальные номера строк и столбцов не сохранены. Перестановкой строк и столбцов восстановить исходный вид матрицы.

25. В результате эксперимента получены наборы значений аргумента x и соответствующих значений функции y. Сформировать и напечатать таблицу значений функции, упорядочив их по возрастанию x. Если одному значению x соответствует несколько значений y, взять их среднее значение.

26. В результате эксперимента получено n пар значений величин x и y в массивах $X(n)$ и $Y(n)$. Массив X не упорядочен, но не содержит одинаковых элементов. Построить таблицу значений функции $y(x)$ с постоянным шагом $h = (x_{max} - x_{min})/(n-1)$, где x_{max} и x_{min} - соответственно наибольшее и наименьшее значения в массиве X. Значения y вычислять по формуле линейной интерполяции соседних наблюдений (после упорядочения по x): $y = y_{i-1} + \frac{x - x_{i-1}}{h}(y_i - y_{i-1})$.

27. В результате m экспериментов разной продолжительности получены наборы значений функций $y^{(k)} = y^{(k)}(x)$, $k = 1, 2, \ldots, m$ с соответствующими значениями аргумента—массивы $X^{(1)}(n_1)$, $Y^{(1)}(n_1)$, $X^{(2)}(n_2)$, $Y^{(2)}(n_2)$,...; $X^{(m)}(n_m)$, $Y^{(m)}(n_m)$. Многие значения x в разных экспериментах совпадают. Получить совокупную таблицу значений всех функций, слив упорядоченные массивы $X^{(1)}$, $X^{(2)}$, ..., $X^{(m)}$. Отсутствующие наблюдения функций $y^{(k)}$ печатать пробелами или другими символами.

28. Произвести слияние двух заданных упорядоченных по возрастанию списков в один неубывающий список.

29. В заданном неупорядоченном списке оставить по одному в каждой группе совпадающих элементов, сохранив порядок следования остальных. Пополнить упорядоченный по возрастанию список A элементами неупорядоченного списка B, сохранив упорядоченность (совпадающие элементы включать единожды).

Лабораторная работа - 6.

Тема: Программная реализация рекурсивных процедур и рекуррентных соотношений.

Цель работы: изучение понятия рекурсии и оценки рекурсивных алгоритмов путём решения рекуррентных соотношений.

В результате выполнения лабораторной работы студенты должны:

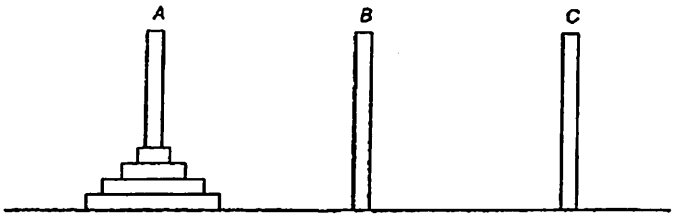
- *знать* основные понятия, связанные с рекурсивными алгоритмами;
- *уметь* реализовывать эти алгоритмы средствами языков программирования и оценивать их сложность.

6.1. Метод получения оценки решения рекуррентных соотношений

Возможно, самым важным и наиболее широко применимым методом проектирования эффективных алгоритмов является метод, называемый *методом декомпозиции* (или метод "разделяй и властвуй", или метод разбиения). Этот метод предполагает такую декомпозицию (разбиение) задачи размера n на более мелкие задачи, что на основе решений этих более мелких задач можно легко получить решение исходной задачи. Мы уже знакомы с рядом применений этого метода, например в сортировке слиянием или в деревьях двоичного поиска.

Чтобы проиллюстрировать этот метод, рассмотрим хорошо известную головоломку "Ханойские башни". Имеются три стержня *A, B* и C. Вначале на стержень *A* нанизаны несколько дисков: диск наибольшего диаметра находится внизу, а выше — диски последовательно уменьшающегося диаметра. Цель головоломки — перемещать диски (по одному) со стержня на стержень так, чтобы диск большего диаметра никогда не размещался выше диска меньшего диаметра и чтобы в конце концов все диски оказались нанизанными на стержень *B*. Стержень *C* можно использовать для временного хранения дисков.

Для решения этой головоломки подходит следующий простой алгоритм. Представьте, что стержни являются вершинами треугольника. Пронумеруем все перемещения дисков. Тогда, при перемещениях с нечетными номерами, наименьший диск нужно перемещать в треугольнике на соседний стержень по часовой стрелке. При перемещениях с четными номерами выполняются другие допустимые перемещения, не связанные с наименьшим диском.



Исходное положение в головоломке "Ханойские башни"

Описанный выше алгоритм, конечно, правильный и, к тому же, весьма лаконичный, правда, нелегко понять, почему он "работает", да и вряд ли такой алгоритм быстро придет вам в голову. Попробуем вместо этого применить метод декомпозиции. Задачу перемещения *n* наименьших дисков со стержня *A* на стержень *B* можно представить себе состоящей из двух подзадач размера *n* — *1*. Сначала нужно переместить *n* — 1 наименьших дисков со стержня *A* на стержень С, оставив на стержне *A n-й* наибольший диск. Затем этот диск нужно переместить с А на Б. Потом следует переместить *n* - 1 дисков со стержня *C* на стержень *B*. Это перемещение *n* - 1 дисков выполняется путем рекурсивного применения указанного метода. Поскольку диски, участвующие в перемещениях, по размеру меньше тех, которые в перемещении не участвуют, не нужно задумываться над тем, что находится под перемещаемыми дисками на стержнях *A*, *B* или *C*. Хотя фактическое перемещение отдельных дисков не столь очевидно, а моделирование вручную выполнить непросто из-за образования стеков рекурсивных вызовов, с концептуальной точки зрения этот алгоритм все же довольно прост для понимания и доказательства его правильности (а если говорить о быстроте разработки, то ему вообще нет равных). Именно легкость разработки алгоритмов по методу декомпозиции обусловила огромную популярность этого метода; к тому же, во многих случаях эти алгоритмы оказываются более эффективными, чем алгоритмы, разработанные традиционными методами.

Анализ рекурсивных программ значительно сложнее и, как правило, требует решения дифференциальных уравнений. Мы будем использовать другие методы, которые похожи на методы решения дифференциальных уравнений, и даже позаимствуем их терминологию.

Сначала рассмотрим пример процедуры сортировки. Эта процедура-функция *mergesort* (сортировка слиянием) в качестве входных данных использует список элементов длиной *n* и возвращает этот список отсортированным (выходные данные). Эта функция использует также процедуру *merge* (слияние), у которой входными данными являются два отсортированных списка L_1 и L_2. Процедура *merge*(L_1,L_2) просматривает эти списки поэлементно, начиная с наибольших элементов. На каждом шаге наибольший элемент из двух сравниваемых (наибольшие элементы из списков L_1 и L_2) удаляется из своего списка и помещается в выходные данные. В

результате получается один отсортированный список, содержащий все элементы из списков L_1 и L_2. Детали выполнения процедуры *merge* сейчас для нас не имеют значения. Сейчас для нас важно только то, что процедура *merge* на списках длиной *n/2* выполняется за время порядка *O(n)*.

Рекурсивная процедура сортировки слиянием

```
function mergesort ( L: LIST; n: integer ): LIST{ L—список типа LIST длиной n.
Предполагается, что n является степенью числа 2 }
var
L1,  L2: LIST;
      begin
if n = 1 then
return (L) ; else begin
разбиение  L на две части  L1 и L2, каждая длиной  n/2;
return (merge(mergesort( L1,n/2),(mergesort (L2, n/2)));
end
      end;  { mergesort }
```

Обозначим через *T(n)* время выполнения процедуры *mergesort* в самом худшем случае. Анализируя приведенный листинг, можно записать следующее рекуррентное неравенство, ограничивающее сверху *T(n)*:

$$T(n) \le \begin{cases} c_1, & \text{если } n = 1, \\ 2T(n/2) + c_2 n, & \text{если } n > 1. \end{cases} \qquad (6.1)$$

В неравенствах (6.1) константа c_1 соответствует фиксированному количеству шагов, выполняемых алгоритмом над списком *L* длиной 1. Если $n > 1$, время выполнения процедуры *mergesort* можно разбить на две части. Первая часть состоит из проверки того, что $n \neq 1$, разбиения списка *L* на две равные части и вызова процедуры *merge*. Эти три операции требуют или фиксированного времени (проверка $n \neq 1$), или пропорционального *n* — разбиение списка *L* и выполнение процедуры *merge*. Поэтому можно выбрать такую константу c_2, что время выполнения этой части процедуры *mergesort* будет ограничено величиной $c_2 n$. Вторая часть процедуры *mergesort* состоит из двух рекурсивных вызовов этой процедуры для списков длины *n/2*, которые требуют времени *2T(n/2)*. Отсюда получаем второе неравенство (6.1).

Отметим, что соотношение (6.1) применимо только тогда, когда *n* четно. Следовательно, формулу для верхней границы *T(n)* в *замкнутой форме* (т.е. в таком виде, когда формула для *T(n)* не включает никаких выражений *T(m)* для $m < n$) можно получить только в случае, если *n* является степенью числа 2. Но если известна формула для *T(n)* тогда, когда *n* является степенью числа *2*, то можно оценить *T(n)* для любых *n*. Например, для практически всех алгоритмов можно предполагать, что значение *T(n)* заключено между $T(2^i)$ и $T(2^{i+1})$, если *n* лежит между 2^i и 2^{i+1}. Более того, нетрудно показать, что в (6.1) выражение *2T(n)* можно заменить на *T((n*

+1)/2)+$T((n-1)/2)$ для нечетных $n > 1$. Таким образом можно найти решение рекуррентного соотношения в замкнутой форме для любых n.

Существуют три различных подхода к решению рекуррентных соотношений.

- Нахождение функции $f(n)$, которая мажорировала бы $T(n)$ для всех значений n (т.е. для всех $n \geq 1$ должно выполняться неравенство $T(n) \leq f(n)$). Иногда сначала мы будем определять только вид функции $f(n)$, предполагая, что она зависит от некоторых пока неопределенных параметров (например, $f(n) = an^2$, где a — неопределенный параметр), затем подбираются такие значения параметров, чтобы для всех значений n выполнялось неравенство $T(n) \leq f(n)$.
- В рекуррентном соотношении в правую часть последовательно подставляются выражения для $T(m)$, $m < n$, так, чтобы исключить из правой части все выражения $T(m)$ для $m > 1$, оставляя только $T(1)$. Поскольку $T(1)$ всегда является константой, то в результате получим формулу для $T(n)$, содержащую только n и константы. Такая формула и называется "замкнутой формой" для $T(n)$.
- Третий подход заключается в использовании общих решений определенных рекуррентных соотношений.

Рассмотрим первый описанный выше подход на примере соотношения (6.1). Предположим, что $T(n) = an \log n$, где a — пока не определенный параметр. Подставляя $n = 1$, видим, что эта оценка "не работает", так как при $n = 1$ выражение $an \log n$ равно 0, независимо от значения a. Попробуем применить другую функцию: $T(n) = an \log n + b$. При $n = 1$ эта оценка "работает", если положить $b \geq c_1$.

В соответствии с методом математической индукции предполагаем, что для всех $k < n$ выполняется неравенство

$$T(k) \leq ak \log k + b, \qquad (6.2)$$

и попытаемся доказать, что $T(n) \leq an \log n + b$.

Пусть $n > 2$. Из неравенств (6.1) имеем $T(n) \leq 2T(n/2) + c_2 n$. Полагая $k = n/2$, используем в последнем неравенстве оценку (6.2). Получаем

$$T(n) \leq 2\left(a\frac{n}{2}\log\frac{n}{2} + b\right) + c_2 n =$$

$$= an \log n - an + c_2 n + 2b \leq an \log n + b. \qquad (6.3)$$

Последнее неравенство получено в предположении, что $a \geq c_2 + b$.

Таким образом, видим, что будет справедлива оценка $T(n) \leq an \log n + b$, если будут выполняться неравенства $b \geq c_1$ и $a \geq c_2 + b$. В данном случае можно удовлетворить этим неравенствам, если положить $b = c_1$ и $a = c_1 + c_2$. Отсюда мы заключаем, что для всех $n > 1$ выполняется неравенство

$$T(n) \leq (c_1 + c_2) n \log n + c_1. \qquad (6.4)$$

Другими словами, $T(n)$ имеет порядок $O(n \log n)$.

Исходя из рассмотренного примера, сделаем два замечания. Во-первых, если мы предполагаем, что $T(n)$ имеет порядок $O(f(n))$, но по индукции мы не можем доказать неравенства $T(n) \leq c f(n)$, то это еще не значит, что $T(n) \neq$

$O(f(n))$. Возможно, надо просто добавить константу к функции $c\,f(n)$, например можно попытаться доказать неравенство $T(n) \le cf(n) - 1$!

Во-вторых, мы не определили точной асимптотической степени роста оценочной функции $f(n)$, мы только показали, что она растет не быстрее, чем $O(n \log n)$. Если мы возьмем в качестве оценки более медленно растущие функции, например, такие как $f(n) = an$ или $f(n) = an\log\log n$, то не сможем доказать, что $T(n) \le f(n)$. Но в чем причина этого: неверный метод доказательства, или для данного алгоритма в принципе невозможна оценочная функция с меньшей степенью роста? Чтобы ответить на этот вопрос, надо подробнее проанализировать исследуемый алгоритм и рассмотреть время выполнения в самом худшем случае. Для нашей процедуры *mergesort* надо показать, что действительно время выполнения равно $\Omega(n \log n)$. Фактически мы показали, что время выполнения процедуры не превосходит $cn \log n$ для любых входных данных, но не для "самых плохих" входных данных. Случай возможных "самых плохих" входных данных мы оставляем в качестве упражнения.

Метод получения оценки решения рекуррентных соотношений, проиллюстрированный в примере, можно обобщить на некоторые другие функции, ограничивающие сверху время выполнения алгоритмов. Пусть имеется рекуррентное соотношение

$$T(1) = c,$$
$$T(n) \le g(T(n/2), n) \text{ для } n > 1. \qquad (6.5)$$

Отметим, что соотношение (6.5) обобщает соотношение (1), которое получается из (6.5), если положить $g(x, y) = 2x + c_2 y$. Конечно, возможны еще более общие соотношения, чем (6.5). Например, функция g может включать в себя все значения $T(n-1), T(n-2), \ldots, T(1)$, а не только $T(n/2)$. Также могут быть заданы значения для $T(1), T(2), \ldots, T(k)$ и рекуррентное соотношение, которое применимо только для $n > k$. Читатель может самостоятельно попробовать применить к этим обобщенным рекуррентным соотношениям описанный выше метод оценки решения этих соотношений.

Вернемся к соотношению (6.5). Предположим, что выбранная оценочная функция $f(a_1, \ldots, a_j, n)$ зависит от параметров $a_1 \ldots a_j$, нам надо доказать индукцией по n, что $T(n) \le f(a_1, \ldots, a_j, n)$. (В примере $f(a_1, \ldots, a_j, n) = a_1 n \log n + a_2$, параметры a_1 и a_2 обозначены как a и b.) Для того чтобы оценка $T(n) \le f(a_1, \ldots, a_j, n)$ была справедливой для всех $n \ge 1$, надо найти такие значения параметров $a_1, \ldots, a_j$, чтобы выполнялись неравенства

$$f(a_1, \ldots, a_j, 1) \ge c,$$
$$f(a_1, \ldots, a_j, n) \ge g(f(a_1, \ldots, a_j, n/2), n). \qquad (6.6)$$

В соответствии с методом математической индукции можно подставить функцию f вместо T в правую часть неравенства (6.5). Получаем

$$T(n) \le g(f(a_1, \ldots, a_j, n/2), n). \qquad (6.7)$$

Если неравенство (6.6) выполняется (уже подобраны соответствующие значения параметров), то, применив его в (6.7), будем иметь то, что требуется доказать: $T(n) \le f(a_1,..., a_i, n)$.

В примере мы имели $g(x, y)=2x + c_2y$ и $f(a_1,..., a_i, n)= a_1 n \log n + a_2$. Параметры a_1 и a_2 надо подобрать так, чтобы выполнялись неравенства

$$f(a_1, a_2, 1) \ge c_1,$$

$$f(a_1, a_2, n) = a_1 n \log n - a_2 \ge 2\left(a_1 \frac{n}{2}\log\frac{n}{2} + a_2\right) - c_2 n.$$

Для этого достаточно положить $a_2 = c_1$ и $a_1 = c_1 + c_2$.

6.2. Оценка решения рекуррентного соотношения методом подстановки

Если мы не знаем вида оценочной функции или не уверены в том, что выбранная оценочная функция будет наилучшей границей для T(n), то можно применить подход, который в принципе всегда позволяет получить точное решение для T(n), хотя на практике он часто приводит к решению в виде достаточно сложных сумм, от которых не всегда удается освободиться. Рассмотрим этот подход на примере рекуррентного соотношения (6.1). Заменим в этом соотношении n на $n/2$, получим

$$T(n/2) \le 2T(n/4) + c_2 n/2. \qquad (6.8)$$

Подставляя правую часть неравенства (6.8) вместо $T(n/2)$ в неравенство (6.1), будем иметь

$$T(n) \le 2(2T(n/4) + c_2 n/2) + c_2 n = 4T(n/4) + 2c_2 n. \qquad (6.9)$$

Аналогично, заменяя в неравенстве (6.1) n на $n/4$, получаем оценку для $T(n/4)$: $T(n/4) \le 2T(n/8) + c_2 n/4$. Подставляя эту оценку в неравенство (6.9), получаем

$$T(n) \le 8T(n/8) + 3c_2 n. \qquad (6.10)$$

Надеемся, читатель понял принцип подстановки. Индукцией по i для любого i можно получить следующее соотношение:

$$T(n) \le 2^i T(n/2^i) + ic_2 n. \qquad (6.11)$$

Предположим, что n является степенью числа 2, например $n = 2^k$. Тогда при $i = k$ в правой части неравенства (6.11) будет стоять $T(1)$:

$$T(n) \le 2^k T(1) + kc_2 n. \qquad (6.12)$$

Поскольку $n = 2^k$, то $k = \log n$. Так как T(1) $\le c_1$, то из (6.12) следует, что

$$T(n) \le c_1 n + c_2 n \log n. \qquad (6.13)$$

Неравенство (6.13) показывает верхнюю границу для $T(n)$, это и доказывает, что $T(n)$ имеет порядок роста не более $O(n \log n)$.

Задания.

1. Запишите рекуррентное соотношение для времени выполнения следующего алгоритма, предполагая, что n является степенью числа 2.

function *path* (*s,t,n*:integer): boolean;

```
begin
if n = 1 then
if edge(s, t) then
return(true) else
return(false); for i:= 1 to n do
if path(s,i,n div 2) and path(i,t,n div 2)
  then return(true);
return(false)
end; { path }
```

Функция *edge(i, j)* возвращает значение true, если вершины i и j в графе с n вершинами соединены ребром либо $i = j$, и значение false — в противном случае. Что делает функция *path* (путь)?

2. Решите следующие рекуррентные уравнения, если $T(1) = 1$.

а) $T(n) = 3T(n/2) + n$;

б) $T(n) = 3T(n/2) + n^2$;

в) $T(n) = 8T(n/2) + n^3$.

3. Решите следующие рекуррентные уравнения, если $T(1) = 1$.

а) $T(n) = 4T(n/2) + n$;

б) $T(n) = 4T(n/2) + n^2$;

в) $T(n) = 9T(n/2) + n^3$.

4. Найдите верхнюю оценку для $T(n)$, удовлетворяющих следующим рекуррентным уравнениям и предположению, что $T(1) = 1$.

а) $T(n) = T(n/2) + 1$;

б) $T(n) = 2\,T(n/2) + \log n$;

в) $T(n) = 2\,T(n/2) + n$;

г) $T(n) = 2T(n/2) + n^2$.

5. Найдите верхнюю оценку для $T(n)$, удовлетворяющих следующим рекуррентным соотношениям:

а) $T(1) = 2$,
$T(n) = 2\,T(n - 1) + 1$ при $n \geq 2$.

б) $T(1) = 1$,
$T(n) = 2\,T(n - 1) + n$ при $n \geq 2$.

6. Проверьте ответы упражнения 5, решив рекуррентные соотношения методом подстановок.

7. Обобщите упражнение 6 для решения произвольных рекуррентных уравнений вида

$$T(1) = 1,$$
$$T(n) = aT(n - 1) + d(n) \text{ при } n \geq 2$$

в терминах параметра a и функции $d(n)$.

8. В упражнении 9.7 положите $d(n) = c^n$ для некоторой константы $c > 1$. Как в этом случае решение $T(n)$ будет зависеть от соотношения a и c? Какой вид решения $T(n)$?

9. Найдите $T(n)$, если

$$T(1)=1,$$

$$T(n) = \sqrt{n}\,T(\sqrt{n}) + n \text{ при } n \geq 2.$$

10. Найдите замкнутые выражения для следующих сумм:

а) $\sum_{i=0}^{n} i$, б) $\sum_{i=0}^{n} t^k$, в) $\sum_{i=0}^{n} 2^i$, г) $\sum_{i=0}^{n} C_n^i$.

11. Покажите, что число различных порядков, в соответствии с которыми можно перемножить последовательность из n матриц, удовлетворяет следующим рекуррентным соотношениям:

$$T(1)=1,$$

$$T(n) = \sum_{i=1}^{n} T(i)T(n-i)$$

Докажите, что $T(n+1) = \frac{1}{n+1} C_{2n}^{n}$. Числа T($n$) называются *числами Каталана.*

12. Покажите, что число сравнений $T(n)$, необходимое для сортировки n элементов посредством метода сортировки слиянием, удовлетворяет соотношениям

$$T(1)=0,$$

$$T(n) = T([n/2]) + T(\lceil n/2 \rceil) + n - 1,$$

где $[x]$ обозначает целую часть числа x, а $\lceil x \rceil$— наименьшее целое, большее или равное x. Докажите, что решение этих рекуррентных соотношений имеет вид

$$T(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1.$$

13. Покажите, что число булевых функций n переменных удовлетворяет рекуррентным соотношениям

$$T(1) = 4,$$

$$T(n) = (T(n-1))^2.$$

Найдите $T(n)$.

14. Покажите, что число двоичных деревьев высотой не более n удовлетворяет рекуррентным соотношениям

$$T(1)=1,$$

$$T(n) = (T(n-1))^2 + 1.$$

Докажите, что $T(n) = \lfloor k^{2^n} \rfloor$ для некоторой константы k. Каково значение k?

Лабораторная работа - 7.

Тема: Программная реализация алгоритмов «Разделяй и властвуй», динамическое программирование, «жадных» алгоритмов и поиска.

Цель работы: изучение основных методов разработки алгоритмов и оценки их эффективности.

В результате выполнения лабораторной работы студенты должны:

- *знать* основные методы разработки алгоритмов, такие как методы декомпозиции, динамического программирования, поиска с возвратом, «жадные» алгоритмы;
- *уметь* реализовывать эти алгоритмы средствами языков программирования.

К настоящему времени специалисты по вычислительной технике разработали ряд эффективных методов, которые нередко позволяют получать эффективные алгоритмы решения больших классов задач. Некоторые из наиболее важных методов: "разделяй и властвуй" (декомпозиция), динамическое программирование, "жадные" методы, поиск с возвратом и локальный поиск.

Возможно, самым важным и наиболее широко применимым методом проектирования эффективных алгоритмов является метод, называемый *методом декомпозиции* (или метод "разделяй и властвуй", или метод разбиения). Этот метод предполагает такую декомпозицию (разбиение) задачи размера n на более мелкие задачи, что на основе решений этих более мелких задач можно легко получить решение исходной задачи.

При использовании алгоритмов декомпозиции желательно, чтобы подзадачи были примерно одинакового размера. Например, сортировку вставками можно рассматривать как разбиение задачи на две подзадачи — одна размером 1, а другая — n-1, причем максимальные затраты на выполнение слияния равняются n шагам. В результате приходим к рекуррентному соотношению $T(n)= T(1) + T(n -1) + n$, которое имеет решение $O(n^2)$. В то же время сортировка слиянием разбивает задачу на две подзадачи, каждая размером $n/2$, а ее эффективность равняется $O(n\ logn)$. Складывается впечатление, что разбиение задачи на равные (или примерно равные) подзадачи является важным фактором обеспечения высокой эффективности алгоритмов.

7.1. Динамическое программирование

Нередко не удается разбить задачу на небольшое число подзадач, объединение решений которых позволяет получить решение исходной задачи. В таких случаях мы можем попытаться разделить задачу на столько подзадач, сколько необходимо, затем каждую подзадачу разделить на еще более мелкие подзадачи и т.д. Если бы весь алгоритм сводился именно к такой

последовательности действий, мы бы получили в результате алгоритм с экспоненциальным временем выполнения.

Но зачастую удается получить лишь полиномиальное число подзадач и поэтому ту или иную подзадачу приходится решать многократно. Если бы вместо этого мы отслеживали решения каждой решенной подзадачи и просто отыскивали в случае необходимости соответствующее решение, мы бы получили алгоритм с полиномиальным временем выполнения.

С точки зрения реализации иногда бывает проще создать таблицу решений всех подзадач, которые нам когда-либо придется решать. Мы заполняем эту таблицу независимо от того, нужна ли нам на самом деле конкретная подзадача для получения общего решения. Заполнение таблицы подзадач для получения решения определенной задачи получило название *динамического программирования* (это название происходит из теории управления).

7.2. „Жадные" алгоритмы

Рассмотрим небольшую "детскую" задачу. Допустим, что у нас есть монеты достоинством 25, 10, 5 копеек и 1 копейка и нужно вернуть сдачу 63 копейки. Почти не раздумывая, мы преобразуем эту величину в две монеты по 25 копеек, одну монету в 10 копеек и три монеты по одной копейке. Нам не только удалось быстро определить перечень монет нужного достоинства, но и, по сути, мы составили самый короткий список монет требуемого достоинства.

Алгоритм, которым читатель в этом случае наверняка воспользовался, заключался в выборе монеты самого большого достоинства (25 копеек), но не больше 63 копеек, добавлению ее в список сдачи и вычитанию ее стоимости из 63 (получается 38 копеек). Затем снова выбираем монету самого большого достоинства, но не больше остатка (38 копеек): этой монетой опять оказывается монета в 25 копеек. Эту монету мы опять добавляем в список сдачи, вычитаем ее стоимость из остатка и т.д.

Этот метод внесения изменений называется *"жадным" алгоритмом.* На каждой отдельной стадии "жадный" алгоритм выбирает тот вариант, который является *локально оптимальным* в том или ином смысле. Обратите внимание, что алгоритм для определения сдачи обеспечивает в целом оптимальное решение лишь вследствие особых свойств монет. Если бы у нас были монеты достоинством 1 копейка, 5 и 11 копеек и нужно было бы дать сдачу 15 копеек, то "жадный" алгоритм выбрал бы сначала монету достоинством 11 копеек, а затем четыре монеты по одной копейке, т.е. всего пять монет. Однако в данном случае можно было бы обойтись тремя монетами по 5 копеек.

7.3. Поиск с возвратом

Иногда приходится иметь дело с задачей поиска оптимального решения, когда невозможно применить ни один из известных методов, способных помочь отыскать оптимальный вариант решения, и остается прибегнуть к последнему средству — полному перебору.

Допустим, что заданы правила некоторой игры. Мы хотим построить дерево этой игры и оценить его корень. Объем памяти, который требуется для хранения такого дерева, может оказаться недопустимо большим, но, если соблюдать определенные меры предосторожности, можно обойтись хранением в памяти в любой заданный момент времени лишь одного пути — от корня к тому или иному узлу. В листинге показана рекурсивная программа *search* (поиск), которая выполняет обход дерева с помощью последовательности рекурсивных вызовов этой процедуры. Эта программа предполагает выполнение следующих условий.

1. Выигрыши являются действительными числами из конечного интервала, например от -1 до +1.
2. Константа ∞ больше, чем любой положительный выигрыш, а -∞ — меньше, чем любой отрицательный выигрыш.
3. Тип данных modetype (тип режима) определяется следующим образом:

```
type
    modetype = (MIN, MAX)
```

4. Предусмотрен тип данных boardtype (тип игровой доски), который определяется способом, подходящим для представления позиций на игровой доске.
5. Предусмотрена функция *payoff* (выигрыш), которая вычисляет выигрыш для любой позиции, которая является листом.

Рекурсивная программа поиска с возвратом

```
function search { B: boardtype; mode: modetype): real;
                { оценивает и возвращает выигрыш для позиции B в
                предположении, что следующим должен ходить игрок 1 (mode =
                MAX) или игрок 2 (mode = MIN) }
    var
                C: boardtype; { сын позиции B }
                value: real; { для временного хранения минимального или
                                максимального значения }

    begin
(1)             if B является листом then
(2)                 return(payoff(B))
                else begin
(3)                 if mode = MAX then
(4)                     value:= -∞
                    else
(5)                     value: = ∞;
(6)                   for для каждого сына C позиции B do
(7)                       if mode = MAX then
(8)                         value:=max(value, search(C, MIN))
                          else
(9)                         value:=min(value, search(C, MAX));
```

```
(10)                    return(value)
    end
        end; { search }
```

7.4. Алгоритмы локального поиска

Описанная ниже стратегия нередко приводит к оптимальному решению задачи.

- Начните с произвольного решения.
- Для улучшения текущего решения примените к нему какое-либо преобразование из некоторой заданной совокупности преобразований. Это улучшенное решение становится новым "текущим" решением.
- Повторяйте указанную процедуру до тех пор, пока ни одно из преобразований в заданной их совокупности не позволит улучшить текущее решение.

Результирующее решение может, хотя и необязательно, оказаться оптимальным. В принципе, если "заданная совокупность преобразований" включает все преобразования, которые берут в качестве исходного одно решение и заменяют его каким-либо другим, процесс "улучшений" не закончится до тех пор, пока мы не получим оптимальное решение. Но в таком случае время выполнения пункта (2) окажется таким же, как и время, требующееся для анализа всех решений, поэтому описываемый подход в целом окажется достаточно бессмысленным.

Этот метод имеет смысл лишь в том случае, когда мы можем ограничить нашу совокупность преобразований небольшим ее подмножеством, что дает возможность выполнить все преобразования за относительно короткое время: если "размер" задачи равняется л, то мы можем допустить $O(n^2)$ или $O(n^3)$ преобразований. Если совокупность преобразований невелика, естественно рассматривать решения, которые можно преобразовывать одно в другое за один шаг, как "близкие". Такие преобразования называются "локальными", а соответствующий метод называется *локальным поиском.*

Пример. **Алгоритм умножения целых чисел методом декомпозиции**

```
    function mult (X, Y, n: integer): integer;
        { X и Y — целые числа со знаком < 2^n. n — степень числа 2. Функция
            возвращает значение произведения XY }
        var
            s: integer; { содержит знак произведения XY }
            m1,m2, m3: integer; { содержат три произведения }
            A,B,C,D:integer;{содержат левые и правые половины X и Y}
        begin
(1)         s:= sign(X) * sign(Y);
(2) X:= abs(X);
(3) Y:=abs( Y) ;  {теперь X и Y — положительные числа}
```

```
(4)if n = 1 then
(5)if (X=1) and (Y=1) then
(6)return (s)
                    else
(7)                 return (0)
                       else begin
(8)                       A:= левые n/2 биты числа X;
(9)                       B:= правые n/2 биты числа X;
(10)                      C:= левые n/2 биты числа Y;
(11)                      D:= правые n/2 биты числа Y;
(13)                      m1:= mult(A,C, n/2);
(14)                      m2: = mult(A-B,  D-C,  n/2);
(15)                      m3:= mult(B, D, n/2);
(16)                      return (s*(m1*2^n + (m1 + m2 + m3)* 2^(n/2) + m3))
                       end
   end; { mult }
```

Задание.

1. Реализовать алгоритм разбора алгебраического выражения с использованием рекурсии.

2. Заданы вершины многоугольника и расстояния между каждой парой вершин. Это расстояние может быть обычным евклидовым расстоянием на плоскости или произвольной функцией стоимости, задаваемой в виде таблицы. Задача заключается в том, чтобы выбрать такую совокупность *хорд* (линий между несмежными вершинами), что никакие две хорды не будут пересекаться, а весь многоугольник будет поделен на треугольники. Общая длина выбранных хорд должна быть минимальной.

3. Даны N матриц, каждая из которых имеет размерность $m_i x n_i$, $1 \le i \le N$. Найти оптимальную с точки зрения количества необходимых операций умножения последовательность для перемножения этих матриц.

4. Дана последовательность чисел. Найти самую длинную монотонно неубывающую подпоследовательность данной последовательности (время работы алгоритма должно иметь порядок $O(n^2)$).

5. На плоскости задано n точек. Определить кратчайший замкнутый путь, соединяющий между собой все эти точки.

6. Имеется шахматная доска размером nxn и шашка, расположенная на первой горизонтали. Шашка может двигаться на соседнюю клетку, расположенную над текущей, либо на 1 клетку по диагонали вверх и влево, либо на 1 клетку по диагонали вверх и вправо. Шашка должна дойти до последней горизонтали. Каждый ход из клетки x в клетку y задаётся стоимостью p(x,y), которая может быть как положительной, так и отрицательной. Определить последовательность ходов с максимальной общей стоимостью.

7. Имеется компьютер и n заданий, каждому из которых соответствует время выполнения tj, стоимость pj и крайний срок выполнения dj. Задание не должно прерываться в течение всего времени выполнения. Если оно выполняется в срок, прибыль равна его стоимости, иначе – нулю. Определить последовательность выполнения заданий, чтобы прибыль была максимальной.

8. Требуется выдать сдачу размером n копеек с помощью набора монет k различных заданных номиналов. Время работы алгоритма должно иметь порядок O(kn).

9. Реализовать алгоритм нахождения минимального остовного дерева графа с использованием локального поиска.

10.Заданы вершины многоугольника и расстояния между каждой парой вершин. Это расстояние может быть обычным евклидовым расстоянием на плоскости или произвольной функцией стоимости, задаваемой в виде таблицы. Задача заключается в том, чтобы выбрать такую совокупность *хорд* (линий между несмежными вершинами), что никакие две хорды не будут пересекаться, а весь многоугольник будет поделен на треугольники. Общая длина выбранных хорд должна быть минимальной.

11.*Задачу разбиения на абзацы* в простейшей форме можно сформулировать следующим образом. Дана последовательность слов $w_1, w_2, \ldots, w_k$ длиной $l_1, l_2, \ldots, l_k$, которые нужно разбить на строки длиной L. Слова разделяются пробелами, стандартная ширина которых равна b, но пробелы при необходимости могут удлиняться или сжиматься (но без "наползания" слов друг на друга) так, чтобы длина строки $w_i, w_{i+1}, \ldots w_j$ равнялась в точности L. Однако *штрафом* за такое удлинение или сжатие пробелов является общая величина, на которую пробелы удлиняются или сжимаются, т.е. стоимость формирования строки $w_i, w_{i+1}, \ldots w_j$ при $j > i$ равна $(j - i)|b' - b|$, где b' — фактическая ширина пробелов, равная $(L - l_i - l_{i+1} - \ldots - l_j)/(j - i)$. При $j = k$ (речь идет о последней строке) эта стоимость равна нулю, если только b' не окажется меньше b, поскольку последнюю строку растягивать не требуется. На основе метода динамического программирования разработайте алгоритм для разделения с наименьшей стоимостью последовательности слов $w_1, w_2, \ldots, w_k$ на строки длиной L.

Допустим, есть n элементов $x_1, x_2, \ldots, x_n$, связанных линейным порядком $x_1 < x_2 < \ldots < x_n$, которое нужно представить в виде двоичного дерева поиска. Обозначим p_i вероятность запроса на поиск элемента x_i. Тогда для любого заданного двоичного дерева поиска средняя стоимость поиска составит $\sum_{i=1}^{n} p_i(d_i + 1)$, где d_i — глубина узла, содержащего x_i. При условии, что заданы все вероятности p_i и предполагая, что x_i никогда не изменяются, можно построить двоичное дерево поиска, минимизирующее стоимость поиска. На основе метода динамического программирования разработайте алгоритм, реализующий такое двоичное дерево поиска.

Лабораторная работа - 8.

Тема: Программная реализация алгоритмов внешней памяти.

Программная реализация управления памятью.

Цель работы: изучение основных алгоритмов для работы со внешней памятью и способов их реализации.

В результате выполнения лабораторной работы студенты должны:

- *знать* структуры данных, обеспечивающие доступ ко внешней памяти; алгоритмы управления внешней памятью;
- *уметь* писать и отлаживать эффективные программы, работающие со внешней памятью.

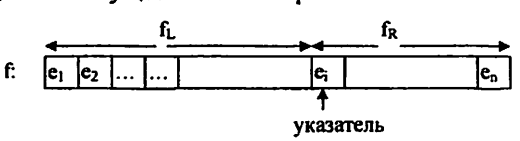
Оценивая время работы алгоритмов, в которых используются данные, хранящиеся в виде файлов, нам придется в первую очередь учитывать количество обращений к блокам, т.е. сколько раз мы считываем в основную память или записываем блок во вторичную память. Такая операция называется *доступом (или обращением) к блоку.*

Последовательным, файлом называется последовательность, над которой определены следующие пять операций:

а) формирование пустой последовательности *T( )*;
б) *first(x);*
в) *tail(x);*
г) *appendr(x,e);*
д) *empty(x).*

Последовательный файл представляет собой последовательность элементов, в которой допускается выборка (доступ) начального элемента последовательности и добавление элемента в конец последовательности. Такие последовательности реализуются на внешней запоминающей среде, например ленте, причем их запись возможна только в одном направлении. В языках программирования процедурного типа операция избирательного обновления является типичной операцией обработки.

Обработка файла реализуется путем введения файловой переменной *f*, а также дополнительной переменной типа T_0, которая называется *буферной переменной файла* и обозначается через $f^\wedge$. Она определяет текущее значение компоненты файла, являющейся объектом очередной операции обработки. Совокупность значений компонент файла и положение буферной переменной определяют текущее состояние файла.



Переменные f_L и f_R, такие, что $f = f_L \,\&\, f_R$, служат для указания уже просмотренной (f_L) и еще не просмотренной частей файла, соответствующего файловой переменной *f*. С их помощью, не используя указатель, можно задать текущее состояние файла в виде (f_L, f_R).

Основными операторами, используемыми для управления файлом, являются *rewrite, reset, write, read* и стандартная логическая функция *eof*. *Rewrite(f)* служит для формирования нового файла и соответствует заданию переменной *f* значения, равного пустому файлу. *Reset(f)* обеспечивает установку файла в начальную позицию. *Write(f,e)* переписывает в конец файла *f* значение переменной *e*. *Read(f,e)* считывает соответствующее положению указателя значение компоненты файла *f* в переменную *e*. Логическая функция *eof(f)* отражает факт достижения конца файла. В случае достижения конца файла *eof(f)* принимает значение **истина,** в противном случае ее значение есть **ложь.** Ниже дано пояснение семантики рассмотренных операторов управления файлом:

1. $rewrite(f) \Leftrightarrow f'_L = T(), f'_R = T()$.
2. $reset(f) \Leftrightarrow f'_L = T(), f'_R = f = concat(f_L, f_R)$.

(concat(x,y) обозначает последовательность, объединяющую две последовательности *x, y)*.

3. $write(f,e) \Leftrightarrow f'_L = appendr(f_L, e), f'_R = f_R = T()$.

Предполагается, что оператор *write(f,e)* выполним только в случае, когда *eof(f)* принимает значение **истина,** т. е. достигнут конец файла.

4. $read(f,e) \Leftrightarrow f'_L = appendr(f_L, first(f_R)), f'_R = tail(f_R), e' = first(f_R)$.

Выполнение *read(f,e)* возможно только в случае, когда *eof(f)* принимает значение **ложь.**

В приведенных выше выражениях переменные f_L, f_R и *e* справа от знака $\Leftrightarrow$ обозначают соответственно значения уже просмотренной, еще не просмотренной частей файла *f* и переменной *e* перед выполнением операторов, а f'_L, f'_R, e' обозначают значения тех же переменных после выполнения операторов. Другими словами, приведенные формулы определяют семантику операторов путем указания того, как изменяются значения f_L, f_R, *e* до и после выполнения операторов.

8.1. Сортировка слиянием

Сортировка данных, организованных в виде файлов, или — в более общем случае — сортировка данных, хранящихся во вторичной памяти, называется *внешней сортировкой.*

Главная идея, которая лежит в основе сортировки слиянием, заключается в том, что мы организуем файл в виде постепенно увеличивающихся *серий,* т.е. последовательностей записей $r_1, \ldots, r_k$, где ключ r_i не больше, чем ключ r_{i+1}, $1 \le i \le k$.[1] Мы говорим, что файл, состоящий из $r_1 \ldots r_m$ записей, *делится на серии длиной k,* если для всех $i \ge 0$, таких, что $ki \le m$ и $r_{k(i-1)+1}, r_{k(i-1)+2}, \ldots, r_{ki}$ является последовательностью длиной *k.* Если *m* не делится нацело на *k,* т.е. *m*

$= pk + q$, где $q < k$, тогда последовательность записей r_{m-q+1}, $r_{k(i-1)+2}$, ... , r_m, называемая *хвостом*, представляет собой серию длиной q. Например, последовательность целых чисел, показанная на рис. 11.1, организована сериями длиной 3. Обратите внимание, что хвост имеет длину, меньшую 3, однако и его записи тоже отсортированы.

7 15 29	8 11 13	16 22 31	5 12

Файл с сериями длиной 3

Главное в сортировке файлов слиянием — начать с двух файлов, например f_1 и f_2, организованных в виде серий длиной k. Допустим, что (1) количества серий (включая хвосты) в f_1 и f_2 отличаются не больше, чем на единицу; (2) по крайней мере один из файлов f_1 или f_2 имеет хвост; (3) файл с хвостом имеет не меньше серий, чем другой файл.

В этом случае можно использовать достаточно простой процесс чтения по одной серии из файлов f_1 и f_2, слияние этих серий и присоединения результирующей серии длиной $2k$ к одному из двух файлов g_1 и g_2, организованных в виде серий длиной $2k$. Переключаясь между g_1 и g_2, можно добиться того, что эти файлы будут не только организованы в виде серий длиной $2k$, но будут также удовлетворять перечисленным выше условиям (1) - (3). Чтобы выяснить, выполняются ли условия (2) и (3), достаточно убедиться в том, что хвост серий f_1 и f_2 слился с последней из созданных серий (или, возможно, уже был ею).

Рассмотрим случай, когда "узким местом" является слияние, а не считывание или запись данных. Сделаем следующие предположения.

1. Мы объединяем серии, размеры которых намного превышают размеры блоков.
2. Существуют два входных и два выходных файла. Входные файлы хранятся на одном внешнем диске (или каком-то другом устройстве, подключенном к основной памяти одним каналом), а выходные файлы — на другом подобном устройстве с одним каналом.
3. Время считывания, записи и выбора для заполнения блока записей с наименьшими ключами среди двух серий, находящихся в данный момент в основной памяти, одинаково.

С учетом этих предположений рассмотрим класс стратегий слияния, которые предусматривают выделение в основной памяти нескольких входных буферов (место для хранения блока). В каждый момент времени какой-то из этих буферов будет содержать невыделенные для слияния записи из двух входных серий, причем одна из них будет находиться в состоянии считывания из входного файла. Два других буфера будут содержать выходные записи, т.е. выделенные записи в надлежащим образом объединенной последовательности. В каждый момент времени один из этих буферов находится в состоянии записи в один из выходных файлов, а другой заполняется записями, выбранными из входных буферов.

Выполняются (возможно, одновременно) следующие действия.

1. Считывание входного блока во входной буфер.
2. Заполнение одного из выходных буферов выбранными записями, т.е. записями с наименьшими ключами среди тех, которые в настоящий момент находятся во входном буфере.
3. Запись данных другого выходного буфера в один из двух формируемых выходных файлов.

Мы должны разработать такую стратегию выбора буферов для считывания, чтобы в начале каждого этапа (состоящего из описанных действий) b невыбранных записей с наименьшими ключами уже находились во входных буферах (b — количество записей, которые заполняют блок или буфер). Допустим, k_1 и k_2 — наибольшие ключи среди невыбранных записей в основной памяти из первой и второй серий соответственно. В таком случае в основной памяти должно быть по крайней мере b невыбранных записей, ключи которых не превосходят $\min(k_1, k_2)$.

8.2. Схема с шестью входными буферами

Для каждого файла предусмотрены три буфера. Каждый буфер рассчитан на b записей. Заштрихованная область представляет имеющиеся записи, ключи расположены по окружности (по часовой стрелке) в возрастающем порядке. В любой момент времени общее количество невыбранных записей равняется $4b$ (если только не рассматриваются записи, оставшиеся от объединяемых серий). Поначалу мы считываем в буферы первые два блока из каждой серии. Поскольку у нас всегда имеется $4b$ записей, а из одного файла может быть не более $3b$ записей, мы знаем, что имеется по крайней мере b записей из каждого файла. Если k_1 и k_2 — наибольшие имеющиеся ключи в двух данных сериях, должно быть b записеи с ключами, не большими, чем k_1, и b записей с ключами, не большими, чем k_2. Таким образом, имеются b записей с ключами, не большими, чем $\min(k_1, k_2)$.

Схема слияния с шестью входными буферами

8.3. Хранение данных в файлах

Файл мы будем рассматривать как последовательность записей, причем каждая запись состоит из одной и той же совокупности полей. Поля могут иметь либо *фиксированную длину* (заранее определенное количество байт),

либо *переменную.* Файлы с записями фиксированной длины широко используются в системах управления базами данных для хранения данных со сложной структурой. Файлы с записями переменной длины, как правило, используются для хранения текстовой информации; в языке Pascal такие файлы не предусмотрены. В этом разделе будем иметь дело с полями фиксированной длины; рассмотренные методы работы после определенной (несложной) модификации могут использоваться для работы с записями переменной длины.

Мы рассмотрим следующие операторы для работы с файлами.

- INSERT вставляет определенную запись в определенный файл.
- DELETE удаляет из определенного файла все записи, содержащие указанные значения в указанных полях.
- MODIFY изменяет все записи в определенном файле, задав указанные значения определенным полям в тех записях, которые содержат указанные значения в других полях.
- RETRIEVE отыскивает все записи, содержащие указанные значения в указанных полях.

Простой способ представления указателей на записи заключается в следующем. У каждого блока есть определенный *физический адрес,* т.е. место начала этого блока на устройстве внешней памяти. Отслеживание физических адресов является задачей файловой системы. Одним из способов представления адресов записей является использование физического адреса блока, содержащего интересующую нас запись, со *смещением,* указывающим количество байт в блоке, предшествующих началу этой записи. Такие пары "физический адрес-смещение" можно хранить в полях типа "указатель на запись".

Древовидные структуры данных можно использовать для представления внешних файлов. B-дерево — это особый вид сбалансированного m -арного дерева, который позволяет нам выполнять операции поиска, вставки и удаления записей из внешнего файла с гарантированной производительностью для самой неблагоприятной ситуации. С формальной точки зрения *B-дерево порядка m* представляет собой m-арное дерево поиска, характеризующееся следующими свойствами.

1. Корень либо является листом, либо имеет по крайней мере двух сыновей.
2. Каждый узел, за исключением корня и листьев, имеет от $[m/2]$ до m сыновей.
3. Все пути от корня до любого листа имеют одинаковую длину.

B-дерево можно рассматривать как иерархический индекс, каждый узел в котором занимает блок во внешней памяти. Корень B-дерева является индексом первого уровня. Каждый нелистовой узел на B-дереве имеет *форму* $(p_0,k_1,p_1,k_2,p_2,\ldots,k_n,p_n)$, где p_i является указателем на i-го сына, $0 \le i \le n$, а k_i — ключ, $1 \le i \le n$. Ключи в узле упорядочены, поэтому $k_1 < k_2 < \ldots < k_n$. Все ключи в поддереве, на которое указывает p_0, меньше, чем k_1 . В случае $1 \le i < n$ все ключи в поддереве, на которое указывает p_i имеют значения, не меньшие,

чем k_i, и меньшие, чем k_{i+1}. Все ключи в поддереве, на которое указывает p_n, имеют значения, не меньшие, чем k_n.

8.4. Управление памятью.

В общем случае при распределении памяти должны быть решены следующие вопросы:

- способ учета свободной памяти;
- дисциплины выделения памяти по запросу;
- обеспечение утилизации освобожденной памяти.

Память всегда выделяется блоками - т.е. обязательно непрерывными последовательностями смежных ячеек. Блоки могут быть фиксированной или переменной длины. Фиксированный размер блока гораздо удобнее для управления: в этом случае вся доступная для распределения память разбивается на "кадры", размер каждого из которых равен размеру блока, и любой свободный кадр годится для удовлетворения любого запроса. К сожалению, лишь ограниченный круг реальных задач может быть сведен к блокам фиксированной длины.

Одной из проблем, которые должны приниматься во внимание при управлении памятью является проблема фрагментации (дробления) памяти. Она заключается в возникновении "дыр" - участков памяти, которые не могут быть использованы. Различаются дыры внутренние и внешние. Управление памятью должно быть построено таким образом, чтобы минимизировать суммарный объем дыр.

Система управления памятью должна прежде всего "знать", какие ячейки имеющейся в ее распоряжении памяти свободны, а какие - заняты. Методы учета свободной памяти основываются либо на принципе битовой карты, либо на принципе списков свободных блоков.

В методах битовой карты создается "карта" памяти - массив бит, в котором каждый однобитовый элемент соответствует единице доступной памяти и отражает ее состояние: 0 - свободна, 1 - занята. Если считать единицей распределения единицу адресации - байт, то сама карта памяти будет занимать 1/8 часть всей памяти, что делает ее слишком дорогостоящей. Поэтому при применении методов битовой карты обычно единицу распределения делают более крупной, например, 16 байт. Карта, таким образом, отражает состояние каждого 16-байтного кадра. Карта может рассматриваться как строка бит, тогда поиск участка памяти для выделения выполняется как поиск в этой строке подстроки нулей требуемой длины.

В другой группе методов участки свободной памяти объединяются в связные списки. В системе имеется переменная, в которой хранится адрес первого свободного участка. В начале первого свободного участка записывается его размер и адрес следующего свободного участка. В простейшем случае список свободных блоков никак не упорядочивается. Поиск выполняется перебором списка.

8.5. Управление блоками одинакового размера

Весьма привлекательным подходом к выявлению недоступных ячеек является включение в каждую ячейку так называемого *контрольного счетчика,* или *счетчика ссылок* (reference count), т.е. целочисленного поля, значение которого равняется количеству указателей на соответствующую ячейку. Работу такого счетчика обеспечить несложно. С момента создания указателя на какую-либо ячейку значение контрольного счетчика для этой ячейки увеличивается на единицу. Когда переназначается ненулевой указатель, значение контрольного счетчика для указываемой ячейки уменьшается на единицу. Если значение контрольного счетчика становится равным нулю, соответствующая ячейка оказывается невостребованной и ее можно вернуть в список свободного пространства.

8.6. Выделение памяти для объектов разного размера

Рассмотрим теперь управление динамической памятью (кучей), когда существуют указатели на выделенные блоки. Эти блоки содержат данные того или иного типа. Если мы хотим, чтобы эти свободные блоки можно было найти, когда программе потребуется память для хранения новых данных, необходимо сделать следующие предположения, касающиеся блоков динамической памяти (эти предположения распространяются на весь данный раздел).

Каждый блок имеет объем, достаточный для хранения

а) *счетчика,* указывающего размер блока (в байтах или машинных словах в соответствии с конкретной компьютерной системой);

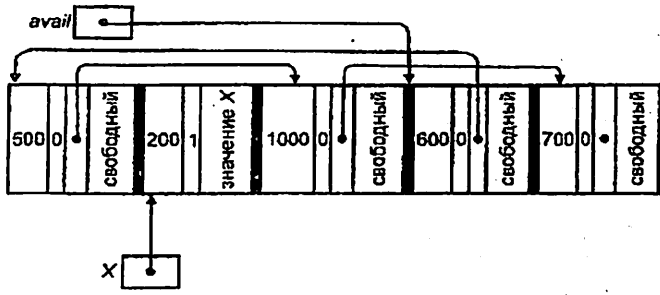
б) указателя (для связи данного блока со свободным пространством);

в) бита заполнения, указывающего, является ли данный блок пустым; этот бит называется битом *full/empty* (заполнено/свободно) или *used/unused* (используется/не используется).

В свободном (пустом) блоке слева (со стороны меньших значений адреса) содержатся счетчик, указывающий длину блока, бит заполнения, содержащий значение 0 (свидетельствует о том, что данный блок — пустой), указатель на следующий свободный блок.

Блок, в котором хранятся данные, содержит (слева) счетчик, бит заполнения, содержащий значение 1 (свидетельствует о том, что данный блок занят), и собственно данные.

На этом рисунке показан пример *фрагментации,* выражающейся в том, что крупные области памяти представляются в списке свободного пространства все более мелкими "фрагментами", т.е. множеством небольших блоков, составляющих целое (блок данных или свободное пространство).



Можно указать на три подхода к борьбе с фрагментацией.

- Можно воспользоваться одним из нескольких подходов (например, хранение списка блоков свободного пространства в отсортированном виде), которые приводят к затратам времени, пропорциональным длине списка свободного пространства, каждый раз, когда блок становится неиспользуемым, но позволяют находить и объединять пустых соседей.
- Можно воспользоваться списком блоков свободного пространства с двойной связью каждый раз, когда блок становится неиспользуемым; кроме того, можно применять указатели на соседа слева во всех блоках (используемых и неиспользуемых) для объединения за фиксированное время пустых соседей.

Для объединения пустых соседей ничего не делать в явном виде. Когда нельзя найти блок, достаточно большой, чтобы в нем можно было запомнить новый элемент данных, нужно просмотреть блоки слева направо, подсоединяя пустых соседей, а затем создавая новый список свободного пространства.

Дисциплины выделения памяти решают вопрос: какой из свободных участков должен быть выделен по запросу. Выбор дисциплины распределения не зависит от способа учета свободной памяти. Две основные дисциплины сводятся к принципам "самый подходящий" и "первый подходящий". По дисциплине "самый подходящий" выделяется тот свободный участок, размер которого равен запрошенному или превышает его на минимальную величину. По дисциплине "первый подходящий" выделяется первый же найденный свободный участок, размер которого не меньше запрошенного. При применении любой дисциплины, если размер выбранного для выделения участка превышает запрос, выделяется запрошенный объем памяти, а остаток образует свободный блок меньшего размера. В некоторых системах вводится ограничение на минимальный размер свободного блока: если размер остатка меньше некоторого граничного значения, то весь свободный блок выделяется по запросу без остатка. Практически во всех случаях дисциплина "первый подходящий" эффективнее дисциплины "самый подходящий".

8.7. Уплотнение памяти

Существуют два общих подхода к решению этой проблемы.

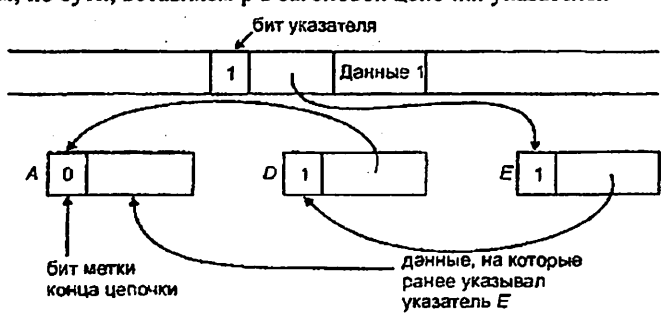
1. Пространство, выделяемое для хранения данных, состоит из нескольких пустых блоков. В таком случае можно потребовать, чтобы все блоки имели

одинаковый размер и чтобы в них было предусмотрено место для указателя и данных. Указатель в используемом блоке указывает на следующий блок, используемый для хранения данных (в последнем блоке указатель равен нулю).
2. Если объединение смежных пустых блоков не позволяет получить достаточно большой блок, данные в динамической памяти нужно переместить таким образом, чтобы все заполненные блоки сместились влево (т.е. в сторону позиций с меньшими номерами); в этом случае справа образуется один крупный блок свободной памяти.

Ф. Л. Моррис (F. L. Morris) разработал метод уплотнения динамической памяти, не предусматривающий резервирования пространства в блоках для хранения адресов передачи. Ему, однако, требуется дополнительный бит метки конца цепочки указателей. Суть этого метода заключается в создании цепочки указателей, исходящей из определенной позиции в каждом заполненном блоке и связывающей все указатели с этим блоком.

Для создания таких цепочек указателей используется следующий метод. Сначала просматриваются все указатели в любом удобном порядке. Допустим, просматриваем указатель *p* на блок *B*. Если бит метки конца в блоке *B* равен 0, значит, *p* является первым найденным нами указателем на блок *B*. Мы помещаем в *p* содержимое тех позиций B, которые используются для цепочки указателей, и делаем так, чтобы эти позиции *B* указывали на p. Затем устанавливаем бит метки конца в блоке Б в 1 (это означает, что теперь у него есть указатель), а бит метки конца в *p* устанавливаем в 0 (это означает конец цепочки указателей и присутствие смещенных данных).

Допустим теперь, что при просмотре указателя *p* на блок *B* бит метки конца в блоке *B* равен 1. Это значит, что блок *B* уже содержит заголовок цепочки указателей. Мы копируем в *p* указатель, содержащийся в B, и делаем так, чтобы B указывал на p, а бит метки конца в p устанавливаем в 1. Тем самым, по сути, вставляем p в заголовок цепочки указателей



Создание цепочки указателей

После того как мы свяжем все указатели на каждый блок в цепочку, исходящую из этого блока, можно переместить заполненные блоки как можно дальше влево (примерно так, как это сделано в рассмотренном выше алгоритме). Наконец, просматриваем каждый блок на его новой позиции и пробегаем всю его цепочку указателей. Теперь надо сделать так, чтобы каждый

встреченный указатель указывал на блок в его новой позиции. Когда обнаруживается конец цепочки, переносим данные из блока *B*, содержащиеся в последнем указателе, в крайнее правое положение в блоке *B* и устанавливаем в 0 бит метки конца в этом блоке.

Пример: **сортировка слиянием.**

```
function Find(var fp:filtype; i:integer):DataItem;
var t:DataItem;
begin Seek(fp, i-1);
Read(fp, t);
Find := t;
end;
procedure Mergesort(var fp: filetype; count:integer);
var i, j, k, l, t, h, m, p, q, r: integer;
ch1, ch2:DataItem
up: Boolean;
begin
up := TRUE;
p := 1;
repeat
h := 1; m := count;

if up then
begin
i := 1; j := count; k := count+1; l := 2*count;
end else begin
k := 1; l := count; i := count+1; j := 2*count;
end; repeat
if m>=p then q := p else q := m;
m := m-q;
if m>=p then r := p else r := m;
m := m-r;
while (q<>0) and (r<>0) do
begin
if Find(fp,i) < Find(fp,j) then
begin
Seek(fp, i-1); Read(fp,ch2);
Seek(fp, k-1); Write(fp,ch2);
k := k+h; i := i+1; q := q-1;
end else
begin
Seek(fp, j-1); Read(fp,ch2);
Seek(fp, k-1); Write(fp,ch2);
k := k+h; j := j-1; r := r-1;
end;
end;
```

```
while r<>0 do
begin
Seek(fp, j-1); Read(fp,ch2);
Seek(fp, k-1); Write(fp,ch2);
k := k+h; j := j-1; r := r-1;
end;
while q<>0 do
begin
Seek(fp, i-1); Read(fp,ch2);
Seek(fp, k-1); Write(fp,ch2);
k := k+h; i := i+1; q := q-1;
end;
h := -1; t := k;
k := l;
l := t;
until m = 0:
up := not up;
p := p*2;
until p >= count;
if not up then
for i := 1 to count do
begin
Seek(fp, i-1+count); Read(fp,ch2);
Seek(fp, i-1); Write(fp,ch2);
end;
end; { конец сортировки методом слияния }
```

Задание.

1. Реализовать алгоритм m-канальной сортировки слиянием с использованием 2m файлов.

2. Реализовать алгоритм многофазной сортировки (m-канальная сортировка с использованием m+1 файлов). В течение одного прохода, когда серии от каждого из *m* файлов объединяются в серии *(m* + 1)-го файла, нет нужды использовать все серии от каждого из *m* входных файлов. Когда какой-либо из файлов становится выходным, он заполняется сериями определенной длины, причем количество этих серий равно минимальному количеству серий, находящихся в сливаемых файлах. В результате каждого прохода получаются файлы разной длины. Поскольку каждый из файлов, загруженных сериями в результате предшествующих *m* проходов, вносит свой вклад в серии текущего прохода, длина всех серий на определенном проходе представляет собой сумму длин серий, созданных за предшествующие *m* проходов.

3. Реализовать сортировку слиянием с использованием 6-буферной схемы.

4. Реализовать сортировку слиянием с использованием 4-буферной схемы.

5. Разработать программный модуль или класс, реализующий тип данных «файл» с функциями INSERT, DELETE, MODIFY и RETRIEVE.

6. Хеширование — широко распространенный метод обеспечения быстрого доступа к информации, хранящейся во вторичной памяти. Записи файла мы распределяем между так называемыми *сегментами,* каждый из которых состоит из связного списка одного или нескольких блоков внешней памяти. Имеется таблица сегментов, содержащая B указателей, — по одному на каждый сегмент. Каждый указатель в таблице сегментов представляет собой физический адрес первого блока связного списка блоков для соответствующего сегмента. Сегменты пронумерованы от 0 до B-1. Хеш-функция h отображает каждое значение ключа в одно из целых чисел от 0 до B-1. Разработать структуру данных и алгоритм такого доступа к файлу.

7. Распространенным способом организации файла записей является поддержание файла в отсортированном (по значениям ключей) порядке. Чтобы облегчить процедуру поиска, можно создать второй файл, называемый *разреженным индексом,* который состоит из пар (x, b), где x — значение ключа, а b — физический адрес блока, в котором значение ключа первой записи равняется x. Этот разреженный индекс отсортирован по значениям ключей. Реализовать структуру данных и алгоритм такого доступа к файлу.

8. Еще одним способом организации файла записей является сохранение произвольного порядка записей в файле и создание другого файла, с помощью которого будут отыскиваться требуемые записи; этот файл называется *плотным индексом.* Плотный индекс состоит из пар (x, p), где p — указатель на запись с ключом x в основном файле. Эти пары отсортированы по значениям ключа. Реализовать структуру данных и алгоритм такого доступа к файлу.

9. Напишите программы, реализующие операторы RETRIEVE, INSERT, DELETE и MODIFY для файлов в виде B-дерева.

10. Реализовать алгоритм Морриса для уплотнения динамической памяти.

Использованная литература

Основная литература.

1. Алфред В. Ахо., Джон Э. Хоп Крофт, Джефри Д. Ульман. Структура данных и алгоритмов. Издательский дом «Вильямс» Москва – Санкт-Петербург – Киев, 2003 – 384 с.
2. Wirth N. Algorithms + Dato Structures = Program, Prentice – Hall, Fuglowood Cliffs, N. ., 1976 (Русский перевод: Вирт Н. Алгоритмы + структуры данных = программы. – N., «Мир», 1985).
3. Aho, A. V., J. E. Hopcroft, and J. D. Ullman (1974). The design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass. (Русский перевод: Ахо А., Хоркрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М., «Мир», 1979.)
4. Berge, C. (1958). The Theory of Graphs and its Applications, Wiley, N. Y. (Русский перевод: Берж С. Теория графов и ее применение. – М., ИЛ, 1962.)
5. Garey, M. R., and D. S. Johnson (1979). Computers and Intractability: a Guide to the Theory of NP-Completeness, Freeman, San Francisco. (Русский перевод: Гэри М., Джонсон Д.С. Вычислительные машины и трудноразрешимые задачи. – М., «Мир», 1982.)
6. Greene, D. H., and D. E. Knuth (1983). Mathematics for the Analysis of Algorithms, Birkhauser, Boston, Mass. (Русский перевод: Грин Д., Кнут Д., Математические методы анализа алгоритмов. – М., «Мир», 1987.)
7. Harary, F. (1969). Graph Theory, Addison – Wesley, Reading, Mass. (Русский перевод: Харари Ф., Теория графов. – М., «Мир», 1973.)
8. Knuth, D. E. (1968). The Art of Computer Programming Vol. I: Fundamental Algorithms, Addison – Wesley, Reading, Mass. (Русский перевод: Кнут Д. Искусство программирования для ЭВМ. Том 1: Основные алгоритмы. – М., «Мир», 1976. Русский перевод переработанного издания: Кнут Д. Искусство программирования. Том 1: Основные алгоритмы. – М., Издательский дом «Вильямс», 2000.)

Дополнительная литература:

9. Pratt, T. W. (1975). Programming Languages: Design and Implementation, Prentice-Hall, Englewood Cliffs, N. J. (Русский перевод: Пратт Т. Языки программирования. Разработка и реализация. – М., «Мир», 1979.)
10. Новиков Ф. А. Дискретная математика для программистов. СПб: Питер, 2004.-302с.
11. Джон Бентли Жемчужины программирования. СПб.: Питер, 2002.-272
12. Непейвода Н.Н., Скоплин И.Н. Основания программирования. – Москва Ижевск: Институт компьютерных исследований, 2003 г. 864 с.
13. Непейвода Н.Н. Стили и методы программирования. Лекции 2004 г. – М.Ижевск: Институт компьютерных исследований.-2004 г. -328 с.

СОДЕРЖАНИЕ

Методическое пособие для лабораторных занятий по дисциплине «Структуры данных и алгоритмов»

Разработка рассмотрена на заседании кафедры «ТП» и рекомендована к печати (протокол № от 2008 года)

Авторы: Зав. каф. Назиров Ш.А.
Асс.каф. И.Х.Бабакулов.
Асс.каф. Н.А.Арипова.
Маг. каф. Л.Х.Миндулина

Ответственный редактор: Проректор по учебной работе ТУИТ, д.т.н., проф. Каримов М.М.

Корректор: Павлова С.И.

Бумага офсетная. Заказ № 452
Тираж. 100
Отпечатано в типографии ТУИТ
Ташкент 700084, ул.А.Тимура – 108