УЗБЕКСКОЕ АГЕНТСТВО СВЯЗИ И ИНФОРМАТИЗАЦИИ

ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра «Технология программирования»

МЕТОДИЧЕСКОЕ ПОСОБИЕ

ДЛЯ ПРАКТИЧЕСКИХ 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 предназначено для проведения практических занятий по предмету «Структура данных и алгоритмов». Изложены способы создания и анализ структуры данных, абстрактные типы данных и анализ программ на псевдоязыке, представление списка с помощью различных структур (массив, указатели и курсор) и их анализ, представление степь, очередей, отображений, рекурсивных процедур, деревья и их представление, представление множеств графов (ориентированных и неориентированных), алгоритмы сортировки (внутренний и внешний), методов анализа алгоритмов и их сравнительный анализ, а также алгоритмы для внешней памяти и их управление.

Методические пособие рассчитано на студентов, аспирантов и

дическим советом

аров

уралиев

Введение

Структуры данных и алгоритмы являются фундаментом современного компьютерного программирования. Поэтому следует осветить структуры данных и алгоритмы в более широком контексте решения задач с использованием вычислительной техники, а также использовали абстрактные типы данных для неформального описания и реализации алгоритмов. И хотя сегодня абстрактные типы данных только начинают применять в современных языках программирования, тем не менее они являются полезным инструментом при разработке программ независимо от применяемого языка программирования.

Известно, что в последнее время широко внедряется идея вычисления и оценки времени выполнения алгоритмов (временную сложность алгоритмов), которую можно рассматривать как составную часть процесса компьютерного решения задач. В этом отражается наша надежда на то, что программисты осознают, что при решении задач прогрессирующе больших размеров особое значение имеет временная сложность выбранного алгоритма, а не возможности новых поколений вычислительных средств.

В методичке показаны разнообразные реализации абстрактных типов данных, начиная от стандартных списков, стеков, очередей и заканчивая множествами и отображениями, которые используются для неформального описания и реализации алгоритмов, а также методам анализа и построения алгоритмов. Приведено и исследовано множество различных алгоритмов для работы с графами, внутренней и внешней сортировки, управления памятью.

Методическое пособие состоит из восьми практических занятий.

Первое практическое занятие посвящено способам создания и анализ структуры данных, абстрактные типы данных, а также анализу программ на псевдоязыке.

Второе практическое занятие посвящено способам представления списка с помощью различных структур (массив, указатели, на основе курсоров) и их анализу.

Третье практическое занятие посвящено представлению стеков, очередей, отображений, рекурсивных процедур, представлению деревьев с помощью массивов, списков сыновей и братьев, а также двоичным деревьям и их представлению.

Четвертое практическое занятие посвящено описанию структуры основных операторов множеств и анализа специальных методов представления множеств.

Пятое практическое занятие посвящено представлению ориентированных и неориентированных графов

Шестое практическое занятие посвящено анализу алгоритмов внутренней и внешней сортировки.

Седьмое практическое занятие посвящено изучению методов анализа алгоритмов. Сравнительному анализу методов разработки алгоритмов.

Восьмое практическое занятие посвящено описанию структуры данных и алгоритмов внешней памяти, а также алгоритмам управления памятью.

В каждом практическом занятии приведены примеры и задания 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.

Практическое занятие - 1.

Тема: Способы создания и анализ структуры данных, абстрактные типы данных. Анализ программ на псевдоязыке.

Цель работы: изучение процесса разработки алгоритмов и структур данных и способов оценки их временной сложности.

В результате выполнения практической работы студенты должны:

- *знать* концепции типы и структуры данных , методы абстрагирования,основные этапы разработки программы; факторы, влияющие на время выполнения программы; О-символику и правила анализа временной сложности программ;
- *уметь* создавать типы и структуры данных , методы абстрагирования, основные этапы разрабатывать алгоритмы решения задач методом пошаговой детализации и оценивать их временную эффективность.

1. 1. Типы данных, структуры данных и абстрактные типы данных

Хотя термины *тип данных* (или просто *тип), структура данных* и *абстрактный тип данных* звучат похоже, но имеют они различный смысл. В языках программирования *тип данных* переменной обозначает множество значений, которые может принимать эта переменная. Например, переменная булевого (логического) типа может принимать только два значения: значение true (истина) и значение false (ложь) и никакие другие. Набор базовых типов данных отличается в различных языках: в языке Pascal это типы целых (integer) и действительных (real) чисел, булев (boolean) тип и символьный (char) тип. Правила конструирования составных типов данных (на основе базовых типов) также различаются в разных языках программирования: как мы уже упоминали, Pascal легко и быстро строит такие типы.

Абстрактный тип данных — это математическая модель плюс различные операторы, определенные в рамках этой модели. Как уже указывалось, мы можем разрабатывать алгоритм в терминах АТД, но для реализации алгоритма в конкретном языке программирования необходимо найти способ представления АТД в терминах типов данных и операторов, поддерживаемых данным языком программирования. Для представления АТД используются *структуры данных,* которые представляют собой набор переменных, возможно, различных типов данных, объединенных определенным образом.

Базовым строительным блоком структуры данных является *ячейка,* которая предназначена для хранения значения определенного базового или составного типа данных. Структуры данных создаются путем задания имен совокупностям (агрегатам) ячеек и (необязательно) интерпретации значения некоторых ячеек как представителей (т.е. указателей) других ячеек.

В качестве простейшего механизма агрегирования ячеек в Pascal и большинстве других языков программирования можно применять (одномерный) массив, т.е. последовательность ячеек определенного типа. Массив также можно рассматривать как отображение множества индексов (таких как целые числа 1, 2, ..., *n)* в множество ячеек. Ссылка на ячейку обычно состоит из имени массива и значения из множества индексов данного массива. В Pascal множество индексов может быть нечислового типа,

например (север, восток, юг, запад), или интервального типа (как 1..10). Значения всех ячеек массива должны иметь одинаковый тип данных. Объявление

имя: **array**[ТипИндекса] of ТипЯчеек;

задает *имя* для последовательности ячеек, тип для элементов множества индексов и тип содержимого ячеек.

Кстати, Pascal необычайно богат на типы индексов. Многие языки программирования позволяют использовать в качестве индексов только множества последовательных целых чисел. Например, чтобы в языке Fortran в качестве индексов массива можно было использовать буквы, надо все равно использовать целые индексы, заменяя "A" на 1, "B" на 2, и т.д.

Другим общим механизмом агрегирования ячеек в языках программирования является *структура записи. Запись* (record) можно рассматривать как ячейку, состоящую из нескольких других ячеек (называемых *полями),* значения в которых могут быть разных типов. Записи часто группируются в массивы; тип данных определяется совокупностью типов полей записи. Например, в Pascal объявление

```
var
reclist:  array[1..4]  of record
data: real;
next: integer end
```

задает имя *reclist* (список записей) 4-элементного массива, значениями которого являются записи с двумя полями: *data* (данные) и *next* (следующий).

Третий метод агрегирования ячеек, который можно найти в Pascal и некоторых других языках программирования, — это *файл.* Файл, как и одномерный массив, является последовательностью значений определенного типа. Однако файл не имеет индексов: его элементы доступны только в том порядке, в каком они были записаны в файл. В отличие от файла, массивы и записи являются структурами с "произвольным доступом", подразумевая под этим, что время доступа к компонентам массива или записи не зависит от значения индекса массива или указателя поля записи. Достоинство агрегирования с помощью файла (частично компенсирующее описанный недостаток) заключается в том, что файл не имеет ограничения на количество составляющих его элементов и это количество может изменяться во время выполнения программы.

1.2. Абстрактные типы данных

Большинство понятий, введенных в предыдущем разделе, обычно излагаются в начальном курсе программирования и должны быть знакомы читателю. Новыми могут быть только абстрактные типы данных, поэтому сначала обсудим их роль в процессе разработки программ. Прежде всего сравним абстрактный тип данных с таким знакомым понятием, как процедура.

Процедуру, неотъемлемый инструмент программирования, можно рассматривать как обобщенное понятие оператора. В отличие от ограниченных по своим возможностям встроенных операторов языка программирования (сложения, умножения и т.п.), с помощью процедур программист может создавать собственные операторы и применять их к операндам различных типов, не только базовым. Примером такой процедуры-оператора может служить стандартная подпрограмма перемножения матриц.

Другим преимуществом процедур (кроме способности создавать новые операторы) является возможность использования их для *инкапсулирования* частей алгоритма путем помещения в отдельный раздел программы всех операторов, отвечающих за определенный аспект функционирования программы. Пример инкапсуляции: использование одной процедуры для чтения входных данных любого типа и проверки их корректности. Преимущество инкапсуляции заключается в том, что мы знаем, какие инкапсулированные операторы необходимо изменить в случае возникновения проблем в функционировании программы. Например, если необходимо организовать проверку входных данных на положительность значений, следует изменить только несколько строк кода, и мы точно знаем, где эти строки находятся.

1.2.1.Определение абстрактного типа данных

Мы определяем *абстрактный тип данных* (АТД) как математическую модель с совокупностью операторов, определенных в рамках этой модели. Простым примером АТД могут служить множества целых чисел с операторами объединения, пересечения и разности множеств. В модели АТД операторы могут иметь операндами не только данные, определенные АТД, но и данные других типов: стандартных типов языка программирования или определенных в других АТД. Результат действия оператора также может иметь тип, отличный от определенных в данной модели АТД. Но мы предполагаем, что по крайней мене один операнд или результат любого оператора имеет тип данных, определенный в рассматриваемой модели АТД.

Две характерные особенности процедур — обобщение и инкапсуляция, — о которых говорилось выше, отлично характеризуют абстрактные типы данных. АТД можно рассматривать как обобщение простых типов данных (целых и действительных чисел и т.д.), точно так же, как процедура является обобщением простых операторов (+, - и т.д.). АТД инкапсулирует типы данных в том смысле, что определение типа и все операторы, выполняемые над данными этого типа, помещаются в один раздел программы. Если необходимо изменить реализацию АТД, мы знаем, где найти и что изменить в одном небольшом разделе программы, и можем быть уверенными, что это не приведет

к ошибкам где-либо в программе при работе с этим типом данных. Более того, вне раздела с определением операторов АТД мы можем рассматривать типы АТД как первичные типы, так как объявление типов формально не связано с их реализацией. Но в этом случае могут возникнуть сложности, так как некоторые операторы могут инициироваться для более одного АТД и ссылки на эти операторы должны быть в разделах нескольких АТД.

Для иллюстрации основных идей, приводящих к созданию АТД, рассмотрим процедуру *greedy* из предыдущего раздела (листинг 1.3), которая использует простые операторы над данными абстрактного типа LIST (список целых чисел). Эти операторы должны выполнить над переменной *newclr* типа LIST следующие действия.

- Сделать список пустым.
- Выбрать первый элемент списка и, если список пустой, возвратить значение *null.*
- Выбрать следующий элемент списка и возвратить значение *null,* если следующего элемента нет.
- Вставить целое число в список.

Возможно применение различных структур данных, с помощью которых можно эффективно выполнить описанные действия. (Подробно структуры данных будут рассмотрены в главе 2.) Если в листинге 1.3 заменить соответствующие операторы выражениями

```
MAKENULL(newclr);
w := FIRST(newclr);
w := NEXT(newclr);
INSERT(v, newclr);
```

то будет понятен один из основных аспектов (и преимуществ) абстрактных типов данных. Можно реализовать тип данных любым способом, а программы, использующие объекты этого типа, не зависят от способа реализации типа — за это отвечают процедуры, реализующие операторы для этого типа данных.

Вернемся к абстрактному типу данных GRAPH (Граф). Для объектов этого типа необходимы операторы, которые выполняют следующие действия.

- Выбирают первую незакрашенную вершину.
- Проверяют, существует ли ребро между двумя вершинами.
- Помечают вершину цветом.
- Выбирают следующую незакрашенную вершину.

Очевидно, что вне поля зрения процедуры *greedy* остаются и другие операторы, такие как вставка вершин и ребер в граф или помечающие все вершины графа как незакрашенные. Различные структуры данных, поддерживающие этот тип данных, будут рассмотрены в главах 6 и 7.

Необходимо особо подчеркнуть, что количество операторов, применяемых к объектам данной математической модели, не ограничено. Каждый набор операторов определяет отдельный АТД. Вот примеры операторов, которые можно определить для абстрактного типа данных SET (Множество).

- MAKENULL(A). Эта процедура делает множество А пустым множеством.

– UNION(A, B, C). Эта процедура имеет два "входных" аргумента, множества A и B, и присваивает объединение этих множеств "выходному" аргументу — множеству C.
– SIZE(A). Эта функция имеет аргумент-множество *A* и возвращает объект целого типа, равный количеству элементов множества *A*.Термин *реализация* АТД подразумевает следующее: перевод в операторы языка программирования объявлений, определяющие переменные этого абстрактного типа данных, плюс процедуры для каждого оператора, выполняемого над объектами АТД. Реализация зависит от *структуры данных*, представляющих АТД. Каждая структура данных строится на основе базовых типов данных применяемого языка программирования, используя доступные в этом языке средства структурирования данных. Структуры массивов и записей — два важных средства структурирования данных, возможных в языке Pascal. Например, одной из возможных реализаций переменной *S* типа SET может служить массив, содержащий элементы множества *S*.

Одной из основных причин определения двух различных АТД в рамках одной модели является то, что над объектами этих АТД необходимо выполнять различные действия, т.е. определять операторы разных типов. В этой книге рассматривается только несколько основных математических моделей, таких как теория множеств и теория графов, но при различных реализациях на основе этих моделей определенных АТД будут строиться различные наборы операторов.

В идеале желательно писать программы на языке, базовых типов данных и операторов которого достаточно для реализации АТД. С этой точки зрения язык Pascal не очень подходящий язык для реализации различных АТД, но, с другой стороны, трудно найти иной язык программирования, в котором можно было бы так непосредственно декларировать АТД. Дополнительную информацию о таких языках программирования см. в библиографических примечаниях в конце главы.

1.3. Основные структуры данных

К данным, которые обрабатывают ЭВМ, чаще всего относятся числовые величины, строки символов, значения измерений, математические формулы, графические изображения. Чтобы осуществить обработку с помощью ЭВМ, данные необходимо поместить в память ЭВМ с учетом способов внутреннего представления. При этом идеальной была бы структура представления, адекватная структуре объекта. Описание структуры объектов осуществляется с помощью универсальных языков программирования, используемых для решения широкого круга задач, либо с помощью языков спецификации.

Однако обычные универсальные языки программирования из-за многообразия структур обрабатываемых объектов не допускают возможности подобного представления. Из опыта известно, что путем абстрагирования обрабатываемого объекта можно представить необходимые для обработки данные, используя лишь несколько различных универсальных структур

данных. Рассматриваемые в данной главе основные структуры данных являются универсальными.

Не существует однозначного определения того, какая структура данных является основной, однако из опыта использования и разработки языков программирования можно считать, что существуют такие структуры данных, которые большинство специалистов считают основными. Такие структуры с математической точки зрения являются *базисными,* и в этом смысле они универсальны.

1.3.1. Структуры данных и типы данных

Прежде чем рассматривать базисные структуры данных, введем понятие *типа данных.* Используемые в языках программирования структуры данных можно классифицировать в зависимости от допустимых операций и целей использования. Такая классификация упрощает не только понимание программы и описываемого ею процесса обработки данных, но и позволяет обнаружить ошибочное использование данных до выполнения программы (обычно во время компиляции). Отдельные классифицированные категории данных называют *типами данных.*

Понятие типа данных играет центральную роль в языках программирования высокого уровня. Для всех переменных в таких языках тип данных задается путем описания, причем для каждой операции однозначно определяется тип ее операндов. Это позволяет определить тип данных в любом выражении. Однозначное определение типа данных обеспечивает:

1. Возможность определения объема памяти для переменных до выполнения обработки и осуществления эффективного управления памятью. Например, описание

var *A:* **array**[1 .. 10] **of** *integer*

определяет переменную *A* как массив, состоящий из 10 элементов данных типа *integer.* Анализ текста программы позволяет определить размер области памяти для хранения значений этой переменной.

2. Возможность обнаружения еще до выполнения программы ошибочных операций и выражений. Это позволяет уменьшить трудоемкость отладки программы и проверить структуру программы. Например, если переменные *x, xx, z* описать как

var *x: real; xx, z: integer*

то можно выявить следующее ошибочное выражение:

$$z := x \ \textbf{mod}\ 10$$

которое для переменной *x* типа *real* ошибочно задает операцию той, допустимую только для данных типа *integer.*

3. Возможность эффективного комментирования и анализа семантики программ.

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

$$D_t = \{d | d \in D \; u \; type(d) = t\}$$

есть множество данных типа *t*.

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

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

Например, для ранее приведенной операции **mod** справедлива следующая функция:

$$\textbf{mod}: \quad D_{integer} \times D_{integer} \rightarrow D_{integer}.$$

Элементами множества ***F*** являются:

1. Элементарные операции манипулирования данными, например вышеуказанная **mod.**

2. Операции отношения, например

$$equal: D_t \times D_t \rightarrow D_{boolean},$$

которые используются для суждения о равенстве двух данных (в большинстве случаев *equal (x,y)* записывается как $x = y$).

3. Операции преобразования типа, используемые для преобразования типа данных; например, в Фортране это *IFIX* и *FLOAT*:

$$IFIX: \quad D_{real} \rightarrow D_{integer},$$

$$FLOAT: \quad D_{integer} \rightarrow D_{real}.$$

4. Операции, обеспечивающие структурирование данных и выборку данных при их реструктурировании.

Таким образом, тип данных в языке программирования определяется совокупностью $D = \{D_t | t \in T\}$ и ***F***, которую формально можно записать как ***(D,F)***.

Указанная концепция типа данных возникла сравнительно недавно и, как будет рассмотрено ниже, основывается на понятии абстрактного типа данных.

Для того чтобы однозначно определить тип данных, необходимо однозначно задать множество *D* и *F*. Обычно в традиционных языках программирования это осуществляется неявно, основываясь на таких общеизвестных понятиях, как последовательность и множество, учитывая организацию памяти и операционной системы ЭВМ, используемой для выполнения программ. Однако для того, чтобы однозначно определить семантику программы, необходимо дать однозначное определение множеств *D* и *F*.

1.3.2. Основные типы данных

Традиционно используемые в универсальных языках программирования типы данных должны удовлетворять следующим двум основным требованиям:

1. Описание объектов реального мира с использованием средств языка программирования должно быть по возможности простым для понимания и при этом небольшим по объему,

2. Реализация языка программирования должна обеспечивать высокую эффективность работы существующих ЭВМ.

Если придавать большое значение первому требованию, то для описания сложных предметных областей потребуется большое количество сложных типов данных, что усложняет создание высокоэффективной системы программирования. Если же уделить большее внимание второму требованию, то возникают проблемы, связанные с использованием средств языка программирования для описания предметной области. Поэтому при выборе соответствующих типов данных для универсальных языков программирования трудно найти приемлемое решение. Исходя из проведенных до настоящего времени исследований в области языков программирования и опыта их использования, большинство специалистов считают уместным рассматривать некоторые типы данных в качестве основных.

Основные типы данных классифицируются следующим образом:

- Простые типы данных.
- Структурированные типы данных.
- Ссылочный тип данных.

1.3.2.1. Простые типы данных

Простые типы данных не обладают внутренней структурой и представляют собой конечный набор типов, называемых также *примитивными, элементарными* или *базисными*. К простым типам данных относятся следующие типы:

1. Логический.
2. Литерный.
3. Целый.
4. Вещественный.
5. Перечисляемый.
6. Интервальный.

1.3.2.2. Структурированные типы данных

Эти типы данных предназначены для задания сложных структур данных. Структурированные типы данных конструируются из составляющих элементов, называемых *компонентами*, которые в свою очередь могут обладать структурой. Основой их конструирования являются определенные *правила (операции) конструирования (конструкторы типов)*, причем каждому из них соответствует определенный структурированный тип данных. Приведенные ниже основные структурированные типы данных интересны не только с точки зрения обработки данных, но также являются важными математическими понятиями:

1. Функция с конечной областью определения, или массив.
2. Декартово (прямое) произведение.
3. Объединение.
4. Множество.
5. Последовательность.
6. Рекурсивные структуры.

Во многих языках программирования функцию с конечной областью определения относят к данным типа массив. В данной книге этот тип данных далее будет называться *массивом*. Тип прямого произведения часто называют типом *записи* или *структурой*.

При структурировании данных обычно используются следующие основные операторы:

1. Оператор конструирования.
2. Оператор выбора.

Оператор конструирования позволяет из компонентов образовывать структурированный тип, и, наоборот, *оператор выбора* из структурированного типа данных выделяет образующие его компоненты.

Пусть C — оператор конструирования, позволяющий получать из компонент с типами $t_1, \ldots, t_n$ структурированный тип данных t, а S_i — оператор выбора, выделяющий i-ю компоненту структурированного типа данных. C и S_i являются следующими функциями:

$$C\colon D_{t_1} \times D_{t_2} \times \ldots \times D_{t_n} \to D_t.$$

$$S_i\colon D_t \to D_{t_i} \quad (i = 1, \ldots, n).$$

В данном случае для C и S_i справедливы следующие выражения:

1. Для любых $d_1 \in D_{t_1}, \ldots, d_n \in D_{t_n}$

$$S_i\left(C\left(d_1, \ldots, d_n\right)\right) = d_i.$$

2. Для любого $d \in D_t$

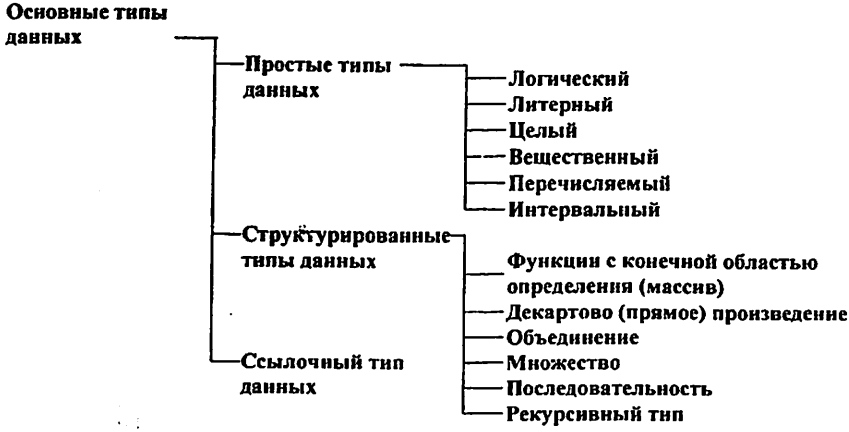
$$C\left(S_1\left(d\right), \ldots, S_n\left(d\right)\right) = d.$$

Конкретный вид операторов C и S_i зависит от используемого структурированного типа данных.

1.3.2.3. Ссылочный тип данных

Этот тип данных предназначен для обеспечения возможности указания на другие данные и называется *указателем (ссылкой)* . Ссылочный тип данных является средством организации и обработки сложных изменяющихся структур данных.

Сочетание в обычных языках программирования структурированных и ссылочного типов данных позволяет формировать сложные структуры данных. Рассмотренные выше основные типы данных можно проиллюстрировать следующей диаграммой:



1.4. Модели языков программирования

Понятие типа данных является одним из основополагающих и представляет собой самостоятельный объект изучения. Тип данных входит в языки программирования как структурный элемент и является полезным средством для реализации контроля типов в рамках языка. В этом смысле рассмотрение типов данных уместнее всего выполнять через языки программирования, использующие эти типы данных. В этой книге приведено краткое рассмотрение языков программирования с позиций типов данных.

Существуют различные языки программирования, и их структура не является единообразной. Наиболее широко распространены в настоящее время языки процедурного типа. Типичными их представителями являются языки Фортран, Кобол, Паскаль, ПЛ/1. В последнее время успешно изучаются и разрабатываются языки гак называемого непроцедурного типа, например языки функционального и логического типа. К этой категории принадлежат, например, «чистый» Лисп и Пролог. Существует еще целый ряд языков, обладающих своими характерными особенностями, но если рассматривать их с точки зрения типовости, то между ними есть много общего. Поэтому ниже будут рассмотрены главным образом языки процедурного типа. Основная цель

состоит не в рассмотрении типов данных некоторого языка, а в разъяснении понятий, общих для большого числа языков программирования. В связи с этим следует обратить внимание на то, что приводимые ниже примеры фрагментов программ взяты только для пояснения излагаемого материала.

1.4.1. Описание переменных и типов данных

В языках процедурного типа для присваивания некоторой переменной вновь вычисленного значения служит *оператор присваивания*. Переменная представляет собой «ящик», в который можно помещать значения данных. Тип переменной явно описывается в тексте программы и определяет класс значений, который может принимать переменная. Здесь имеют место два случая: 1) когда каждое значение, присваиваемое переменной, принадлежит определенному типу и 2) случай без такого ограничения, когда по ходу выполнения программы переменной можно присваивать значения любого типа. Языки программирования, в которых выполняется указанное ограничение (первый случай), называются *типизированными*, а во втором случае говорят о *нетипизированных*, т. е. бестиповых, языках.

В типизированном языке переменным должны быть явно приписаны их типы. В данной книге, следуя языку Паскаль, для описания типа переменной используется следующая конструкция:

var *x*: *t*; ...

где *x* — имя переменной, *t* является либо типом, либо типовым выражением. Таким образом, *t* является или именем типа переменной, или типовым выражением, состоящим из имен и операторов конструирования структурированного типа данных. Здесь могут иметь место два случая: 1) именем является идентификатор, соответствующий определенному в данном языке программирования типу данных, например *integer*, *real*, и 2) именем типа может являться идентификатор, задаваемый пользователем и соответствующий определяемому им типу данных. Предположим, что определение типа данных имеет следующий вид:

type *T* = *t*; ...

где *t* является типовым выражением, задающим тип данных, а *T* — имя, присваиваемое типу данных.

1.4..2. Эквивалентность типов данных

Рассмотрим, например, следующие описания типов данных o1, a2:

type *a1* =**array**[1 .. 10] **of** *integer*

a2 = **array**[1 .. 10] **of** *integer*

Эти описания типов определяют массивы целых чисел, состоящих из 10 элементов, которым присвоены различные имена *a1* и *a2*. В данном случае следует рассмотреть две ситуации: 1) *a1* и *a2* имеют одинаковые структуры и, как следствие, являются эквивалентными типами данных; 2) *a1* и *a2* имеют различные имена и, как следствие, являются различными типами данных. В первом случае считается, что оба типа данных являются *структурно*

эквивалентными, несмотря на различие присвоенных им имен, а также имен компонент, образующих эти структуры.
Например, если

type *b = integer;*
a3 = **array**[1 .. 10] **of** *b*

то *a1*, *a2* и *a3* все являются структурно эквивалентными.

В противоположность указанной существует точка зрения, что имя, присваиваемое типу данных, является также частью типа данных, и при установлении эквивалентности типов данных должны учитываться их имена.

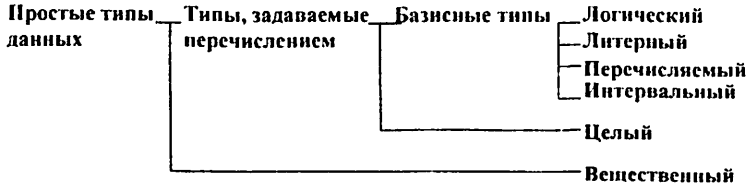
var *A: a1;*
B: a3;
A:=B

Данная точка зрения соответствует эквивалентности по имени и означает, что типы, которым в программе даны разные имена, считаются разными независимо от того, каковы их структуры, множества значений и т. п. Выбор той или иной точки зрения зависит от языка программирования; например, в сообщении о языке Паскаль нет точного определения эквивалентности типов данных, но в большинстве других языков принята позиция структурной эквивалентности. В стандарте языка Паскаль B56192; 1982 принята именная эквивалентность типа. С другой стороны, в языке Ада используются именные эквиваленты. С точки зрения, например, понимаемости программы, а также контроля типов данных лучшей является позиция именного эквивалента, но, с другой стороны, для именного эквивалента трудно обеспечить достаточно строгую запись. Какая из данных позиций лучше, безусловно определить трудно. В данной книге принята точка зрения именного эквивалента.

1.4.4. Простые типы данных

Простой тип данных не обладает внутренней структурой и служит для отображения простых фактов и задания основных базисных компонент структурированных типов данных.

В универсальных языках программирования традиционно существуют приведенные ниже простые типы данных. Кроме *перечисляемого* и *интервального* типов все остальные типы данных представляют собой *встроенные* типы.



Типы, задаваемые перечислением, характеризуются тем, что позволяют обозначать данные этих типов в виде списка символических имен. К этим типам относятся *логический, литерный, перечисляемый, интервальный* и *целый*

типы. В существующих языках программирования данные вещественного типа не являются вещественными с математической точки зрения и представляют собой тип, состоящий из конечных значений. Однако по замыслу он близок к вещественному, но объективный смысл в перечислении его элементов отсутствует. Поэтому обычно вещественный тип не является типом, задаваемым перечислением.

Если T --- один из типов, задаваемых перечислением, то обычно упорядоченность элементов в D_T задается порядком их перечисления. Упорядоченность последовательности $d_1, d_2, \ldots, d_N$ элементов множества D_T позволяет ввести функции $succ(d_i)$ и $pred(d_i)$ получения элементов d_{i+1} и d_{i-1}, являющихся по отношению к элементу d_i соответственно последующим и предыдущим элементами:

$$succ, pred : D_T \to D_T$$
$$succ(d_i) = d_{i+1}, 1 \le i \le N-1,$$
$$pred(d_i) = d_{i-1}, 2 \le i \le N.$$

Операция отношения

$$<: D_T \times D_T \to D_{boolean},$$

задающая отношение следования элементов D_T, определяется следующим образом:

$$d_i < d_j \Leftrightarrow i < j$$

Знак < в выражении $i<j$ обозначает отношение «меньше», заданное на множестве целых чисел. В выражении $d_i < d_j$ знак < имеет другой смысл, и для его обозначения было бы лучше употребить другой символ, но, следуя традиции, здесь использованы одинаковые символы.

Сказанное выше справедливо и для других операций отношений: $\le$, >, $\ge$, =, $\ne$.

Среди типов, задаваемых перечислением, тип, в котором действительно можно перечислять, в языке Паскаль называется скалярным типом. В данном случае для таких типов, как логический, литерный или устанавливаемый пользователем перечисляемый тип, число всех элементов, образующих соответствующий тип, невелико. Все рассматриваемые языки программирования позволяют осуществлять перечисление. В языках программирования диапазон представления типа *целый* является достаточно большим числом, и перечислить все принадлежащие ему целочисленные значения реально невозможно. Однако пространство целых чисел можно использовать в качестве индексов массива, например от 1 до 10, или для представления множества, состоящего из 10 элементов. Такие данные выражаются с помощью *интервального* типа данных.

1.4.4.1. Логический тип

Логический тип представляет тип, состоящий из логических значений **true** и **false**, и применяется для выражения, например, значений логических условий. Логический тип данных именуется как *boolean*, т.

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

В качестве операций над логическим типом используются

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

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

Они обозначают соответственно конъюнкцию, дизъюнкцию и отрицание:

p	*q*	*p* **and** *q*	*p* **or** *q*
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

p	**not** *p*
true	false
false	true

1.4.4.2. Литерный тип

Литерный тип предназначен для образования текстов, используемых для общения людей, состоит из литер и обозначается *character* Этот тип данных состоит из цифр, букв латинского алфавита и специальных знаков. Набор букв определяется языком. Существуют стандарты на литерные множества, например ASCII (American Standard Code for Information Interchange), JIS, ISO.

В языке Паскаль существуют две стандартные функции *ord* и *chr*, называемые *функциями преобразования*, которые позволяют отображать множество литер на подмножество натуральных чисел и наоборот:

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

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

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

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

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

Данные функции применяются, например, для преобразования литерных величин в отрезок целого типа и для выполнения обратного преобразования.

1.4.4.3. Целый и вещественный типы

Целый и *вещественный* типы данных предназначены для представления числовых значений. Отметим, что в математическом смысле целые и вещественные числа не эквивалентны. Целый и вещественный типы именуются *integer* и *real* соответственно.

Целый тип формируется из множества целочисленных значений, определяемого данным языком. Максимальное и минимальное представимые значения зависят от языка и используемой ЭВМ. Диапазон представимых

значений для ЭВМ с длиной слова в L бит определяется следующим соотношением:

$$-2^{L-1} \le n \le 2^{L-1} - 1.$$

Вещественный тип представляется в ЭВМ числами с плавающей точкой вида

где e — целое число, $a_i = 0$ или 1. Как правило, операции, определенные над данными вещественного типа, в строгом математическом смысле не эквивалентны операциям над вещественными числами из-за необходимости обращать внимание на ошибку округления результатов операций. Например, если x, y, z имеют вещественный тип, то соотношение

$$(x+y)+z=x+(y+z)$$

является некорректным.

По тем же причинам запись

if $x = y$ **then** z:=1 **else** z: =2 необходимо преобразовать в запись

if *abs* $(x - y) <$ *epsilon* **then** z:=1 **else** z: =2

предполагая, что число *epsilon* достаточно малое.

1.4.4.4. Перечисляемый тип

Перечисляемый тип обладает только свойством перечисления принадлежащих ему данных. Для данных такого типа кроме функций *pred, succ*, которые определяют в соответствии с заданным перечислением порядком предыдущие и последующие элементы, определены порядковые функции *ord* и *chr*, результат применения которых зависит от типа аргумента.

Перечисляемый тип относится к задаваемому пользователем типу данных. Задание типа осуществляется путем перечисления принадлежащих этому типу значений данных.

type $T = (c_1, c_2, \ldots, c_n)$

Здесь $c_1, c_2, \ldots, c_n$ являются значениями перечисляемого типа T. Например,

type *day* = *(Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)*

type *shape* = *(triangle, rectangle, circle, ellipse)*

type *color* = *(green, red, yellow)*

В этом примере тип данных *day* состоит из семи элементов *Sunday, . . . , Saturday*. Эти элементы не имеют какого-либо смысла, кроме того, что они просто задаются своими идентификаторами.

Таким образом, перечисляемый тип представляет собой наиболее простой тип данных и является удобным с точки зрения создания программ. Это обусловлено тем, что использованные в качестве значений перечисляемого типа имена легко запомнить, а их применение в качестве идентификаторов и меток делает программу более ясной. Например, с помощью перечисляемого типа данных *shape* можно задать множество, значения элементов которого соответствуют понятиям треугольника, четырехугольника, круга, эллипса, а для задания количества углов каждой такой фигуры можно использовать массив *angle*. Рассмотрим описание массива.

var *x: shape;*

angle: **array** *[shape]* **of** *integer*

В этом случае операторы

if *x = rectangle* **then** *angle* [*x*]:=4

и *angle [rectangle]:=4* более просты для понимания, чем

if *x = 2* **then** *angle[x]:=4*

и *angle[2]:=4;*

при задании соответствующих элементов *shape* в виде целых чисел 1, 2, 3, 4.

Перечисляемые типы и операторы цикла и выбора. При обработке данных для обеспечения возможности перечисления значений данных перечисляемого типа и представления программы в более простом виде можно использовать операторы цикла и выбора следующего вида:

1. *Оператор цикла*

for *x* : *T* **do** *statement(x)*

2. *Оператор выбора*

case *x* **of**

c_1 : $statement_1$;

c_2: $statement_2$;

……

c_n : $statement_n$

end

Оператор цикла означает, что для всех значений $x = c_1, c_2, \ldots, c_n$, принадлежащих *T*, выполняется *statement(x)*, а оператор выбора означает выполнение оператора $statement_i$, соответствующего значению c_i переменной *x* типа *T*.

1.4.4.5. Интервальный тип

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. В этом случае ошибки в программе легко обнаруживаются, а изменения вносятся без труда.

Интервальный тип задается указанием нижнего и верхнего значений в этом интервале. Пусть задано описание

например

type $T=t_{min}\ldots t_{max}$

например .

type *digit*= 0 . . 9

type *year*= 1900 .. 1999

type *Weekday*= *Monday . . Saturday*

В данном примере *digit* представляет тип данных, состоящий из 10 целых чисел от 0 до 9. *Year* определяет подмножество целых чисел, состоящее из 100 элементов в диапазоне от 1900 до 1999. В данном случае базисным типом является целый тип *integer*. Для *Weekday* в качестве базисного типа взят перечисляемый тип *day*, рассмотренный в разд. 2.4.4. *Weekday* задает тип

данных, где последовательно перечислены шесть элементов *Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.* В следующем примере рассмотрен интервальный тип, состоящий только из литер алфавита *(примечание:* литерные константы взяты в кавычки):

type *alphabet* = "A" .. "Z"

В общем случае, если определена операция op_0 для типа данных T_0, являющегося базисным для интервального типа T, то можно считать, что автоматически определена операция *op*, ограничивающая op_0 на T в интервале $[t_{min}, t_{max}]$. Например, для типа данных *digit* при операциях, связанных с типом целый, его область определения и область значений ограничены областью [0,9]. В операции сложения используется знак +'.

$$x +' y = \begin{cases} x + y, \text{ если } 0 \le x, y, x + y \le 9, \\ \text{неопределено в противном случае.} \end{cases}$$

Обычно для *op* и op_0 используются одинаковые знаки и знак +' записывается как +.

При подстановке значений переменным интервального типа требуется, чтобы значения принадлежали заданному интервалу.

Благодаря этому перед выполнением программы можно обнаружить присваивание недопустимых значений, например

var *x: year;*

x:=2050

Указанное значение переменной *x* объявленного выше типа *year* является недопустимым. В случае оператора присваивания вида

$x := x + 1$

перед выполнением программы невозможно определить корректность результата выполнения этого оператора присваивания. В данном случае потребуется дополнительная проверка во время выполнения программы. Такая проверка возможна и весьма важна при проверке значений индексов массивов, однако если об этом не позаботиться специально, то может произойти значительное снижение эффективности.

1.5. Структурированные типы данных

Структурированные типы данных предназначены для конструирования из конечного набора базисных типов сложных структур данных. Типичными примерами структур являются: функция с конечной областью определения, прямое (декартово) произведение, объединение, множество, последовательности и рекурсивно определенные на них понятия.

1.5.5.1. Функция с конечной областью определения

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*, поэтому в данной книге ниже будет использоваться этот термин. Конечное множество, являющееся областью определения данной функ-

ции (массива), называется *индексным множеством (типом)*, а его элементы называются *индексами*. Значение данной функции для некоторого значения индекса *i* называется *элементом массива, соответствующим i*.

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}:

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

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

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

Множество индексов *I* массива должно быть конечным, при этом должна обеспечиваться возможность перечисления его элементов. Поэтому требуется, чтобы тип индекса являлся простым типом с возможностью перечисления, например интервальным, перечисляемым, логическим или литерным типом. Массив *A* при $I = \{i_1, i_2, \ldots, i_n\}$ можно проиллюстрировать следующим образом:

$A[i_1]$	$A[i_2]$	...	$A[i_n]$
i_1	i_2		i_n

где $A[i]$ является значением *A* для $i \in I$.

Пример.

type *line* = **array** [1 .. 100] **of** *character*
type *employee* (Танака, Накамура, Като, Ямамото);
salary = **array***[employee]* **of** *integer*

Массив представляет собой некоторое количество расположенных в определенном порядке элементов одного типа. Индекс предназначен для обеспечения возможности указания на элементы массива. В рассмотренном выше примере последовательность из 100 литер определяет строку *(line)* и каждая из образующих строку литер имеет определенное положение в строке.

С другой стороны, массив *salary* соответствует таблице окладов. В данном случае типом индекса является перечисляемый тип *employee* и образованный из имен служащих. Массив *salary* соответствует вектору-строке, состоящему из целых чисел (окладов). Отличие от *line* состоит в том, что для указания элементов массива (окладов) в качестве индексов использованы идентификаторы:

150000	*200000*	*215000*	*180000*
Танака	*Накамура*	*Като*	*Ямамото*

Здесь необходимо обратить внимание на то, что, например, значение «Танака» не является именем, т. е. отличительной чертой данных, а представляет собой одно из значений элементов, образующих тип данных *employee*. Если последовательности литер необходимо дать имя, то лучше это сделать с помощью массива, задав соответствие между именем как отличительной чертой данных и идентификатором как последовательностью знаков:

type *nametable* = **array** *[employee]* **of** *name;*
name = **array** [1 .. 8] **of** *character*

Тип *nametable* является примером такого массива. Тип *name* представляет собой массив элементов литерного типа, состоящих из 8 · букв, предназначенных для задания имен как последовательности литер. Приведенное выше описание можно объединить в одно следующим образом:

type *nametable* = **array** *[employee]* **of array** [1 .. 8] **of** *character*

Операции над данными. Типичными операциями над данными типа массив являются:

1. Задание начальных значений элементов массива.
2. Выбор элементов массива по заданным значениям индексов.
3. Избирательное обновление массива.

Задание начальных значений элементов массива

Значения элементов массива Л можно идентифицировать указанием значений их индексов. Множество индексов $I = (i_1, i_2, ..., i_n)$ является перечисляемым, поэтому A можно задать перечислением значений A для $i \in I$. Если $a_k = A[I_k]$ (k=1, ..., n), то массив A типа T можно представить в виде

$T(a_1, a_2, ..., a_n)$.

Например, если A — массив типа *salary*, то

salary(150000, 200000, 215000, 180000)

В общем случае задание значений элементов массива сводится к заданию следующей функции:

$$T(\_,...,\_) : D_{T_0} n \to D_I .$$

Выбор элементов массива по заданным значениям индексов

Выбор элементов массива является операцией, обратной заданию значений элементов массива. Для заданного массива A операция выбора элемента массива заключается в определении значения элемента массива A, соответствующего значению *i-го* элемента множества индексов I, и обычно выражается с помощью следующей нотации:

$A[i]$,

где i представляет собой следующую функцию:

$$[i] : D_T \to D_{T_0} .$$

Например, значение элемента массива A типа *salary*, соответствующее значению индекса *Като*, задается следующим образом:

A [Като].

Очевидно, что оператор задания значений элементов массива T (_, ..., _) и оператор выбора элементов массива *[i]* связаны с операторами образования и выбора, рассмотренными в разд. 2.2. Существуют следующие соотношения:

$T(a_1, ..., a_n)[i] = a_i$,

$T(A[i_1], ..., A[i]) = A$.

Избирательное обновление массива

Основной способ преобразования одного массива в другой заключается в том, что значение некоторого элемента исходного массива заменяется на другое значение. Это называется *избирательным обновлением массива.* Массив A, в котором значение элемента, соответствующее значению индекса i, равно b, записывается следующим образом:

(A, i, b).

Для него можно записать следующее выражение:

$(A, i. b)[k]$ = **if** $k = i$ **then** b **else** $A[k]$

Обычно в языках процедурного типа переменные и присваиваемые им значения имеют одинаковый тип. В процессе выполнения программы значения, присваиваемые переменным, изменяются. Избирательное обновление массива можно реализовать с помощью операции присваивания значений элементам массива

$A[i]:=b$.

Однако при этом необходимо учитывать, что хранимое значение заменяется на (A,i, b) и операция присваивания является средством такого изменения. Даже в языках программирования функционального и логического типа, где отсутствует понятие присваивания значений переменным, избирательное обновление массивов является основной операцией для данных перечисляемого типа. В данном случае нет операций присваивания значений элементам массивов и имеет место только операция над всем массивом.

Очевидно, что избирательное обновление массива можно представить в виде следующей функции:

$$D_I \times D_I \times D_{I_0} \rightarrow D_I .$$

Многомерные массивы. До сих пор рассматривались одномерные массивы, содержащие только одно индексное множество. Аналогичным образом можно рассмотреть многомерные массивы. Многомерный массив представляет собой массив, у которого индексное множество является прямым (декартовым) произведением множеств I_1, I_2.... I_n и определяется следующим образом:

$$\textbf{type } T = \textbf{array } [I_1, I_2, ..., I_n] \textbf{ of } T_0$$

Если A — массив указанного типа T, то A является функцией следующего вида:

$$A : D_{I_1} \times D_{I_2} \times \ldots \times D_{I_n} \rightarrow D_{I_0} .$$

В языках программирования принята точка зрения, что такие многомерные массивы представляют собой структуру, составленную из

одномерных массивов, т. е. компонентами массива являются массивы. Если следовать данной концепции, то рассмотренный выше тип данных T можно описать следующим способом:

type T = **array** [I_1] **of array** [I_2] **of** ... **of array** [I_n] **of** T_0

Эти два способа описания, строго говоря, отличаются. В первом случае значения элементов массива $A[i_1, i_2, \ldots, i_n]$ определяются только для набора значений индексов $i_1 \in I, i_2 \in I, \ldots, i_n \in I$ и значение A $[i_1]$, соответствующее только значению индекса i_1, не имеет смысла. Во втором случае и $A[i_1]$, и A $[i_1, i_2, \ldots, i_n]$ (если это рассматривать как сокращенный способ описания $A[i_1]$ $[i_2] \ldots [i_n]$) имеют смысл.

Массивы и оператор цикла. Обычно в языках процедурного типа основными операторами манипулирования массивами являются операторы присваивания и цикла. Оператор присваивания уже был рассмотрен ранее. Чтобы выполнить некоторую операцию $P(e)$ над элементами e заданного массива A, необходимо перечислить его элементы и для них выполнить операцию P. Для этого применяют оператор цикла следующего вида:

for i:=I **do** $P(A[i])$

Если I находится в интервале 1 .. n, то можно записать следующее выражение:

for I := 1 **to** n **do** P $(A$ $[i])$

Например, если дан массив A с базисным типом данных *salary*, то увеличение оклада служащим на 10000 иен можно задать как

for x := *employee* **do** A $[x]$:=$A[x]$ + 10000

1.5.5.2. Прямое (декартово) произведение

Компоненты массива представляют собой структурированные данные одного типа. Массив объединяет данные с одинаковыми свойствами. В противоположность массивам компоненты прямого (декартова) произведения могут иметь различные типы. 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:

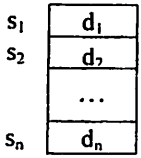
type T= **record** $s_1 : T_1;$
$s_2 : T_2;$
.....
$s_n : T_n;$

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 $D_{T_1}, D_{T_2}, \ldots D_{T_n}$:

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

Каждое значение D_T можно проиллюстрировать следующим образом:



где $d_i \in D_{T_i}$.

Идентификаторы s_1, s_2..., s_n используются для указания каждой компоненты d.

Пример.

```
type complex =   record    x:real;
                           y:real
                 end

type person =    record    name:array[1..10] of character;
                           age:0..100;
                           income:integer;
                 end

type date =      record    d:1..31;
                           m:month;
                           y:year;
                 end
```

В данном примере тип данных *complex* образован из двух компонент вещественного типа. Каждая компонента обозначена соответственно именами *x* и *y*. Тип данных *person* содержит три компоненты — *name, age, income*. Этот тип обеспечивает указание имени, возраста и дохода человека. Следующий тип *date* вводится для задания дня, месяца и года. Таким образом, тип прямого произведения объединяет данные нескольких различных типов, образуя при этом новый тип данных:

Операции над данными. Для записей, так же как для массивов, определены следующие основные операции:

1) задание значений записи;
2) выборка компонент;
3) избирательное обновление компонент записи.

Задание значений записи

Если $d_i \in D_{T_i}$ является значением данных типа T_i, то значение типа записи T можно выразить следующим образом:

$$T(d_1, d_2, \ldots, d_n).$$

Например, тип *complex* имеет следующие значения:

complex(1.0, 2.56),

Выборка компонент

Выборка значения заданной компоненты записи осуществляется с помощью операции выборки, которая вводится через имя заданной компоненты. Значение компоненты s_i в значении r записи типа T в общем случае указывается как $r.s_i$ где $.s_i$ является отображением следующего вида:

$$.s_i : D_T \rightarrow D_{T_i}$$

При этом справедливы следующие выражения:

$$T(d_1, d_2, \ldots, d_n).s_i = d_i,$$

$$T(d.s_1, d.s_2, \ldots, d.s_n) = d.$$

Избирательное обновление компонент

Замену одного значения компоненты s_i в значении r записи типа T на другое, например b, можно выразить как (r, s_i, b).

Аналогично массивам в процедурных языках программирования изменение в r значения компоненты s_i на (r, s_i, b) можно реализовать с помощью следующего оператора присваивания: $r.s_i := b$.

Операция избирательного обновления является функцией следующего вида: $(\_, s_i, \_) : D_T \times D_{T_i} \rightarrow D_T$

Массивы и записи. Отличие между массивом и прямым произведением, как уже было сказано, состоит в том, что массив формируется из компонент одного типа, а компоненты записи могут быть различного типа. Данное отличие отражено также в способах выборки их составляющих. Индекс массива может вычисляться по некоторой формуле, причем значение индекса не определено до момента его вычисления. В этом случае элемент массива определяется перед его выборкой. Перед выполнением программы имеется возможность контроля типов, для чего оператор выборки должен содержать правильное имя компоненты записи.

1.5.5.3. Объединение

Объединение представляет собой множество, отдельные элементы которого классифицируются по категориям. Например, на земном шаре имеются различные виды животных. Существуют млекопитающие, птицы, рыбы и другие животные различных видов. Животные, принадлежащие соответствующему виду, обладают характерными этому виду свойствами. Наиболее подходящим способом отображения сведений о животных некоторого вида является использование для их представления набора атрибутов. В принципе можно объединить наборы атрибутов, свойственных каждому виду животных, и представить сведения о животных некоторого вида, используя весь этот набор атрибутов. Однако при таком представлении будет много неопределенных значений данных и трудно выразить семантику объекта.

Для представления сведений об отдельных категориях обобщенного объекта целесообразно использовать набор атрибутов, соответствующий данной категории. Например, для того чтобы обеспечить возможность подсчета расселения всех животных на земном шаре за прошедшие 10 лет, желательно представить сведения о расселении животных каждого вида, объединенных в обобщенный объект «животные». Для этого необходимо следующее определение:

животные = млекопитающие + рыбы + птицы + ...

Размеченным объединением называется объединение в одно целое при сохранении индивидуальности каждой категории.

Объединение издавна являлось одним из основных понятий в математике. Например, размеченное объединение множеств определяется следующим образом. *Размеченное объединение двух множеств S_1 и S_2, равное $S_1 \oplus S_2$, есть объединение элементов этих множеств, позволяющее распознать принадлежность элементов множества-суммы к соответствующему множеству-слагаемому:*

$$S_1 \oplus S_2 = \{(s,1) \mid s \in S_1\} \cup \{(s,2) \mid s \in S_2\}.$$

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

T_1 и T_2 будем называть соответственно компонентами T. Например, рассмотрим тип данных *coordinate,* выражающий координаты точки на плоскости. В большинстве случаев координаты задаются в ортогональной системе координат:

type *coordinate1* = **record**
x, y : real
end

Однако в некоторых случаях лучше применять полярные координаты, для выражения которых используется тип *coordinate2:*

type *coordinate2* = **record**
r:real;
θ : *angle*
end

Если существуют данные в двух системах координат, то можно определить тип данных *coordinate* как прямую сумму *coordinate1* и *coordinate2:*

type *coordinate* = **union**
Cartesian: coordinate1 ;

Polar: coordinate2
end

Можно не использовать типы *coordinate1* и *coordinate2*, задав следующее определение типа *coordinate:*

type *coordinate* = **union**
Cartesian: **record**
x, y : real
end
Polar: **record**
r:real;
θ : angle
end

end

В языке Паскаль размеченное объединение представляют как *запись с вариантами*

type *ckind* = *(Cartesian, Polar);*
coordinate = **record** *ckind* **of**
Cartesian: (x,y:real);
Polar: (r:real;θ:angle)
end

Множество данных типа объединения T можно задать с помощью следующего выражения:

$$D_T = \{t_1 : d_1 \mid d_1 \in T_{T_1}\} \cup \{t_2 : d_2 \mid d_2 \in T_{T_2}\} \cup \ldots$$

Таким образом, значение типа объединения $t_i:d_i$ образуется из признака t_i, определяющего компоненту и значения d_i.

t_i	*Признак*
d_i	

Например, значение *coordinate* имеет следующую форму:

Cartesian
1.0
2.56

Polar	
2.5	*Значение*
0.2	

Признак

Операции над данными. Основными операциями над данными типа объединения являются:

1) операция задания объединения;
2) операция преобразования объединения к значениям составляющих.

Задание объединения

Значение d типа объединения T означает, что значениям составляющих типа T_1, T_2, ... поставлены в соответствие признаки t_1, $t_2 \ldots t_n$, и это выражается следующим образом:

$$t_i : d_i, где\ d_i \in D_{T_i}, i = 1,2,\ldots$$

В данном случае t_i можно рассматривать как функцию вида

$$t_i : D_{T_i} \rightarrow D_T .$$

Например, значения типа *coordinate* выражаются следующим образом:

Cartesian: coordinate1 (1.0, 2.56),
Polar: coordinate2 (2.5, 0.2).

Преобразование объединения к значениям составляющих

Пусть d — значение типа прямой суммы T. Для d имеет место одна из форм $t_1 : d_1, где\ d_1 \in D_{T_1}$, или $t_2 : d_2, где\ d_2 \in D_{T_2}$. При $d = t_1 : d_1$ запись называют составляющей t_1 значения d и записывают $d_1 = d.t_1$. В данном случае можно определить отображение

$$.t_1 : D_T \rightarrow D_{T_1}$$

Отображение $.t_1$ определяется следующим образом:

$$d.t_1 = \begin{cases} d_1 если\ d = t_1 : d_1 \\ не\ определено\ в\ противном\ случае. \end{cases}$$

В общем случае отображение $.t_1 : D_T \rightarrow D_{T_1}$ такого же вида. Данные отображения, если задано значение объединения, предназначены для преобразования в соответствующие значения составляющих. Между отображениями $.t_i$ и t_i существует следующая связь:

для произвольного $d_i \in D_{T_i}$ $(t_i : d_i).t_i = d_i$ для произвольного $d \in D_T$, если $d.t_i$ определено, $t_i : (d.t_i) = d$.

Оператор выборки. Для данных типа обьединение $d = t_i : d_i$ толкование значения составляющей d_i изменяется в зависимости от признака t_i, поэтому при обработке d вначале необходима обработка d_i, поскольку t_i указано. Это можно сделать, например, следующим образом:

```
var x :T
при этом
with x do
t1:statement1(x);
t2:statement2(x);
...
end
```

В данном случае проверяют значение признака x; если оно равно t_i, то выбирают оператор *statement1(x)* и выполняют его. Ниже приведен оператор выборки для расчета расстояния r от начала координат для данных d типа *coordinate*.

```
var   d:coordinate;
      z:real;
with d do
      Cartesian: z:=sqrt(d.x^2+d.y^2);
      Polar: z:=d.r;
end
```

1.5.5.4. Множество

Множество представляет собой тип данных, значениями которого является множество всех подмножеств базисного типа.

Использование этого типа данных в ряде случаев упрощает понимание текстов программ. Например, если в языке Фортран имена переменных начинаются с символов I, J, K, L, M, N, то это означает, что данные переменные имеют целый тип. В противном случае имеется неявное указание на вещественный тип. Проверку этих условий выполняет программа синтаксического анализа Фортран-компилятора. Если использовать множество *ItoN*= {"I", "J", "K", "L", "M", "N"} или {"I".. "N"}, состоящее из указанных литер, то следующая последовательность операторов является достаточно наглядной:

```
const ItoN= {"I" .. "N"};
type intreal= (integertype,realtype);
var ch : character; result: intreal;
if ch in ItoNthen result:=integertype
                 else result:=realtype
```

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:

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

Для эффективной реализации базисный тип не должен иметь большое значение основания системы счисления. Обычно базисный тип должен быть простым типом.

Операции над множествами. Ниже определены основные операции для множеств.

Задание значений элементов множества

Множество, элементами которого являются $a, b, \ldots, c \in D_{T_0}$, можно представить следующим образом: *T {a, b, ..., c}*.

Если *a, b, . .., c* образуют в D_{T_0} интервал, то получаем выражение *T {a ... c}*.

Операции над множествами

Для заданных множеств $S_1, S_2 \in D_T$ операции объединения, пересечения и разности $\cup, \cap, - : D_T \times D_T \to D_T$ определены следующим образом:

$$S_1 \cup S_2 = \{d \mid d \in S_1 \text{ или } d \in S_2\}$$
$$S_1 \cap S_2 = \{d \mid d \in S_1 \text{ и } d \in S_2\}$$
$$S_1 - S_2 = \{d \mid d \in S_1 \text{ и } d \notin S_2\}.$$

Операция отношения.

Данная операция определяется следующим образом:

$$\text{in: } D_{T_0} \times D_T \to D_{boolean}$$

Для $d \in D_{T_0}$, $S \in D_T$

$$d \text{ in } S = \begin{cases} истина, если\ d \in S \\ ложь, если\ d \notin S \end{cases}$$

Множество и оператор цикла. Если для каждого элемента x множества S необходимо выполнить какую-либо обработку, то можно использовать оператор цикла следующего вида:

for *x* **in** *S* **do** *F(x)*

Например, количество элементов множества 5 определяется следующим образом:

number :=0;
for *x* **in** *S* **do** *number:=number+1*

Следует отметит, что другие структуры данных такие как деревья, графы, отображения и последовательность (список), разновидностями которого являются файл, стек и операция над ними, а также рекурсивные, ссылочные типы данных и их использования, приведены в следующих главах (практических занятиях) методического пособия.

1.6. АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

1.6.1. Смысл абстрагирования данных

1.6.1.1. Иерархическое описание и абстрагирование данных

Повышение требований к обработке информации обусловливает необходимость разработки крупномасштабных сложных программ. Создание таких программ требует большого числа людей и времени. Для получения программ, обладающих высокой надежностью, важными являются не только последовательная детализация программ, но и декомпозиция решаемой проблемы на задачи, сложность решения которых соответствовала бы возможности их эффективной реализации. В связи с этим существуют метод *иерархической декомпозиции проблем* и основанный на этом методе *иерархический способ создания программы.*

Что из себя представляет иерархический способ создания программы? 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_1 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.*

1.6.1.2. Защита данных и абстрагирование данных

С позиции обработки данных построение абстрактных типов данных можно считать способом структурирования программы. В этом смысле спецификация абстрактных типов данных является важной с точки зрения создания наглядной программы и защиты данных от некорректного использования данных. Рассмотрим приведенную ниже программу:

```
program processstack;
    const n=100;
    type stack=array [1..n] of integer;
    var     astack: stack;
     pointer:0..n;
    procedure create;
     begin pointer:=0;
     end;
    procedure push(d:integer);
     begin
           pointer:=pointer+1;
           astack[pointer]:=d;
     end;
    procedure pop;
     begin pointer:=pointer-1;
     end
    function empty: boolean;
     begin if pointer=0        then empty:=true
                               else empty:=false
     end;
    function top:integer;
     begin top:=astack[pointer]
     end;
    begin {main}
     create;
     ...
     push(5);
     ...
     pop;
     ...
    end.
```

В данном примере стек *astack* определяется вместе с группой процедур, реализующих операции *create, push, pop, empty,top.* При управлении стеком с помощью только этих процедур не возникает каких-либо трудностей, но возможен также прямой доступ к массиву *astack,* например *stack[10]* := 0. В этом случае утрачиваются все достоинства стека. В языках с абстрактным типом данных имеется возможность защиты данных от такого некорректного использования. В данном примере указано, что управление массивом осуществляется только процедурами *create, push, pop, empty, top.*

1.6.2. Методы абстрагирования данных

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

1. Определить множество значений данных $\boldsymbol{D} = \{D_t \mid t \in T\}$, 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 $f(d)$.

К настоящему времени существует много предложений по способам абстрагирования данных. В работе [13] приведены

шесть критериев, которым должны удовлетворять такие способы:

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

Формальность предполагает, что способ абстрагирования с математической точки зрения должен быть корректным, что в свою очередь означает отсутствие семантических и синтаксических неоднозначностей описаний, а также проблем, связанных с их автоматической обработкой и проверкой. Минимальность предполагает возможность определения только данных, представляющих интерес, без добавления какой-либо излишней информации. Наряду с этим предполагается отсутствие так называемой избыточной спецификации.

Методы абстрагирования данных укрупненно можно классифицировать следующим образом:

1. Прагматический метод абстрагирования.
2. Аксиоматический метод абстрагирования.
3. Алгебраический метод абстрагирования.

Данные способы могут быть применены не только для спецификации абстракций данных, но и для формального описания смысла.

1.6.3. Прагматический метод абстрагирования

В прагматическом методе абстрагирования семантику данных и операций описывают путем явного определения множества данных D_t и операций f, используя математические понятия, семантика которых считается заданной. Для этого используются либо известные, либо гипотетические языки программирования, а также такие математические понятия, как функции и множества, абстрактные механизмы типа автоматов. Ниже на примере стека рассматривается метод абстрагирования данных с использованием языка программирования. Не касаясь формы и деталей такого определения, укажем, что этот способ аналогичен в принципе способам абстрагирования данных в языках CLU, *Euclid,* Ada.

В данном способе предполагают, что язык программирования L неявно использует абстрактный тип данных. В языке L, описывают $d \in D$, и $f \in F$ и вводят механизм, исключающий возможность выполнения неправильных по отношению к d операций. Описание для стека можно сделать следующим образом (рассмотрим отличие от программы на стр. 76)

```
Абстрактные типы данных
type stack(size:integer);
    operatons create, push,pop,empty,top;
    var     astack: array[1..size] of integer;
     pointer:0..size;
    procedure create;
     begin pointer:=0;
     end;
    procedure push(d:integer);
     begin
          pointer:=pointer+1;
          astack[pointer]:=d;
     end;
    procedure pop;
     begin pointer:=pointer-1;
     end
    function empty: boolean;
     begin if pointer=0        then empty:=true
                               else empty:=false
     end;
    function top:integer;
     begin top:=astack[pointer]
     end
```

В данном описании абстрактный тип данных *stack* определяется наличием пяти операций *create, push, pop, empty, top,* семантика которых описывается явно путем задания соответствующих процедур. Множество значений данных выражается значениями, которые можно хранить в *astack,* т. е. выражается явно множеством значений данных типа **array** [1 ..*size*] **of** *integer* и множеством значений данных 0 . . *size* интервального типа.

Чтобы использовать в программе абстрактный тип данных *stack,* выполняют следующее описание переменной:

```
var S:stack(100)
 Операции над S, например
...
S.create;
...
S.push(5);
...
S.pop;
```

...

можно выполнить, используя конструкции *S.create, S.push, S.pop, S. empty, S.top,* где перед каждым оператором указано «S». Во внешней части программы доступными являются пять операций *create, push, pop, empty, top,* определенные для типа *stack,* и запрещен доступ к внутренней части соответствующих процедур, а также к переменным *astack, pointer.* Таким образом, за счет объединения в единое целое данных и относящихся к ним операций, а также запрещения доступа к этим данным в других операциях можно обеспечить защиту данных от некорректного использования. То, что на некотором уровне абстрагирования нельзя увидеть детали следующего за ним уровня, называют *скрытием информации.*

В определении *stack* параметр *size,* указывающий значение верхней границы индекса массива, в котором хранятся данные типа *stack,* является неопределенным. В описании переменной типа *stack* задается его конкретное значение (в предыдущем примере оно равно 100).

Абстрактный тип данных *stack* определяет стек, элементами которого являются значения типа *integer,* но такое определение практически полностью применимо и для любого другого типа элементов. Поэтому стек с произвольным типом элементов *element_type* можно определить следующим образом:

```
type stack(size:integer,element_type);
    operatons create, push,pop,empty,top;
    var     astack: array[1..size] of element_type;
      pointer:0..size;
    procedure create;
      ...
    procedure push(d:integer);
      ...
    procedure pop;
      ...
    function empty: boolean;
      ...
    function top:integer;
      ...
```

Естественно, что при описании переменной типа *stack* кроме значения *size* указывают также значение *element_type.*

```
var S:stack(100, integer)
```

Введение в определение типа данных возможности указания типа-параметра называют *параметризацией типа.* Параметризованные типы — это один из аспектов более общего понятия *полиморфизм.* Понятие полиморфной операции, у которой хотя бы один аргумент допускает значения не одного, а нескольких типов, встречается почти в любом языке программирования. Использование этих операций позволяет единообразно описать обработку, не зависящую от типа данных.

Как очевидно из рассмотренного материала, в случае прагматического способа абстрагирования типы данных выражаются через реализуемые в языках типы данных. Поэтому с точки зрения использования в языках программирования это очень удобно, но с точки зрения определения самих типов данных имеются недостатки, связанные с ограничением только специальными реализуемыми способами и наличием в таких способах характерных для них несущественных деталей. В рассмотренном выше примере стек выражен с помощью массива, но кроме этого стек можно выразить также и с помощью списочной структуры. В этом смысле реализуемые способы абстрагирования не обладают минимальностью.

1.6.4. Аксиоматический метод абстрагирования

В этом методе для определения абстрактного типа данных ***(D, F)*** используется логика предикатов. Благодаря универсальности логики предикатов такой метод абстрагирования является достаточно мощным, и, кроме того, он обладает минимальностью описания.

1.6.4.1. Принцип аксиоматического метода абстрагирования

В аксиоматическом методе абстрагирования тип данных описывают аналогично описанию алгебраических систем, например групп или колец, используя при этом формальную логику. Основной принцип данного метода заключается в том, что определение абстракции данных осуществляют путем описания свойств, которыми обладает абстракция данных, в виде набора аксиом, которые задают область определений и область значений каждой операции *f*, а также связи, существующие между функциями, принадлежащими *F*.

Если с помощью аксиоматического метода абстрагирования определить стек, элементами которого являются целые числа, то получим следующий набор аксиом:

A1 *stack (CREATE)*

A2 $stack\,(s) \wedge integer(i) \supset stack\,(PUSH\,(s,i))$

A3 $stack\,(s) \wedge \neg EMPTY\,(s) \supset stack\,(POP\,(s))$

A4 $stack\,(s) \wedge \neg EMPTY\,(s) \supset integer\,(TOP\,(s))$

A5 Для произвольного предиката *P*

$P\,(CREATE)$

$\wedge(\forall s)\,(\forall i)\,[stack(s) \wedge integer(i) \wedge P(s) \supset P(PUSH(s,i))]$

$\wedge(\forall s)\,[stack(s) \wedge \neg EMPTY\,(s) \wedge P(s) \supset P(POP(s))]$

$\supset(\forall s)\,(\forall i)\,[stack(s) \supset P(s)]$

A6 $stack\,(s) \wedge integer(i) \supset TOP(PUSH\,(s,i))=i$

A7 $stack\,(s) \wedge integer(i) \supset POP(PUSH\,(s,i))=s$

A8 *EMPTY(CREATE)*

A9 $stack\,(s) \wedge integer(i) \supset \neg EMPTY(PUSH(s,i))$

В приведенном выше описании *stack* и *integer* являются предикатными символами. *s* и *i* представляют собой символы переменных. *stack(s)* выражает то, что *s* является стеком, а *integer(i)* выражает то, что *i* является целым числом.

CREATE, PUSH, POP, EMPTY, TOP являются символами функций, соответствующих операциям стека *create, push, pop, empty, top. CREATE* представляет собой функцию без аргументов и является постоянной типа *stack.*

Аксиомы А1 — А4 описывают области определения и значений функций *CREATE, PUSH, POP, TOP.* Например, аксиома А2 утверждает, что если *s* является стеком, а *i* — целым числом, то *PUSH(s,i)* является стеком. Это выражает тот факт, что *push* является функцией вида

$$push : D_{stack} \times D_{integer} \rightarrow D_{stack}.$$

Аксиома А5 — это аксиома индукции, на основании которой относительно произвольного предиката *P(s)* можно сделать следующие три замечания:

1. *P (CREATE)* — истина.
2. Из истинности *P(s)* следует истинность *P(PUSH(s,i)).*
3. Из истинности *P(s)* и ¬*EMPTY (s)* следует истинность *P(POP(s)).*

Аксиома индукции в действительности выражает то, что стек формируется только с помощью аксиом А1 — А3 и ограничивает множество стеков теми стеками, которые могут быть построены из пустого стека и всех стеков, получаемых из него путем применения последовательностей операций *push* и *pop.*

Аксиома А6 определяет результат операции *top,* а А7 выражает связь между операциями *push* и *pop.* Таким образом, А6 выражает то, что если применить операцию *top* непосредственно после операции *push(s,i),* то можно получить значение *i,* а А7 означает, что если выполнить операцию *pop* сразу же после выполнения операции *push,* то стек не изменяет своего состояния. Аксиомы А8, А9 определяют операцию *empty.*

1.6.4.2. Сементика аксиоматического метода абстрагирования

Аксиомы А1 — А9 обобщают в форме аксиом стек, элементами которого являются целые числа. Они задают перечень свойств, которыми обладает стек, определяемый данными аксиомами и правилами вывода логики предикатов. Следовательно, можно считать, что при задании набора аксиом, определяющих абстракцию данных, используя стандартные приемы логики предикатов (дедукцию и теоретическое толкование модели), можно выразить смысл абстракции данных. Например, покажем, что выражение

top (pop (push (push (create, 1), 2))) = 1

является правильно построенным выражением для заданной системы аксиом:

1. *stack(CREATE)*
   A1
2. *stack(PUSH(CREATE,* 1)) — 1, *integer(1),* A2
3. *POP(PUSH(PUSH(CREATE,* 1), 2))= — 2, *integer(2),* A7
   = *PUSH(CREATE,* 1)
4. *TOP(PУ8H(CKEATE,* !))=1 — 1, *integer(1),* A6
5. *TOP(POP(PUSH(PUSH(CREATE,* 1),2)))=1 — 3, 4=

В процессе получения данного выражения с целью упрощения предполагается, что свойства, относящиеся к *integer*, уже известны, а относительно операций *CKEATE, ..., TOP* считается, что они удовлетворяют аксиомам A1 — A9. В процессе получения п. 5 из пп. 3 и 4 использованы аксиомы равенства. Это говорит о том, что аксиомы A1 — A9 предполагают использование логики предикатов с равенством.

1.6.4.3. Оценка аксиоматического метода абстрагирования

Как было рассмотрено ранее, в аксиоматическом методе абстрагирования тип данных выражают с помощью логики предикатов, являющейся наиболее мощной формальной системой, удовлетворяющей введенным ранее критериям широты области применимости, формальности и минимальности. Наиболее серьезные недостатки аксиоматического подхода связаны с критериями понимаемости и конструктивности. Устранение этих недостатков возможно при использовании более структурированного подхода, основанного на использовании многосортной логики предикатов.

1.6.5. Алгебраический метод абстрагирования

В данном подходе математическая формулировка абстракции данных базируется на формальной системе, называемой *многосортной алгеброй.* Многосортная алгебра применяется для исследования алгебраических систем, состоящих из множеств, называемых *сортами,* функций на данных множествах, а также из отношений между функциями, задаваемых в форме аксиом-равенств. В данном случае можно считать, что логика предикатов, использованная в разд. 3.4, ограничена рамками логики с равенством.

1.6.5.1. Принцип алгебраического метода абстрагирования

В алгебраическом методе абстрагирования типы данных описываются в терминах многосортной алгебры путем установления соответствия сортам переменных множеств значений данных. Данный метод сводится к заданию:

1) имени t определяемого типа данных;

2) других типов данных $t_1, \ldots, t_n$, необходимых для определения t;

3) функций $f_1, \ldots, f_m$, определяемых на множествах значений данных $t, t_1, \ldots, t_n$;

4) областей определения и областей значений функций $f_1, \ldots, f_m$;

5) отношений между функциями $f_1, \ldots, f_m$, задаваемых в форме аксиом-равенств.

Еще раз рассмотрим стек *stack*, элементами которого являются целые числа. Абстрактный тип данных *stack* алгебраически определяется следующим образом:

type *stack*

syntax

F1 *CREATE:*$\rightarrow$*stack*

F2 *PUSH: stack*$\times$*integer*$\rightarrow$*stack*

F3 *POP: stack* $\rightarrow$ *stack* U *{stackerror}*

F4 *TOP: stack → integer* U *{integererror}*
F5 *EMPTY: stack → boolean*
equations *for s:stack,i:integer*
E1 *TOP(PUSH(s,i))=i*
E2 *TOP(CREATE)=integererror*
E3 *POP(PUSH(s,i))=s*
E4 *POP(CREATE)=stackerror*
E5 *EMPTY(CREATE)*=**true**
E6 *EMPTY(PUSH(s,i))*=**false**

В блоке **syntax** определяются области значений и области определения функций, относящихся к каждой операции абстракции «стек». Представленные здесь типы, например *stack* или *integer,* отличаются от типов в случае аксиоматического способа абстрагирования и выражают множество значений данных, принадлежащих такому типу. Область определения для *CREATE* отсутствует, поскольку это константа, выражающая пустой стек. В блоке **equations** определение свойств введенных функций дано с помощью аксиом-равенств. При таком определении в форме аксиом-равенств описываются необходимые и достаточные условия, которыми должны обладать данные функции как операции абстракции «стек». Более подробных описаний не дается. *integererror* и *stackerror* выражают специальные константы, не принадлежащие определяемому типу *stack.*

1.6.5.2. Связь с аксиоматическим методом абстрагирования

Такое описание по содержанию практически аналогично аксиоматическому описанию стека, приведенному в предыдущем разделе, но имеет следующие формальные отличия:

1. В алгебраическом методе абстрагирования описание выполняется с помощью понятий, ограниченных функциями и равенствами. В противоположность этому в аксиоматическом методе описание выполняется с помощью предикатов и логических связок и выражается в виде правильно построенных выражений. С точки зрения синтаксиса алгебраический метод задания абстрактных типов данных является более однородным и простым.

2. Аксиоматический метод явно содержит аксиому индукции А5, в то время как при алгебраическом абстрагировании она отсутствует. Это объясняется тем, что при алгебраическом абстрагировании описание типа функции осуществляется в синтаксической части спецификации, откуда легко вывести аксиому индукции. Для простоты изложения рассмотрим описание стека. Пусть *P* — предикат, который должен определять стек. Если рассмотреть блок **syntax**, то функциями, у которых подлежащий определению тип *stack* является областью значений, будут *CREATE, PUSH, POP.* Среди них только *CREATE* не содержит *stack* как область определения. В связи с этим устанавливается начальное состояние стека, что соответствует истинности *P (CREATE).*

Области определения функций *PUSH, POP* содержат *stack*. Если некоторое значение стека *s* удовлетворяет *P* и значения *PUSH(s, i)* и *POP(s)* удовлетворяют *P*, т. е.

$$P(s) \supset P(PUSH(s, i)), \quad P(s) \supset P(POP(s))$$

высказывания истинны, то делается вывод, что для произвольного значения стека *s* удовлетворяется *P(s)*. Это можно записать следующим образом:

$$\frac{P(CREATE),\ P(s) \supset P(PUSH(s, i)),\ P(s) \supset P(POP(s))}{P(s)}$$

3. В рассмотренных выше пп. 1 и 2 показаны преимущества ограничения средств спецификации абстрактных типов данных при использовании алгебраического метода абстрагирования. Однако из-за этих ограничений спецификации абстрактных типов данных становятся более громоздкими. Например, если рассмотреть операцию POP, то при аксиоматическом абстрагировании невозможность применения операции POP для пустого стека можно выразить естественным образом в виде аксиомы

A3 $stack(s) \wedge \neg EMPTY(s) \supset stack(POP(s))$

в то время как при алгебраическом абстрагировании это утверждение выражается более сложным образом:

F3 $POP: stack \rightarrow stack \cup \{stackerror\}$

E4 $POP(CREATE) = stackerror$

1.6.5.3. Семантика алгебраического абстрагирования

При алгебраическом абстрагировании данных множество значений типа, подлежащего определению, и операции определяются неявно как объекты, удовлетворяющие заданной группе аксиом-равенств. В результате такого неявного определения получается описание, обладающее минимальностью, для которого определены только существенные особенности типа данных. Что же конкретно определяется таким описанием?

Один из способов состоит в рассмотрении группы аксиом-равенств, предназначенных для определения эквивалентности двух выражений, относящихся к типу данных. По существу, это аналогично тому, что равенство $Q_1 = Q_2$ по правилу замены можно рассматривать как $Q_1\ Q_2$ и $Q_2 \Rightarrow Q_1$, а доказательство эквивалентности двух выражений, относящихся к типу данных, сводится к доказательству того, что одно выражение выводится из другого. Так, если рассмотреть пример предыдущего раздела, то

$$TOP(POP(PUSH(PUSH(CREATE, 1), 2))) = TOP(PUSH(CREATE, 1))$$

E3= 1 E1

Другой способ заключается в том, что алгебраическое описание абстрактного типа непосредственно задает множество значений данных, удовлетворяющих группе аксиом-равенств. и операции над элементами множества, определяя таким образом тип данных. Для этой цели используется метод символических вычислений. Рассмотрим это на примере *stack*. Пусть

каждой операции f с областью значений *stack* соответствует символ f, а Σ — множество, состоящее из символов этих операций и целочисленных значений. Для простоты будем считать, что скобки (,) и запятая также содержатся в Σ:

$$\Sigma = \{\textbf{CREATE, PUSH, POP, (,),,}\} \cup D_{integer}$$

Множество S последовательностей символов в Σ определяется индуктивно следующим образом.

1. **CREATE** $\in S$.
2. Если $s \in S$ и $i \in D_{integer}$, то **PUSH** $(s, i) \in S$.
3. Если $s \in S$, то **POP**$(s) \in S$.
4. Никакие другие объекты не содержатся в S.

Множество S задает все операции, возможные для стека. Среди них содержится большое число различных выражений» рассматриваемых с самого начала как один и тот же стек. Например, все выражения

PUSH(CREATE, 1)
POP (PUSH(PUSH (CREATE, 1), 2))
PUSH (POP (PUSH (CREATE, 3)), 1)

соответствуют одному и тому же состоянию стека. Отношение эквивалентности $\equiv$ на S определяется следующим образом.

Если равенство, относящееся к $s_1 \equiv s_2 \Leftrightarrow$ *stack*, рассматривать как правило замены для последовательностей символов операций, то при одной и той же последовательности символов возможна замена s_1 на s_2 или s_2 на s_1.

Отношение $\equiv$ задает на S классы эквивалентности. В таком случае множество значений данных D_{stack} абстрактного типа определяется как множество классов эквивалентности на S, т. е.

$$D_{stack} = S / \equiv = \{[s] s \in S\}$$

Здесь [s] выражает класс эквивалентности, который содержит S. Таким образом,

$$[s] = \{s' \mid s' \in S \; u \; s \equiv s'\}.$$

Операции на D_{stack} определяются следующим образом.

$CREATE$=**[CREATE]**

$PUSH\ ([s],i)=[PUSH(s,i)]$

$$POP([s]) = \begin{cases} stac\,\mathrm{ker}\,ror & CREATE \in [s] \\ [POP(s)] & \textit{в противном случае} \end{cases}$$

$$TOP([s]) = \begin{cases} \mathrm{int}\,egerror & CREATE \in [s] \\ i & \textit{в противном случае.Здесь i определяется из } PUSH(...,i) \in [s] \end{cases}$$

$$EMPTY([s]) = \begin{cases} true & CREATE \in [s] \\ false \end{cases}$$

(Читатель может сам проверить, что вышеуказанные операции определены корректно, исходя из определения отношения эквивалентности.)

Очевидно, что определенное таким образом множество значений данных D_{stack} и операции удовлетворяют заданному множеству равенств. Существуют и другие D_{stack} и операции, удовлетворяющие этим равенствам, но в данной книге они не рассматриваются.

1.7. Этапы разработки программ

Процесс создания компьютерной программы для решения какой-либо практической задачи состоит из нескольких этапов.

1. ***Формализация и создание технического задания на исходную задачу.***

Практически любую область математики или других наук можно привлечь к построению модели определенного круга задач. Для задач, числовых по своей природе, можно построить модели на основе общих математических конструкций, таких как системы линейных уравнений, дифференциальные уравнения. Для задач с символьными или текстовыми данными можно применить модели символьных последовательностей или формальных грамматик.

2. ***Разработка алгоритма решения задачи.***

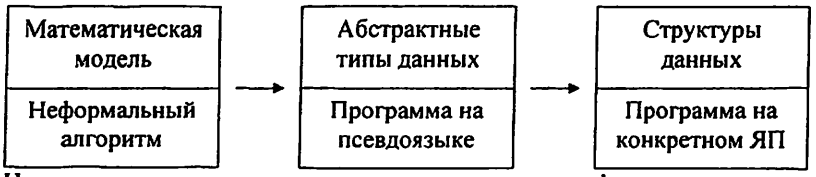
Алгоритм – это строго детерминированная система правил для решения поставленной задачи. Алгоритм должен состоять из конечной последовательности инструкций, каждая из которых должна иметь чёткий смысл и выполняться за конечное время.

3. ***Написание, тестирование, отладка и документирование программы;***

Для преобразования 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.

4. ***Получение решения исходной задачи путем выполнения законченной программы.***

Таким образом, последовательность этапов разработки программы представлена на следующем рисунке:



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.

Под *временной сложностью алгоритма* понимается "время" выполнения алгоритма, измеряемое в "шагах" (инструкциях алгоритма), которые необходимо выполнить алгоритму для достижения запланированного результата.

Часто время выполнения программы зависит от объёма входных данных. Обычно говорят, что время выполнения программы имеет порядок $T(n)$ от входных данных размера n. Единица измерения $T(n)$ точно не определена, но мы будем понимать $T(n)$ как количество инструкций, выполняемых на идеализированном компьютере.

Для описания скорости роста функций используется O-символика.

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* $f(n)$.

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 $T(n) \geq cg(n)$).

Правило сумм. 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 $O(max(f(n), g(n)))$.

Правило произведений. 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 $O(f(n)g(n))$.

1.7.1. Правила анализа:

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 $O(1)$.
2. Время выполнения последовательности операторов определяется с помощью правила сумм.
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.

5. 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*.

Пример: **метод пошаговой детализации.** Разработать программу, которая с заданной точностью e находит значение аргумента *x* по заданному значению функции *y* при известном значении *n*

$$y = \frac{(x+1)^n + 1}{x}.$$

где n>1, x>0.

При *n*>1 данная функция является монотонно возрастающей. Для нахождения значения *x* можно применить метод половинного деления. Суть данного метода заключается в следующем. Вначале определяют отрезок [*x1*, *x2*] такой, что $f(x1) \le y \le f(x2)$. Затем делят его пополам $x1 = (x1+x2)/2$ и определяют, в какой половине отрезка находится *x*, для чего сравнивают *f(x1)* и *y*. Полученный отрезок опять делят пополам и так до тех пор, пока разность *x1* и *x2* не станет меньше заданного значения *e*.

Для разработки алгоритма программы используем метод пошаговой детализации.

Шаг 1. Определяем общую структуру программы.
Ввести y, n, eps;
Определить x;
Вывести x, y;
end.

Шаг 2. Детализируем операцию определения x.
Ввести y, n, eps;
Определить x1 такое, что $f(x1) \le y$;
Определить x2 такое, что $f(x2) \ge y$;
Определить x на интервале [x1, x2];
Вывести x, y;
end.

Шаг 3. Детализируем операцию определения x1. Значение x1 должно быть подобрано так, чтобы выполнялось условие $f(x1) \le y$. Известно, что $x > 0$, следовательно, можно взять некоторое значение x, например, x1=1, и

последовательно уменьшая его, например в два раза, определить значение xl, удовлетворяющее данному условию.

```
Ввести y, n, eps.
xl:=l;
while f(xl) > y do
xl:=xl/2;
Определить x2 такое, что f(x2) ≥ y;
Определить x на интервале [xl, x2];
Вывести x, y;
end.
```

Шаг 4. Детализируем операцию определения x2. Значение x2 определяем аналогично xl, но исходное значение будем увеличивать в два раза.

```
Ввести y, n, eps;
xl:=l;
while f(xl) > y do
xl:=xl/2;
x2:=1;
while f(x2) < y do
x2:=x2*2;
Определить x на интервале [xl, x2];
Вывести x, y;
end.
```

Шаг 5. Детализируем операцию определения x. Определение x выполняется последовательным сокращением отрезка [xl, x2].

```
Ввести y, n, eps;
xl:=l;
while f(xl) > y do
xl:=xl/2;
x2:=1;
while f(x2) < y do
x2:=x2*2;
while  x2-xl>eps do
Сократить отрезок [xl, x2];
Вывести x, y;
end.
```

Шаг 6. Детализируем операцию сокращения интервала определения x. Сокращение отрезка достигается делением пополам и отбрасыванием половины, не удовлетворяющей условию f(xl) < y < f(x2)

```
Ввести y, n, eps;
xl:=l;
while f(xl) > y do
xl:=xl/2;
```

```
x2:=1;
while f(x2) < y do
x2:=x2*2;
while  x2-xl>eps do
begin
xt := (xl + x2)/2;
if f(xt) > y
then  x2 := xt
else xl := xt;
end;
Вывести x, y;
end.
```

На дальнейших шагах можно детализировать ввод и вывод данных, и в результате мы получим готовую программу.

Пример 2: **оценка временной сложности алгоритма.** Пусть задан набор из n попарно неравных отрезков. Требуется определить количество всевозможных «троек» из этих отрезков, которые могут составить невырожденные треугольники.

```
function trcount(a:integerarray; n:integer):integer;
var i,j,k,t:integer;
begin
(1)    t:=0;
(2)   for i:=1 to n do
(3) for j:=i+1 to n do
(4)   for k:=j+1 to n do
(5)   if (a[i]+a[j]>a[k])and(a[j]+a[k]>a[i])and(a[k]+a[i]>a[j])
(6)            then t:=t+1;
(7)   trcount:=t;
end;
```

Присваивание в строке (1) имеет порядок O(1) (см. правило 1).

Условное выражение внутри вложенных циклов (5) имеет время выполнения, равное сумме времени выполнения вычисления условия и наибольшего времени выполнения ветвей. Вычисление условия имеет порядок O(1). Ветвь then содержит оператор присваивания (6), который имеет порядок O(1) согласно правилу 1. Ветвь else пуста. Тогда согласно правилу 3 получаем общее время выполнения условного оператора O(1).

Для цикла в строке (4) время выполнения, согласно правилу 4, равно количеству итераций цикла, умноженному на время одной итерации. Максимальное количество возможных итераций здесь (n-2), а значит, время выполнения цикла имеет порядок O(n-2).

Для цикла в строке (3) аналогично получаем временную сложность O(n-1)*O(n-2), или по правилу произведений: O((n-1)(n-2)).

И, наконец, для цикла в строке (2) – O(n(n-1)(n-2)).

Строка (7) имеет временную сложность O(1) по правилу 1.

Тогда по правилу 2 имеем, что общая временная сложность алгоритма O(n(n-1)(n-2)), или $O(n^3-3n^2+2n)$, что приближённо можно считать трудоёмкостью $O(n^3)$.

1.8. Анализ программ на псевдоязыке

Если мы знаем степень роста времени выполнения операторов, записанных с помощью неформального "человеческого" языка, то, следовательно, сможем проанализировать программу на псевдоязыке (такие программы будем называть псевдопрограммами). Однако часто мы не можем в принципе знать время выполнения той части псевдопрограммы, которая еще не полностью реализована в формальных операторах языка программирования. Например, если в псевдопрограмме имеется неформальная часть, оперирующая абстрактными типами данных, то общее время выполнение программы в значительной степени зависит от способа реализации АТД. Это не является недостатком псевдопрограмм, так как одной из причин написания программ в терминах АТД является возможность выбора реализации АТД, в том числе по критерию времени выполнения.

При анализе псевдопрограмм, содержащих операторы языка программирования и вызовы процедур, имеющих неформализованные фрагменты (такие как операторы для работы с АТД), можно рассматривать общее время выполнения программы как функцию пока не определенного времени выполнения таких процедур. Время выполнения этих процедур можно параметризовать с помощью "размера" их аргументов. Так же, как и "размер входных данных", "размер" аргументов — это предмет для обсуждения и дискуссий. Часто выбранная математическая модель АТД подсказывает логически обоснованный размер аргументов. Например, если АТД строится на основе множеств, то часто количество элементов множества можно принять за параметр функции времени выполнения. В последующих главах вы встретите много примеров анализа времени выполнения псевдопрограмм.

Задание

1. Определите типы и структуры данных
2. Написать программу для решения задачи согласно варианту.
3. Оценить ее временную сложность при помощи О-символики.

1. Напишите программу для решения системы из n линейных уравнений методом Гаусса.

2. Напишите программу для нахождения значения интеграла функции $y=ax^2+bx+c$ на заданном интервале методом трапеций (при этом делить интервал на n частей).

3. Напишите программу для нахождения значения интеграла функции $y=ax^2+bx+c$ на заданном интервале методом Симпсона (при этом делить интервал на n частей).

4. Напишите программу для нахождения значения интеграла функции y=tg x на заданном интервале методом средних прямоугольников (при этом делить интервал на n частей).

5. Напишите программу для нахождения минимума функции y=sin x на заданном интервале с точностью *(длина интервала)/n* методом дихотомии.

6. Напишите программу для нахождения седловой точки матрицы размером nxn. Седловой точкой называется элемент, максимальный в строке и минимальный в столбце.

7. Заданы два массива целых чисел: a и b. Найти все элементы массива a, встречающиеся в массиве b (оба массива длиной n).

8. Заданы два массива вещественных чисел: a и b. Найти все элементы массива a, не встречающиеся в массиве b (оба массива длиной n).

9. Дан массив вещественных чисел длиной n. Найти в этом массиве элемент, встречающийся наиболее часто.

10. Дан массив вещественных чисел длиной n. Найти в этом массиве наиболее редко встречающийся элемент.

11. Дана матрица размерности nxn. Найти в этой матрице элемент, встречающийся наиболее часто.

12. Дана матрица размерности nxn. Найти в этой матрице элемент, встречающийся наиболее редко.

13. Дана матрица размерности nxn. Проверить, является ли матрица вырожденной. Вырожденной называется матрица, определитель которой равен нулю. Признак вырожденности матрицы: соответствующие элементы хотя бы двух её строк пропорциональны друг другу (то есть можно умножить одну из строк на число, и получится другая строка).

14. Найти все простые числа, не превосходящие заданное число n.

15. Найти n простых чисел, начиная с числа 2.

16. Вывести n первых строк треугольника Паскаля. Треугольник Паскаля имеет следующий вид:

```
1
1  1
1  2      1
1  3  3  1
1  4  6  4  1        и т. д.
```

17. Напишите программу для шифрования текста длиной n символов, состоящего из строчных латинских букв, методом простой моноалфавитной замены. Метод простой замены заключается в том, что каждая буква шифруемого текста заменяется другой буквой того же алфавита. Одним из вариантов метода является шифрование, осуществляемое по следующей формуле: номер_буквы_шифротекста = (a*номер_буквы_исходного_текста + b) mod длина_алфавита.

18. Напишите программу для расшифрования текста длиной n, состоящего из строчных латинских букв, зашифрованного методом простой замены.

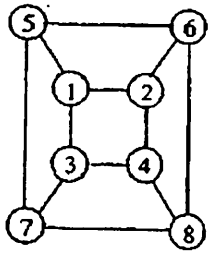
19. Напишите программу для шифрования текста длиной n, состоящего из строчных латинских букв, методом таблиц Вижинера. Таблица Вижинера представляет собой матрицу размерности, равной длине алфавита, где в первую строку вписывается сам алфавит, а каждая следующая строка получается циклическим сдвигом вправо на 1 позицию. При шифровании используется ключевое слово. Буква шифротекста получается в таблице на пересечении столбца, соответствующего букве исходного текста, и строки, соответствующей очередной букве ключевого слова.

20. Напишите программу для расшифрования текста длиной n, состоящего из строчных латинских букв, зашифрованного методом таблиц Вижинера.

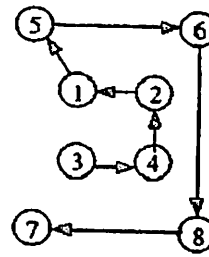
21. Напишите программу для шифрования текста длиной n, состоящего из строчных латинских букв, методом перестановок с использованием матрицы 4x4. Текст вписывается в матрицу по строкам и считывается по столбцам.

22. Расшифрование текста длиной n, состоящего из строчных латинских букв, зашифрованного методом перестановок с использованием матрицы 4x4.

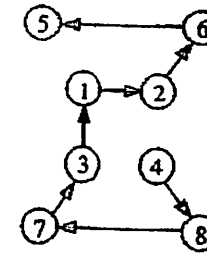
23. Напишите программу для шифрования текста длиной n, состоящего из строчных латинских букв, методом маршрутов Гамильтона. Текст вписывается в таблицу и считывается по соответствующему маршруту, например:



Таблица



Маршрут № 1



Маршрут № 2

24. Расшифрование текста длиной n, состоящего из строчных латинских букв, зашифрованного методом маршрутов Гамильтона.

25. На нитку нанизано n бусинок, которые пронумерованы от 1 до n. Если снимать по кругу, начиная с 1-й, каждую 5-ю бусинку, не разрывая нитку, то какая бусинка останется последней?

26. Многоугольник на плоскости задан n вершинами с координатами (x_i, y_i). Определить, является ли этот многоугольник выпуклым.

27. n прямоугольников на плоскости заданы точками концов одной из своих диагоналей, а стороны их параллельны осям координат. Найти количество прямоугольников, не пересекающихся ни с одним другим прямоугольником.

28. n кругов на плоскости заданы координатами центра (x_i, y_i) и радиусом r_i. Найти количество кругов, пересекающихся хотя бы с двумя другими кругами.

29. Заданы n точек на плоскости при помощи координат (x_i, y_i). Вывести все возможные комбинации из 3 точек, являющиеся треугольниками. Треугольником является совокупность 3 точек, не лежащих на одной прямой.

30. Треугольник в пространстве задан координатами x_i, y_i, z_i своих вершин. Дано n точек. Приняв направление луча зрения вдоль оси Oz из точки начала координат, проверить, является ли каждая из этих точек видимой, если треугольник непрозрачен. Точка является видимой, если она лежит в пространстве ближе заданного треугольника к началу координат или же её проекция на плоскость xOy лежит вне соответствующей проекции треугольника.

Практическое занятие - 2.

Тема: Представление списка с помощью различных структур (массив, указатели, на основе курсоров) и их анализ.

Цель работы: ознакомление с способами представление списка с помощью различных структур (массив, указатели, на основе курсоров) и их анализ а также их применением для решения различных задач.

В результате выполнения практической работы студенты должны:

- *знать* понятия абстрактного типа данных, структуры данных и типа данных; структуры и алгоритмы для АТД «Список»;
- *уметь* применять вышеназванные типы данных при решении задач.

Изучим некоторые из наиболее общих абстрактных типов данных (АТД). Мы рассмотрим списки (последовательности элементов) и два специальных случая списков: стеки, где элементы вставляются и удаляются только на одном конце списка, и очереди, когда элементы добавляются на одном конце списка, а удаляются на другом. Мы также кратко рассмотрим отображения — АТД, которые ведут себя как функции. Для каждого из этих АТД разберем примеры их реализации и сравним по нескольким критериям.

2.1. 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. Методы управления памятью, которые мы обсудим в главе 12, широко используют технику обработки списков. В этом разделе будут описаны основные операции, выполняемые над списками, а далее мы представим структуры данных для списков, которые эффективно поддерживают различные подмножества таких операций.

В математике *список* представляет собой последовательность элементов определенного типа, который в общем случае будем обозначать как elementtype (тип элемента). Мы будем часто представлять список в виде последовательности элементов, разделенных запятыми: $a_1, a_2, \ldots, a_n$, где $n > 0$ и все a_i имеют тип elementtype. Количество элементов n будем называть *длиной списка*. Если $n > 1$, то a_1 называется *первым элементом*, а a_n — *последним элементом* списка. В случае $n = 0$ имеем *пустой список*, который не содержит элементов.

Важное свойство списка заключается в том, что его элементы можно линейно упорядочить в соответствии с их позицией в списке. Мы говорим, что

элемент a_i *предшествует* a_{i+1} для $i = 1, 2, ..., n - 1$ и a_i *следует за* a_{i-1} для $i = 2, 3, ..., n$. Мы также будем говорить, что элемент a_i *имеет позицию* i. Кроме того, мы постулируем существование позиции, следующей за последним элементом списка. Функция END(L) будет возвращать позицию, следующую за позицией n в n-элементном списке L. Отметим, что позиция END(L), рассматриваемая как расстояние от начала списка, может изменяться при увеличении или уменьшении списка, в то время как другие позиции имеют фиксированное (неизменное) расстояние от начала списка.

Для формирования абстрактного типа данных на основе математического определения списка мы должны задать множество операторов, выполняемых над объектами па LIST[1] (Список). Однако не существует одного множества операторов, выполняемых над списками, удовлетворяющего сразу все возможные приложения. (Это утверждение справедливо и для многих других АТД, рассматриваемых в этой книге.) В этом разделе мы предложим одно множество операторов, а в следующем рассмотрим несколько структур для представления списков и напишем соответствующие процедуры для реализации типовых операторов, выполняемых над списками, в терминах этих структур данных.

Чтобы показать некоторые общие операторы, выполняемые над списками, предположим, что имеем приложение, содержащее список почтовой рассылки, который мы хотим очистить от повторяющихся адресов. Концептуально эта задача решается очень просто: для каждого элемента списка удаляются все последующие элементы, совпадающие с данным. Однако для записи такого алгоритма необходимо определить операторы, которые должны найти первый элемент списка, перейти к следующему элементу, осуществить поиск и удаление элементов.

Теперь перейдем к непосредственному определению множества операторов списка. Примем обозначения: L — список объектов типа elementtype, x — объект этого типа, p — позиция элемента в списке. Отметим, что "позиция" имеет другой тип данных, чья реализация может быть различной для разных реализаций списков. 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. Таким образом, если список L состоит из элементов $a_1, a_2, ..., a_n$, то после выполнения этого оператора он будет иметь вид $a_1, a_2, ..., a_{p-1}, x, a_p, ..., a_n$. Если p принимает значение END(L), то будем иметь $a_1, a_2, ..., a_n, x$. 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. Так, если список L состоит из элементов a_1 , a_2, ..., a_n, то после выполнения этого оператора он будет иметь вид a_1 , a_2, ..., a_{p-1},a_{p+1},..., a_n,. 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.

Пример 2.1. Используя описанные выше операторы, создадим процедуру PURGE (Очистка), которая в качестве аргумента использует список и удаляет из него повторяющиеся элементы. Элементы списка имеют тип elementtype, а список таких элементов имеет тип LIST (данного соглашения мы будем придерживаться на протяжении всей этой главы). Определим функцию *same(x, y)*, где x и y имеют тип elementtype, которая принимает значение true (истина), если x и y "одинаковые" (same), и значение false (ложь) в противном случае. Понятие "одинаковые", очевидно, требует пояснения. Если тип elementtype, например, совпадает с типом действительных чисел, то мы можем положить, что функция *same(x, y)* будет иметь значение true тогда и только тогда, когда $x = y$. Но если тип elementtype является типом записи, содержащей поля почтового индекса *(acctno)*, имени *(name)* и адреса абонента *(address)*, этот тип можно объявить следующим образом:

```
type
    elementtype = record
        acctno:  integer;
        name:  packed array[1..20] of char;
        address:  packed array[1..50]of
    char;
    end
```

[1] Строго говоря, тип объекта определен словами "список элементов типа elementtype". Однако мы предполагаем, что реализация списка не зависит от того, что подразумевается под "elementtype", — именно это обстоятельство объясняет то особое внимание, которое мы уделяем концепции списков. Поэтому мы будем использовать термин "тип LIST", а не "список элементов типа elementtype", и в таком же значении будем трактовать другие АТД, зависящие от типов элементов.

Теперь можно задать, чтобы функция *same(x, y)* принимала значение true всякий раз, когда *x.acctno = y.acctno*[1].

В листинге 2.1 представлен код процедуры PURGE. Переменные *p* и *q* используются для хранения двух позиций в списке. В процессе выполнения программы слева от позиции *p* все элементы уже не имеют дублирующих своих копий в списке. В каждой итерации цикла (2) - (8) переменная *q* используется для просмотра элементов, находящихся в позициях, следующих за позицией *p*, и удаления дубликатов элемента, располагающегося в позиции *p*. Затем *p* перемещается в следующую позицию и процесс продолжается.

Листинг 2.1. Программа удаления совпадающих элементов

```
procedure PURGE ( var L: LIS);
    var
        p, q: position;
        { p -- "текущая" позиция в списке L, q
        перемещается вперед от позиции p }
    begin
        p:= FIRST(L);
        while p <> END(L) do
            begin
            q:=NEXT(p,L);
            while q<>END(L)
            do
                if same(RETRIEVE(p,L),RETRIEVE(q,L))then
                    DELETE(q,L)
                else
                    q:=
                NEXT(q,L);
                    p:= NEXT(p,L)
                end
    end;  {  PURGE  }
```

В следующем разделе мы покажем объявления типов LIST и position (позиция) и реализацию соответствующих операторов, так что PURGE станет вполне работающей

Необходимо сделать замечание о внутреннем цикле, состоящем из строк (4) - (8) листинга 2.1. При удалении элемента в позиции *q*, строка (6), элементы в позициях *q* + 1, *q* + 2, ... и т.д. переходят на одну позицию влево. Иногда может случиться, что *q* является последней позицией в списке, тогда после удаления элемента позиция *q* становится равной END(L). Если теперь мы выполним оператор в строке (7), NEXT(END(L), L), то получим

[1] Конечно, если мы собираемся использовать эту функцию для удаления совпадающих записей, то в этом случае необходимо проверить также равенство значений полей имени и адреса.
программой. Как уже указывалось, программа не зависит от способа представления списков, поэтому мы свободны в экспериментировании с различными реализациями списков.

неопределенный результат. Но такая ситуация невозможна, так как между очередными проверками q <> END(L) в строке (4) может выполниться или строка (6), или строка (7), но не обе сразу.

2.2. Реализация списков

В этом разделе речь пойдет о нескольких структурах данных, которые можно использовать для представления списков. Мы рассмотрим реализации списков с помощью массивов, указателей и курсоров. Каждая их этих реализаций осуществляет определенные операторы, выполняемые над списками, более эффективно, чем другая.

2.2.1. Реализация списков посредством массивов

При реализации списков с помощью массивов элементы списка располагаются в смежных ячейках массива. Это представление позволяет легко просматривать содержимое списка и вставлять новые элементы в его конец. Но вставка нового элемента в середину списка требует перемещения всех последующих элементов на одну позицию к концу массива, чтобы освободить место для нового элемента. Удаление элемента также требует перемещения элементов, чтобы закрыть освободившуюся ячейку.

При использовании массива мы определяем тип LIST как запись, имеющую два поля. Первое поле *elements* (элементы) — это элементы массива, чей размер считается достаточным для хранения списков любой длины, встречающихся в данной реализации или программе. Второе поле целочисленного типа *last* (последний) указывает на позицию последнего элемента списка в массиве. Как видно из рис. 2.1, i-й элемент списка, если $1<i<last$, находится в i-й ячейке массива. Позиции элементов в списке представимы в виде целых чисел, таким образом, i-я позиция — это просто целое число i. Фу я END(L) возвращает значение $last+1$.

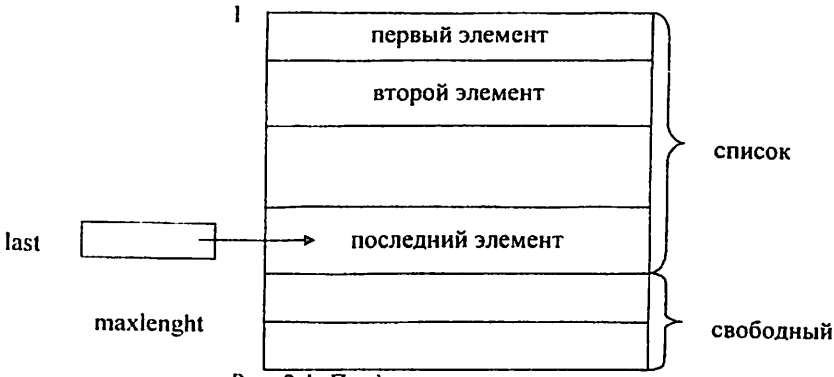


Рис. 2.1. Представление списка с помощью массива (пояснения в тексте)

Приведем необходимые объявления (константа *maxlength* определяет максимальный размер массива):

```
const
    maxlength = 100 {или другое подходящее число };
type
    LIST = record
        elements:array[1..maxlength] of elementtype;
        last:
    integer end;
    position =
    integer;

function END(var L:LIST): position;
    begin
        return(L.las
    t+1) end;   {
    END  }
```

В листинге 2.2 показано, как можно реализовать операторы INSERT, DELETE и LOCATE при представлении списков с помощью массивов. Оператор INSERT перемещает элементы из позиций *p, p* + 1, ..., *last* в позиции *p* + 1, *p*+2, ..., *last* +1и помещает новый элемент в позицию *p.* Если в массиве уже нет места для нового элемента, то инициализируется подпрограмма *error* (ошибка), распечатывающая соответствующее сообщение, затем выполнение программы прекращается. Оператор DELETE удаляет элемент в позиции *p.* перемещая элементы из позиций p+1,*p*+ 2, ..., *last* в позиции *p, p*+1, ..., *last*-1. Функция LOCATE последовательно просматривает элементы массива для поиска заданного элемента. Если этот элемент не найден, то возвращается *last*+1.

Листинг 2.2. **Реализация операторов списка**

```
procedure INSERT (x: elementtype; p: position; var L: LIST);
{ INSERT вставляет элемент x в позицию p в списке L  }
    var
        q:
    position;
    begin
        if L.last >= maxlength then
          error('Список полон')
        else if (p>L.last+1) or (p<1) then
          error('Такой позиции не
          существует')
        else begin
          for q:=L.last downto p do
          { перемещение элементов из позиций p,p+1, ... на
                одну позицию к концу списка }
                L.elements[q+1]:= L.elements[q]
          ;
                L.last:=L.last+ 1;
```

```
                    L.elements[p]:=x
                end
            end; { INSERT }
    procedure DELETE (p:position; var L:LIST );
        { DELETE удаляет элемент в позиции p списка L
        }
        var
            q:
        position;
        begin
            if (p>L.last)or (p<1) then
                error('Такой позиции не
            существует') else begin
                L.last:=L.last -
                1;
                for q:=p to L.last
                do
                    { перемещение элементов из позиций p+1, p+2, ...
                        на одну позицию к началу
                    списка } L.elements[q]:= L.elements[q+1]
            end
        end; {
        DELETE }

    procedure LOCATE ( x: elementtype; L: LIST ): position;
    { LOCATE возвращает позицию элемента x в списке L }
        var
            q: position;
        begin
            for q:=1 to L.last do
                    if L.elements[q]=x then return(q);
            return(L.last+ 1 ) {элемент x не найден}
        end; { LOCATE }
```

Легко видеть, как можно записать другие операторы списка, используя данную реализацию списков. Например, функция FIRST всегда возвращает 1, функция NEXT возвращает значение, на единицу большее аргумента, а функция PREVIOUS возвращает значение, на единицу меньшее аргумента. Конечно, последние функции должны делать проверку корректности результата. Оператор MAKENULL(L) устанавливает *L.last* в 0.

Итак, если выполнению процедуры PURGE (листинг 2.1) предшествуют определения типа elementtype и функции *same*, объявления типов LIST и position и задание функции END (как показано выше), написание процедуры DELETE (листинг 2.2), подходящая реализация простых процедур FIRST, NEXT и RETRIEVE, то процедура PURGE станет вполне работоспособной программой.

Вначале написание всех этих процедур для управления доступом к основополагающим структурам может показаться трудоемким делом. Но если мы все же подвигнем себя на написание программ в терминах операторов управления абстрактными типами данными, не задерживаясь при этом на частных деталях их реализаций, то сможем изменять программы, изменяя реализацию этих операторов, не выполняя утомительного поиска тех мест в программах, где необходимо внести изменения в форму или способ доступа к основополагающим структурам данных. Эта гибкость может иметь определяющее значение при разработке больших программных проектов. К сожалению, читатель не сможет в полной мере оценить эту сторону использования АТД из-за небольших (по размеру) примеров, иллюстрирующих эту книгу.

2.2.2. Реализация списков с помощью указателей

В этом разделе для реализации однонаправленных списков используются указатели, связывающие последовательные элементы списка. Эта реализация освобождает нас от использования непрерывной области памяти для хранения списка и, следовательно, от необходимости перемещения элементов списка при вставке или удалении элементов. Однако ценой за это удобство становится дополнительная память для хранения указателей.

В этой реализации список состоит из ячеек, каждая из которых содержит элемент списка и указатель на следующую ячейку списка. Если список состоит из элементов $a_1, a_2, \ldots, a_n$, то для $i = 1, 2, \ldots, n$-1 ячейка, содержащая элемент a_i, имеет также указатель на ячейку, содержащую элемент a_{i+1}. Ячейка, содержащая элемент a_n, имеет указатель **nil** (нуль). Имеется также ячейка *header* (заголовок), которая указывает на ячейку, содержащую a_1. Ячейка *header* не содержит элементов списка[1]. В случае пустого списка заголовок имеет указатель **nil**, не указывающий ни на какую ячейку. На рис. 2.2 показан связанный список описанного вида.

[1] Объявление ячейки заголовка "полноценной" ячейкой (хотя и не содержащей элементов списка) упрощает реализацию операторов списка на языке Pascal. Можете использовать указатели на заголовки, если хотите каким-либо специальным способом реализовать операторы вставки и удаления в самом начале списка.

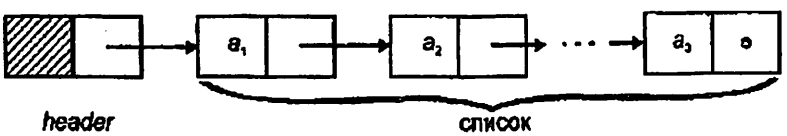


Рис. 2.2. Связанный список

Для однонаправленных списков удобно использовать определение позиций элементов, отличное от того определения позиций, которое применялось в реализации списков с помощью массивов. Здесь для i — 2, 3, ... , n позиция i определяется как указатель на ячейку, содержащую указатель на

элемент a_i. Позиция 1 — это указатель в ячейке заголовка, а позиция END(L) — указатель в последней ячейке списка L.

Бывает, что тип списка совпадает с типом позиций, т.е. является указателем на ячейку, реже на заголовок. Формально определить структуру связанного списка можно следующим образом:

```
type
    celltype = record
        element:  elementtype;
        next:
    ↑celltype end;
    LIST = ↑celltype;
    position = ↑celltype;
```

В листинге 2.3 показан код функции END(L). Результат функции получаем путем перемещения указателя q от начала списка к его концу, пока не будет достигнут конец списка, который определяется тем, что q становится указателем на ячейку с указателем nill. Отметим, что эта реализация функции END неэффективна, так как требует просмотра всего списка при каждом вычислении этой функции. Если необходимо частое использование данной функции, как в программе PURGE (листинг 2.1), то можно сделать на выбор следующее.

1. Применить представление списков, которое не использует указатели на ячейки.
2. Исключить использование функции END(L) там, где это возможно, заменив ее другими операторами. Например, условие p<>END(L) в строке (2) листинга 2.1 можно заменить условием p↑.*next* <> nill, но в этом случае программа становится зависимой от реализации списка.

Листинг 2.3. Функция END

```
function END(L: LIST): position;
{ END  возвращает указатель  на последнюю ячейку списка  L  }
    var
        q:position;
    begin
(1)     q:= L;
(2)     while q↑.next <> nil do
(3)         q:= q↑.next;
(4)     return (q)
    end;  { END }
```

Листинг 2.4 содержит процедуры для операторов INSERT, DELETE, LOCATE и MAKENULL, которые используются в реализации списков с помощью указателей. Другие необходимые операторы можно реализовать как одношаговые процедуры, за исключением PREVIOUS, где требуется просмотр списка с начала. Мы оставляем написание оцедур читателю в качестве упражнений. Отметим, что многие команды не требуют параметра L, списка, поэтому он опущен.

Листинг 2.4. Реализация некоторых операторов при представлении списков с помощью указателей

```
procedure INSERT (x: elementtype; p: position );
        var
            temp: position;
        begin
            temp:= p↑.next;
            new (p↑.next) ;
            p↑.next↑.element:= x;
            p↑.next↑.next:=  temp
        end; { INSERT }

procedure DELETE (p: position );
        begin
            p↑.next:= p↑.next↑.next
        end; { DELETE }

function LOCATE (x: elementtype; L: LIST):position;
        var
            p:position;
        begin
            p:= L;
            while p↑.next <> nil do
                if p↑.next↑.element=x then
                    return (p)
                else
                    p:= p↑.next; return(p){ элемент не найден }
        end; { LOCATE }

function MAKENULL (var L: LIST ):position;
        begin
            new(L) ;
            p↑.next:= nil;
            return(L)
        end;  { MAKENULL  }
```

Механизм управления указателями в процедуре INSERT (листинг 2.4) показан на рис. 2.3. Рис. 2.3, а показывает ситуацию перед выполнением процедуры INSERT. Мы хотим вставить новый элемент перед элементом *b*, поэтому задаём *p* как указатель на ячейку, содержащую элемент *b*. В строке (2) листинга создается новая ячейка, а в поле *next* ячейки, содержащей элемент *a*, ставится указатель на новую ячейку. В строке (3) поле *element* вновь созданной ячейки принимает значение *x*:, а в строке (4) поле *next* этой ячейки принимает значение переменной *temp*, которая хранит указатель на ячейку, содержащую элемент *b*. На рис. 2.3, б представлен результат выполнения процедуры

INSERT, где пунктирными линиями показаны новые указатели и номерами (совпадающими с номерами строк в листинге 2.4) помечены этапы из создания.

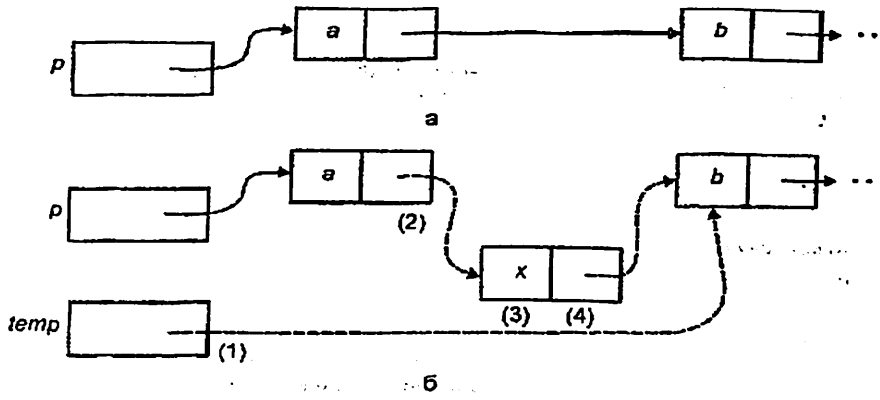


Рис. 2.3 Результат выполнения процедуры INSERT

Процедура DELETE более простая. На рис. 2.4 показана схема манипулирования указателем в этой процедуре. Старые указатели показаны сплошными линиями, а новый — пунктирной.*2.3. Диаграммы, иллюстрирующие работу процедуры INSERT*

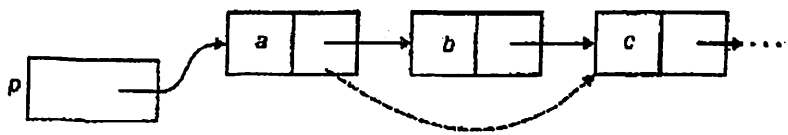


Рис. 2.4. Диаграмма, иллюстрирующая работу процедуры DELETE

Еще раз подчеркнем, что позиции в реализации однонаправленных списков ведут себя не так, как позиции в реализации списков посредством массивов. Предположим, что есть список из трех элементов *a*, *b* и *c* и переменная *p* типа position (позиция), чье текущее значение равно позиции 3, т.е. это указатель, находящийся в ячейке, содержащей элемент *b*, и указывающий на ячейку, содержащую элемент *c*. Если теперь мы выполним команду вставки нового элемента *x* в позицию 2, так что список примет вид *a*, *x*, *b*, *c*, элемент *b* переместится в позицию 3. Если бы мы использовали реализацию списка с помощью массива, то элементы *b* и *c* должны были переместиться к концу массива, так что элемент *b* в самом деле должен оказаться на третьей позиции. Однако при использовании реализации списков с помощью указателей значение переменной *p* (т.е. указатель в ячейке,

содержащей элемент *b)* вследствие вставки нового элемента не изменится, продолжая указывать на ячейку, содержащую элемент *c*. Значение этой переменной надо изменить, если мы хотим использовать ее как указатель именно на третью позицию, т.е. как указатель на ячейку, содержащую элемент *b*.

2.2.3. Сравнение реализаций

Разумеется, нас не может не интересовать вопрос о том, в каких ситуациях лучше использовать реализацию списков с помощью указателей, а когда — с помощью массивов. Зачастую ответ на этот вопрос зависит от того, какие операторы должны выполняться над списками и как часто они будут использоваться. Иногда аргументом в пользу одной или другой реализации может служить максимальный размер обрабатываемых списков. Приведем несколько принципиальных соображений по этому поводу.

1. Реализация списков с помощью массивов требует указания максимального размера k до начала выполнения программ. Если мы не можем заранее ограничить сверху длину обрабатываемых списков, то, очевидно, более рациональным выбором будет реализация списков с помощью указателей.
2. Выполнение некоторых операторов в одной реализации требует больших вычислительных затрат, чем в другой. Например, процедуры INSERT и DELETE выполняются за постоянное число шагов в случае связанных списков любого размера, но требуют времени, пропорционального числу элементов, следующих за вставляемым (или удаляемым) элементом, при использовании массивов. И наоборот, время выполнения функций PREVIOUS и END постоянно при реализации списков посредством массивов, но это же время пропорционально длине списка в случае реализации, построенной с помощью указателей.
3. Если необходимо вставлять или удалять элементы, положение которых указано с помощью некой переменной типа position, и значение этой переменной будет использовано позднее, то не целесообразно использовать реализацию с помощью указателей, поскольку эта переменная не "отслеживает" вставку и удаление элементов, как показано выше. Вообще использование указателей требует особого внимания и тщательности в работе.
4. Реализация списков с помощью массивов расточительна в отношении компьютерной памяти, поскольку резервируется объем памяти, достаточный для максимально возможного размера списка независимо от его реального размера в конкретный момент времени. Реализация с помощью указателей использует столько памяти, сколько необходимо для хранения текущего списка, но требует дополнительную память для указателя каждой ячейки. Таким образом, в разных ситуациях по критерию используемой памяти могут быть выгодны разные реализации.

2.2.4. Реализация списков на основе курсоров

Некоторые языки программирования, например Fortran и Algol, не имеют указателей. Если мы работаем с такими языками, то можно смоделировать ука-

затели с помощью курсоров, т.е. целых чисел, которые указывают на позиции элементов в массивах. Для всех списков элементов, имеющих тип elementtype, создадим один массив *SPACE* (область данных), состоящий из записей. Каждая запись будет состоять из поля *element* для элементов списка и поля *next* для целых чисел, используемых в качестве курсора. Чтобы создать описанное представление, определим

```
var
    SPACE: array [1..maxlength] of record
        element: elementtype;
        next:
    integer end
```

Для списка L объявим целочисленную переменную (например, *Lhead)* в качестве заголовка списка L. Можно трактовать *Lhead* как курсор ячейки заголовка массива *SPACE* с пустым значением поля *element.* Операторы списка можно реализовать точно так же, как описано выше в случае использования указателей.

Здесь мы рассмотрим другую реализацию, позволяющую использовать ячейки заголовков для специальных случаев вставки и удаления элементов в позиции 1. (Эту же технику можно использовать и в однонаправленных списках.) Для списка L значение *SPACE[Lhead].element* равно первому элементу списка, значение *SPACE[Lhead].next* является индексом ячейки массива, содержащей второй элемент списка, и т.д. Нулевое значение *Lhead* или нуль в поле *next* указывает на "указатель nil", это означает, что последующих элементов нет.

Список будет иметь целочисленный тип, поскольку заголовок, представляющий список в целом, является целочисленной переменной. Позиции также имеют тип целых чисел. Мы изменим соглашение о том, что позиция *i* в списке L является индексом ячейки, содержащей *(i* - 1)-й элемент списка L, поскольку поле *next* этой ячейки содержит курсор, указывающий на *i*-й элемент списка. При необходимости первую позицию любого списка можно представить посредством 0. Поскольку имя списка является обязательным параметром операторов, использующих позиции элементов, с помощью имен можно различать первые позиции различных списков. Позиция END(L) — это индекс последнего элемента списка L.

На рис. 2.5 показаны два списка, L — *a, b, c* и *M* = d, *e, f*, вложенные в один массив *SPACE* длиной 10. Отметим, что все ячейки массива, незанятые элементами списков, образуют отдельный список, называемый *свободным* (ниже в листингах он обозначается *available).* Этот список используется как "источник" пустых ячеек при вставке элементов в любой список, а также как место хранения перед дальнейшим использованием освободившихся (после удаления элементов) ячеек.

Для вставки элемента *x* в список *L* мы берем первую ячейку из свободного списка и помещаем ее в нужную позицию списка *L.* Далее элемент *x* помещается в поле *element* этой ячейки. Для удаления элемента *x* из списка *L* мы удаляем ячейку, содержащую элемент *x,* из списка и помещаем ее в начало

свободного списка.Эти два действия являются частными случаями следующей операции. Пусть есть ячейка С, на которую указывает курсор *p*, необходимо сделать так, чтобы на ячейку С указывал другой курсор *q*, курсор ячейки *С* должен указывать на ячейку, на которую ранее указывал курсор *q*, курсор *p* должен указывать на ячейку, на которую до выполнения этой операции указывал курсор ячейки С. Другими словами, надо переместить ячейку из одного места списка (указываемого курсором *p)* в другое место того же или другого списка, новое место указывается курсором *q*. Например, если необходимо удалить элемент *b* из списка *L* (рис. 2.5), то С — это строка 8 в массиве *SPACE, p* равно *SPACE[5].next*, а курсор *q* указывает на первую ячейку свободного списка.

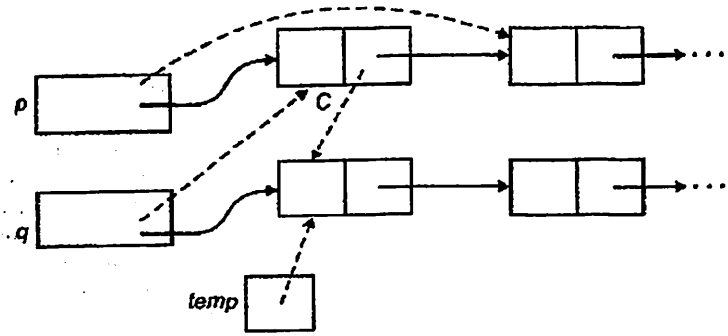


Рис. 2.5. Реализация связанных списков с использованием курсоров

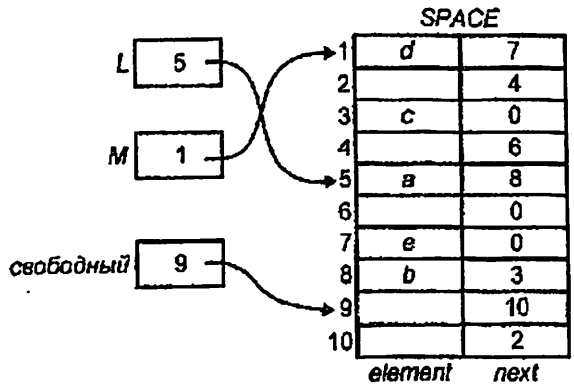


Рис. 2.6. *Перемещение ячейки из одного списка в другой*

На рис. 2.6 схематически показан процесс перемещения ячейки С из одного списка в другой (курсоры до и после выполнения операции показаны соответственно сплошными и пунктирными линиями). Код функции *move* (перемещение), выполняющей описанную операцию, приведен в листинге 2.5.

Если в массиве нет ячейки С, то функция возвращает значение false (ложь), при успешном выполнении функции возвращается значение true (истина).

Листинг 2.5. функция перемещения ячейки

```
function move (var p,q:integer ):boolean;
    var
        temp: integer;
    begin
        if p=0 then begin { ячейка не существует }
            writeln('Ячейка не существует')
            return(false)
        end
        else begin
            temp:=q;
            q:= p;
            p:=SPACE[q].next;
            SPACE[q].next:= temp;
            return(true)
        end
    end;  { move }
```

В листинге 2.6 приведены коды процедур INSERT и DELETE, а также процедуры *initialize* (инициализация), связывающей ячейки массива *SPACE* в свободный список. В этих процедурах опущены проверки "нештатных" ситуаций, которые могут вызвать ошибки (оставляем написание кода этих проверок читателю в качестве упражнений). Кроме того, в качестве упражнений оставляем читателю реализацию других операторов, выполняемых над списками (их код подобен коду аналогичных процедур при использовании указателей).

Листинг 2.6. Реализация некоторых операторов списка при использовании курсоров

```
procedure INSERT ( x: elementtype; p: position; var L: LIST );
    begin
        if p=0 then begin { вставка x в первую позицию }
            if move(available,L) then SPACE[L].element:= x
        end
        else { вставка x в позицию, отличную от первой }
            if move (available, SPACE[p].next) then
                SPACE[SPACE[p].next].element:=x
    end; {INSERT }
procedure DELETE (p:position; var L: LIST );
    begin
        if p = 0 then
            move(L,ava
        ilable) else
            move(SPACE[p].next,avai
    lable) end; { DELETE }
```

```
procedure initialize
    {initialize, связывает ячейки SPACE в один свободный список
    } var
        i:
    integer;
    begin
        for i:= maxlength -1 downto 1 do
            SPACE[i].next:=i
        +1; available:=1;
        SPACE[maxlength].nex
        t:= 0
            { помечен конец свободного
    списка }
    end; { initialize }
```

2.2.5. Дважды связные списки

Во многих приложениях возникает необходимость организовать эффективное перемещение по списку как в прямом, так и в обратном направлениях. Или по заданному элементу нужно быстро найти предшествующий ему и последующий элементы. В этих ситуациях можно дать каждой ячейке указатели и на следующую, и на предыдущую ячейки списка, т.е. организовать *дважды связный список* (рис. 2.7). В главе 12 будут приведены ситуации, когда использование дважды связных списков особенно эффективно.

Рис. 2.7. Дважды связный список

Другое важное преимущество дважды связных списков заключается в том, что мы можем использовать указатель ячейки, содержащей i-й элемент, для определения i-й позиции — вместо использования указателя предшествующей ячейки. Но ценой этой возможности являются дополнительные указатели в каждой ячейке и определенное удлинение некоторых процедур, реализующих основные операторы списка. Если мы используем указатели (а не курсоры), то объявление ячеек, содержащих элементы списка и два указателя, можно выполнить следующим образом (*previous* — поле, содержащее указатель на предшествующую ячейку):

```
type
    celltype = record
        element: elementtype;
        next,previous:↑cellty
    pe
        end;
    position = ↑celltype;
```

Процедура удаления элемента в позиции *p* дважды связного списка показана в листинге 2.7. На рис. 2.8 приведена схема удаления элемента в предположении, что удаляемая ячейка не является ни первой, ни последней в списке[1]. На этой схеме сплошными линиями показаны указатели до удаления, а пунктирными — после удаления элемента. В процедуре удаления сначала с помощью указателя поля *previous* определяется положение предыдущей ячейки. Затем в поле *next* этой (предыдущей) ячейки устанавливается указатель, указывающий на ячейку, следующую за позицией *p*. Далее подобным образом определяется следующая за позицией *p* ячейка и в ее поле *previous* устанавливается указатель на ячейку, предшествующую позиции *p*. Таким образом, ячейка в позиции *p* исключается из цепочек указателей и при необходимости может быть использована повторно.

Листинг 2.7. Удаление элемента из дважды связного списка

```
procedure DELETE ( var p: position );
    begin
        if p↑.previous <> nil then
        {удаление ячейки, которая не является первой }
          p↑.previous↑.next:= p↑.next;
        if p↑.next <> nil then
          { удаление ячейки, которая не является последней }
          p↑.next↑.previous:= p↑.previous
    end; { DELETE }
```

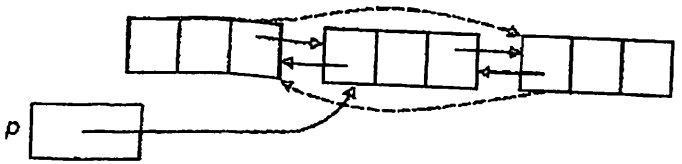


Рис. 2.8. Схема удаления ячейки в дважды связном списке

Задание

1. Программа должна по выбору пользователя осуществлять следующие действия:
   - находить среднее арифметическое элементов непустого однонаправленного списка вещественных чисел,
   - заменять все вхождения числа x на число y,
   - менять местами первый и последний элементы,
   - проверить, упорядочены ли числа в списке по возрастанию.
2. Напечатать текст в обратном порядке, напечатать слова текста в обратном порядке. Использовать курсор строк.
3. Вводятся два списка строк. Написать следующие функции:
   - сравнить списки на равенство
   - определить, входит ли первый список во второй

- скопировать в конец списка второй список

4. В списке строк подсчитать количество слов:
   - начинающихся и оканчивающихся одной и той же буквой
   - начинающихся с той же буквы, что и следующее слово
   - совпадающих с последним словом.

Каждый из пунктов должен быть оформлен как функция.

5. Дан список вещественных чисел. Написать следующие функции:
   - проверить, есть ли в нем два одинаковых элемента
   - перенести в начало его последний элемент
   - перенести в конец его первый элемент
   - вставить в список сам в себя вслед за первым вхождением числа x
6. Дан список строк. Написать следующие функции:
   - обращение списка (изменить ссылки в списке так, чтобы элементы оказались расположены в противоположном порядке)
   - из каждой группы подряд идущих элементов оставить только один
   - оставить в списке только первые вхождения одинаковых элементов
7. Даны два списка l1 и l2 пар вещественных чисел. Написать функции, возвращающие новый список l, включающий
   - пары списка l1, первая координата которых встречается как вторая координата у пар списка l2
   - пары (x,y) списка l1 встречающиеся в виде (y,x) в списке l2
   - пары (x,y) x 41.
8. Даны два списка l1 и l2 вещественных чисел. Написать функции, возвращающие новый список l, включающий по одному разу числа, которые
   - входят одновременно в оба списка
   - входят хотя бы в один из списков
   - входят в один из списков l1 и l2, но в то же время не входят в другой из них
   - входят в список l1, но не входят в список l2.
9. Объединить два упорядоченных по неубыванию списка вещественных чисел в один упорядоченный по неубыванию список. Реализовать в виде:
list* join1(list *l1, list *l2);
функция join1() возвращает новый список, не меняя списки *l1 и *l2
10. Объединить два упорядоченных по неубыванию списка вещественных чисел в один упорядоченный по неубыванию список. Реализовать в виде:
void join2(list **l1, list **l2),
функция join2() не создает нового списка, а меняет соответствующим образом ссылки в l1 и l2. После объединения списков *l1 и *l2 оба указывают на начало списка-результата.
11. Целое число представляется строкой цифр. Написать функцию, упорядочивающую по неубыванию числа в непустом списке целых чисел с S разрядами.
12. Дан список слов среди которых есть пустые. Написать функции:
   - переставляющие местами первое и последнее непустые слова

- печатать текст из первых букв непустых слов
- удалить из непустых слов первые буквы
- определить количество слов в непустом списке, отличных от последнего

13. Предположим, что уже построен и задан указателем P однонаправленный список, элементами которого являются вещественные числа. Написать программу, которая по списку P строит два новых списка: L1-из положительных элементов списка P, L2 - из отрицательных элементов списка P.

14. Написать программу, которая объединяет два упорядоченных по неубыванию списка L1 и L2 в один упорядоченный по неубыванию список путем построения нового списка.

15. Часто на практике имеет смысл некоторая перестройка линейного списка после каждого обращения к списку для поиска некоторого заданного элемента, состоящая в помещении найденного элемента в начало списка (тем самым минимизируется длина прохода по списку при повторном поиске того же элемента). Описанный метод называется поиском по списку с переупорядочиванием. Написать процедуру поиска по списку с переупорядочением в однонаправленном линейном списке.

16. Рассмотрим пары целых чисел (i,j). Говорят, что пара (i,j) меньше, чем другая пара (h,k) (записывается (i,j)<(h,k)), если либо i<h, либо (i=h) AND (j<k). Например, (-1,5)<(5,1)<(5,2). Такой порядок называется лексикографическим упорядочением пар целых чисел. Задан линейный однонаправленный список, содержащий пары целых чисел. Упорядочить его лексикографически по возрастанию пар.

Практическое занятие - 3.

Тема: Представление стеков, очередей, отображений, рекурсивных процедур. Представление деревьев с помощью массивов, списков сыновей и братьев. Двоичные деревья и их представление

Цель работы: изучение АТД «Стек», «Очередь», «Отображение», «Дерево» и применения его при решении задач.

В результате выполнения практической работы студенты должны:

- *знать* определения стека, очереди, отображения, дерева и основных понятий, связанных с ними; виды и способы представления перечисленных АТД;
- *уметь* применять АТД Стек», «Очередь», «Отображение», «Дерево» при решении различных задач.

3.1. Стеки

Стек — это специальный тип списка, в котором все вставки и удаления выполняются только на одном конце, называемом *вершиной* (top). Стеки также иногда называют "магазинами", а в англоязычной литературе для обозначения стеков еще используется аббревиатура LIFO (last-in-first-out — последний вошел — первый вышел). Интуитивными моделями стека могут служить колода карт на столе при игре в покер, книги, сложенные в стопку, или стопка тарелок на полке буфета; во всех этих моделях взять можно только верхний предмет, а добавить новый объект можно, только положив его на верхний. Абстрактные типы данных семейства STACK (Стек) обычно используют следующие пять операторов.

1. MAKENULL(S). Делает стек *S* пустым.
2. TOP(S). Возвращает элемент из вершины стека S. Обычно вершина стека идентифицируется позицией 1, тогда TOP(S) можно записать в терминах общих операторов списка как RETRIEVE(FIRST(S), S).
3. POP(S). Удаляет элемент из вершины стека (выталкивает из стека), в терминах операторов списка этот оператор можно записать как DELETE(FIRST(S), S).

Иногда этот оператор реализуется в виде функции, возвращающей удаляемый элемент.

4. PUSH(*x*, S). Вставляет элемент *x* в вершину стека *S* (заталкивает элемент в стек). Элемент, ранее находившийся в вершине стека, становится элементом, следующим за вершиной, и т.д. В терминах общих операторов списка данный оператор можно записать как INSERTS, FIRST(S), S).
5. EMPTY(S). Эта функция возвращает значение true (истина), если стек *S* пустой, и значение false (ложь) в противном случае.

3.2. Очереди

Другой специальный тип списка — *очередь* (queue), где элементы вставляются с одного конца, называемого *задним* (rear), а удаляются с другого, *переднего* (front). Очереди также называют "списками типа FIFO" (аббревиатура FIFO расшифровывается как first-in-first-out: первым вошел — первым вышел). Операторы, выполняемые над очередями, аналогичны операторам стеков. Существенное отличие между ними состоит в том, что вставка новых элементов осуществляется в конец списка, а не в начало, как в стеках. Кроме того, различна устоявшаяся терминология для стеков и очередей. Мы будем использовать следующие операторы для работы с очередями.

1. MAKENULL(Q) очищает очередь Q, делая ее пустой.
2. FRONT(Q) — функция, возвращающая первый элемент очереди Q. Можно реализовать эту функцию с помощью операторов списка как RETRIEVE(FIRST(Q), Q).
3. ENQUEUE(x, Q) вставляет элемент *x* в конец очереди *Q*. С помощью операторов списка этот оператор можно выполнить следующим образом: INSERT(x, END(Q), Q).
4. DEQUEUE(Q) удаляет первый элемент очереди Q. Также реализуем с помощью операторов списка как DELETE(FIRST(Q), Q).
5. EMPTY(Q) возвращает значение true тогда и только тогда, когда Q является пустой очередью.

3.3. Отображения

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). Тот факт, что отображение *M* ставит в соответствие элемент *d* типа domaintype из области определения элементу r типа range-type из области значений, будем записывать как *M(d) =r*.

Операторы, которые можно выполнить над отображением *M*.

1. MAKENULL(*M*). Делает отображение *M* пустым.
2. ASSIGN(*M, d, r*). Делает *M(d)* равным *r* независимо от того, как *M(d)* было определено ранее.
3. COMPUTE(*M, d, r*). Возвращает значение true и присваивает переменной *r* значение *M(d)*, если последнее определено, и возвращает false в противном случае.

Пример: **Использование АТД STACK.** Будем рассматривать последовательности открывающихся и закрывающихся круглых и квадратных скобок ( ) [ ] длиной n. Среди всех таких последовательностей выделим правильные - те, которые могут быть получены по таким правилам:

- пустая последовательность правильна.
- если A и B правильны, то и AB правильна.
- если A правильна, то [A] и (A) правильны.

Пример. Последовательности (), [[]], [()[]()][] правильны, а последовательности],)(, (], ([)] - нет.

Проверить правильность последовательности за время порядка O(n). Предполагается, что члены последовательности закодированы числами:

(1
[2
) -1
] -2

Решение. Пусть a[1]..a[n] - проверяемая последовательность.

Рассмотрим стек, элементами которого являются открывающиеся круглые и квадратные скобки (т. е. 1 и 2).

Вначале стек делаем пустым. Далее просматриваем члены последовательности слева направо. Встретив открывающуюся скобку (круглую или квадратную), помещаем её в стек. Встретив закрывающуюся, проверяем, что вершина в стеке - парная ей скобка; если это не так, то можно утверждать, что последовательность неправильна, если скобка парная, то заберем её (вершину) из стека.

Последовательность правильна, если в конце стек оказывается пуст.

```
Function Parenth(a:chararr; n:integer):boolean;
var
    s:STACK; {тип элемента -2..2}
    error:boolean;
    i:integer;
    t:-2..2;
begin
    MAKENULL (s);
    i := 0;
    error := false;
    {прочитано i символов последовательности}
    while (i < n) and not(error) do begin
            i := i + 1;
            if (a[i] = 1) or (a[i] = 2) then begin
                    PUSH(a[i], s);
            end else begin  {a[i] равно -1 или -2}
                            if EMPTY(s) then begin
                            error := true;
                            end else begin
                            t:=TOP(s);
                            POP(s);
                            error := (t <> - a[i]);
                            end;
                    end;
    end;
    Parenth:= (not error) and EMPTY(s);
    end;
```

Дерево — это совокупность элементов, называемых *узлами* (один из которых определен как *корень*), и отношений ("родительских"), образующих иерархическую структуру узлов. Узлы, так же, как и элементы списков, могут быть элементами любого типа. Мы часто будем изображать узлы буквами, строками или числами. Формально *дерево* можно рекуррентно определить следующим образом.

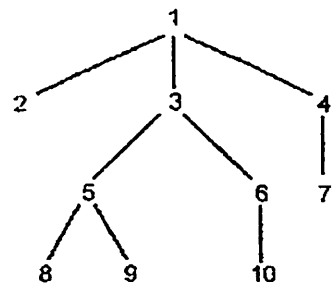
1. Один узел является деревом. Этот же узел также является корнем этого дерева.
2. Пусть n — это узел, а T_1, T_2, ..., T_k — деревья с корнями n_1, n_2, ..., n_k соответственно. Можно построить новое дерево, сделав n родителем узлов n_1, n_2, ..., n_k. В этом дереве n будет корнем, а T_1 T_2, ..., T_k — *поддеревьями* этого корня. Узлы n_1, n_2, ..., n_k называются *сыновьями* узла n.

Часто в это определение включают понятие *нулевого дерева*, т.е. "дерева" без узлов, такое дерево мы будем обозначать символом Λ.

Путем из узла n_1 в узел n_k называется последовательность узлов n_1, n_2, ..., n_k, где для всех i, $1 < i < k$, узел n_i является родителем узла n_{i+1}. *Длиной пути* называется число, на единицу меньшее числа узлов, составляющих этот путь. Если существует путь из узла a в b, то в этом случае узел a называется *предком* узла b, а узел b — *потомком* узла a. Предок или потомок узла, не являющийся таковым самого себя, называется *истинным предком* или *истинным потомком* соответственно. В дереве только корень не имеет истинного предка. Узел, не имеющий истинных потомков, называется *листом*. Теперь поддерево какого-либо дерева можно определить как узел (корень поддерева) вместе со всеми его потомками.

Высотой узла дерева называется длина самого длинного пути из этого узла до какого-либо листа. *Глубина узла* определяется как длина пути (он единственный) от корня до этого узла.

Пример дерева, упорядоченного слева направо:



Три наиболее часто используемых способа обхода дерева называются *обход в прямом порядке*, *обход в обратном порядке* и *обход во внутреннем порядке* (последний вид обхода также часто называют *симметричным обходом*, мы будем использовать оба этих названия как синонимы). Все три способа обхода рекурсивно можно определить следующим образом.

• Если дерево T является нулевым деревом, то в список обхода заносится пустая запись.

• Если дерево T состоит из одного узла, то в список обхода записывается этот узел.

• Далее, пусть T — дерево с корнем n и поддеревьями T_1, T_2, ..., T_k

• При *прохождении в прямом порядке* (т.е. при *прямом упорядочивании)* узлов дерева T сначала посещается корень n, затем узлы поддерева T_1 далее все узлы поддерева T_2, и т.д. Последними посещаются узлы поддерева T_k.

• При *симметричном обходе* узлов дерева T сначала посещаются в симметричном порядке все узлы поддерева T_1, далее корень n, затем последовательно в симметричном порядке все узлы поддеревьев T_2, ..., T_k.

• Во время *обхода в обратном порядке* сначала посещаются в обратном порядке все узлы поддерева T_1, затем последовательно посещаются все узлы поддеревьев T_2, ..., T_k, также в обратном порядке, последним посещается корень n.

Дерево, у которого узлам сопоставлены метки, называется *помеченным деревом.* Метка узла — это значение, которое "хранится" в узле.

Можно предложить большой набор операторов, выполняемых над деревьями. Здесь мы рассмотрим следующие операторы.

1. 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.
2. 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.
3. 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. Например, для дерева на рис. 3.6 LEFTMOST_CHILD(n_2) = n_4, RIGHT_SIBLING(n_4) = n_5 и RIGHT_SIBLING(n_5) =Λ.
4. LABEL(n, T). Возвращает метку узла n дерева T. Для выполнения этой функции требуется, чтобы на узлах дерева были определены метки.
5. 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.
6. ROOT(T) возвращает узел, являющимся корнем дерева T. Если T – - пустое дерево, то возвращается Λ.
7. MAKENULL(T). Этот оператор делает дерево T пустым деревом.

3.4. Представление деревьев с помощью массивов

Пусть *T* — дерево с узлами 1, 2, ..., *n*. Возможно, самым простым представлением дерева *T*, поддерживающим оператор PARENT (Родитель), будет линейный массив *A*, где каждый элемент *A[i]* является указателем или курсором на родителя узла *i*. Корень дерева *T* отличается от других узлов тем, что имеет нулевой указатель или указатель на самого себя как на родителя.

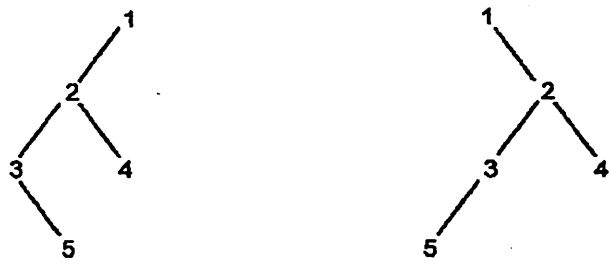
Представление деревьев с использованием списков сыновей. Здесь есть массив ячеек заголовков, индексированный номерами (они же имена) узлов. Каждый заголовок *(header)* указывает на связанный список, состоящий из "элементов"-узлов. Элементы списка *header[i]* являются сыновьями узла *i*.

Логическим продолжением представления дерева, показанного на рис. 3.8, будет замена массива заголовков на отдельный массив *nodespace* (область узлов), содержащий записи с произвольным местоположением в этом массиве.

3.5. Двоичные деревья

Двоичное дерево может быть или пустым деревом, или деревом, у которого любой узел или не имеет сыновей, или имеет либо *левого сына*, либо *правого сына*, либо обоих. Тот факт, что каждый сын любого узла определен как левый или как правый сын, существенно отличает двоичное дерево от упорядоченного ориентированного дерева.

На следующем рисунке приведены два различных двоичных дерева.



Представление двоичных деревьев. Если именами узлов двоичного дерева являются их номера 1, 2, ..., n, то подходящей структурой для представления этого дерева может служить массив cellspace записей с полями leftchild (левый сын) и rightchild (правый сын), объявленный следующим образом:

```
var
    cellspace: array[1..maxnodes] of record
        leftchild: integer;
        rightchild: integer
end;
```

В этом представлении *cellspace[i].leftchild* является левым сыном узла *i*, а *cellspace[i].rightchild* — правым сыном. Значение 0 в обоих полях указывает на то, что узел *i* не имеет сыновей.

Для указания на правых и левых сыновей (и родителей, если необходимо) вместо курсоров можно использовать настоящие указатели языка Pascal. Например, можно сделать объявление

```
type
node = record
    leftchild:  ↑node;
    rightchild: ↑node;
    parent:  ↑node
end
```

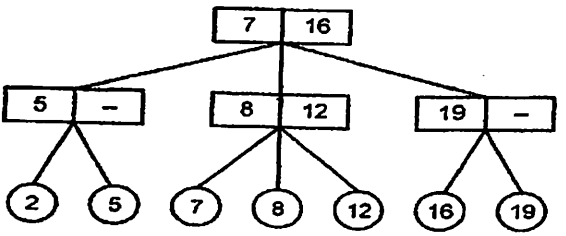
2-3 деревья. *2-3 дерево* и имеет следующие свойства.

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

Будем считать, что пустое дерево и дерево с одним узлом также являются 2-3 деревьями.

Предполагаем, что упорядочивание элементов по используемому отношению линейного порядка основывается только на значениях одного поля (среди других полей записи, содержащей информацию об элементе), которое формирует тип элементов. Это поле назовем *ключом.*

В каждый внутренний узел записываются ключ наименьшего элемента, являющегося потомком второго сына, и ключ наименьшего элемента — потомка третьего сына, если, конечно, есть третий сын. На рис. 5.6 показан пример 2-3 дерева. В этом и последующих примерах мы идентифицируем элементы с их ключевым полем, так что порядок элементов становится очевидным.



Отметим, что 2-3 дерево с k уровнями имеет от 2^{k-1} до 3^{k-1} листьев. Другими словами, 2-3 дерево, представляющее множество из n элементов, будет иметь не менее $1 + \log_3 n$ и не более $1 + \log_2 n$ уровней. Таким образом, длины всех путей будут иметь порядок $O(\log n)$.

Можно найти запись с ключом x в множестве, представленном 2-3 деревом, за время $O(\log n)$ путем простого перемещения по дереву, руководствуясь при этом значениями элементов, записанных во внутренних узлах. Во внутреннем узле *node* ключ x сравнивается со значением y наименьшего элемента, являющегося потомком второго сына узла *node.* (Напомним, что мы трактуем элементы так, как будто они состоят только из одного ключевого поля.) Если $x < y$, то перемещаемся к первому сыну узла

node. Если $x \geq y$ и узел *node* имеет только двух сыновей, то переходим ко второму сыну узла *node*. Если узел *node* имеет трех сыновей и $x \geq y$, то сравниваем x со значением z — вторым значением, записанным в узле *node*, т.е. со значением наименьшего элемента, являющегося потомком третьего сына узла *node*. Если $x < z$, то переходим ко второму сыну, если $x \geq z$, переходим к третьему сыну узла *node*.

Пример: **реализация поиска со включением в бинарном дереве.**

В этой задаче задана последовательность слов н нужно установить число появлений каждого слова. Это означает, что, начиная с пустого дерева, каждое слово ищется в дереве. Если оно найдено, увеличивается его счетчик появлений, если нет — в дерево вставляется новое слово (с .начальным значением счетчика, равным 1).

Предполагаются следующие описания типов:

```
word= record
        key: integer;
        count: integer;
        left, right: ↑word
        end
```

Считая, кроме того, что у нас есть исходный файл ключей $f_{,}$ а переменная *root* указывает на корень дерева поиска, мы можем записать программу следующие образом:

```
reset(f);
while not eof(f) do
begin read(f,x);
search(x,root)
end

  procedure search(x: integer; root: ↑word);
     var p1, p2: ↑word;
     d: integer;
   begin
  p2 := root;
  p1 := p2↑.right;
  d := 1;
       while (p1<>nil)  and  (d<>0) do
       begin p2 := p1;
        if x < p1↑.key then
            begin
          p1 := p1↑.left;
          d := - 1
            end else
       if x > p1↑.key then
            begin
                  p1 := p1↑.right;
```

```
                d := 1
            end else
                d := 0
        end ;
    if d= 0 then p1↑.count :=p1↑.count + 1 else
      begin {включение}
                new(p1);
          with p1↑ do
            begin key := x;
                left := nil;
               right := nil;
               count := 1
            end ;
    if d < 0 then p2↑.left := p1 else p2↑.right := p1
    end
end.
```

Задание

1. Вводится скобочное арифметическое выражение. Вывести его в обратной польской записи.

2. Вводится скобочное арифметическое выражение. Вывести его в прямой польской записи.

3. Вводится скобочное арифметическое выражение. В выражении могут встречаться открывающая и закрывающая круглые скобки (и), знаки операций +, -, *, / (деление - целочисленное) и цифры от 0 до 9. Найти значение этого скобочного выражения. Пример: '(3+5*2)/3-1'=3.

4. N ребят располагаются по кругу. Начав отсчет от первого, удаляют каждого k-ого, смыкая круг после каждого удаления. Определить порядок удаления ребят из круга, то есть напечатать номера ребят в том порядке, как они удаляются из круга.

5. Даны обозначения двух полей шахматной доски (например, A5 и C2). Найти минимальное число ходов, которые нужны шахматному коню для перехода с первого поля на второе.

6. Из листа клетчатой бумаги размером 8x8 клеток удалили некоторые клетки. На сколько кусков распадется оставшаяся часть листа?

7. Разработать программу сортировки потока данных с использованием 2-х стеков. Для сортировки входного потока символов используются два стека: Lstack и Hstack. Первоначально Lstack инициализируется символом с кодом 0, Hstack инициализируется символом с максимальным кодом 255. В процессе сортировки часть символов сохраняется в стеке Lstack, другая часть - в стеке Hstack. Стек Lstack предназначен для хранения данных в порядке возрастания их величин. В стеке Hstack символы хранятся в порядке убывания их кодов. При этом, код символа в вершине стека Lstack не должен превосходить по величине код символа в вершине стека Hstack:

Lstack_top <= Hstack_top

8. Составить программу сортировки потока данных на основе дека.

Дек - абстрактная структура данных, в которой разрешен стековый доступ с обоих концов последовательности данных.

9. Вокруг считающего стоит N человек, из которых выделен первый, а остальные занумерованы по часовой стрелке числами от 2 до N. Считающий, начиная с кого-то, ведет счет до M. Человек на котором остановился счет, выходит из круга. Счет продолжается со следующего человека и так до тех пор, пока не останется один человек. Определить номер оставшегося человека, если известно M и то, что счет начинался с первого человека;

10. Вокруг считающего стоит N человек, из которых выделен первый, а остальные занумерованы по часовой стрелке числами от 2 до N. Считающий, начиная с кого-то, ведет счет до M. Человек на котором остановился счет, выходит из круга. Счет продолжается со следующего человека и так до тех пор, пока не останется один человек. Определить номер человека с которого начинался счет, если известно M и номер оставшегося человека L.

11. В таблице A размера N за один просмотр необходимо каждый элемент заменить на ближайший следующий за ним элемент, который больше его. Если такого элемента нет, то заменить его на ноль. Можно использовать дополнительную память. ПРИМЕР A=1 3 2 5 3 4 ОТВЕТ A=3 5 5 0 4 0

12. Задана матрица. Пройдя все элементы матрицы один раз, найти все вхождения числа, являющегося максимальным элементом этой матрицы и вывести соответствующие индексы.. Использовать очередь для хранения индексов элементов.

13. Напечатать в порядке возрастания первые n натуральных чисел, в разложение которых на простые множители входят только числа 2, 3, 5.

14. Между двумя городами расстояние равно 50 километров, причем через каждый километр имеется остановка. Между городами ходят автобусы. Плата в автобусе зависит от количества километров, которое вы желаете проехать, причем один билет не позволяет проехать более 10 км. Вы знаете стоимость Ci, i=1,...,10 каждого билета для проезда i километров. Необходимо определить минимальною стоимость билетов, купив которые, вы доберетесь из одного города в другой. (Использовать очередь).

15. Определить структуру данных для представления n-арных деревьев и написать процедуру обхода такого дерева и построения двоичного дерева с теми же элементами.

16. Пусть при построении деревьев используется определение типа:

```
RECTYPE Tree=RECORD
x:integer;
left,right: Tree
end
```

Напишите процедуру для обнаружения элемента с заданным ключом и удаления его из дерева.

17. Напишите процедуру вставки нового элемента как потомка элемента с заданным ключом (см. задачу 2).

18. Узлы дерева помечены целыми числами. Напишите программу,

удаляющую узлы с метками, попавшими в заданный интервал.

19. В некоторой файловой системе справочник файлов организован в виде упорядоченного двоичного дерева. Каждой вершине соответствует некоторый файл, здесь содержится имя файла и, кроме всего прочего, дата последнего обращения к нему, закодированная целым числом. Напишите программу, которая обходит дерево и удаляет все файлы, последнее обращение к которым происходило до некоторой определенной даты.

20. В некоторой древовидной структуре опытным путём измеряется частота обращения к каждому из элементов. Для этого с любым элементом связан счётчик обращений. Напишите программу, которая будет строить новое дерево, в которое ключи включаются в порядке убывания счетчиков частот обращения.

21. Напишите программу поиска ключа в двоичном B-дереве.

22. Напишите программу включения элемента в двоичное B-дерево.

23. Напишите программу удаления заданного элемента из двоичного B-дерева.

24. При прохождении дерева в порядке уровней в список узлов сначала заносится корень дерева, затем все узлы глубины 1, далее все узлы глубины 2 и т.д. Узлы одной глубины заносятся в список узлов в порядке слева направо. Напишите программу обхода деревьев в порядке уровней.

25. Использовать метод обхода дерева для решения следующей задачи: дан массив из n целых положительных чисел a[1]..a[n] и число s; требуется узнать, может ли число s быть представлено как сумма некоторых из чисел массива a. (Каждое число можно использовать не более чем по одному разу.)

26. Перечислить все последовательности из n нулей, единиц и двоек, в которых никакая группа цифр не повторяется два раза подряд (нет куска вида XX).

27. Аналогичная задача для последовательностей нулей и единиц, в которых никакая группа цифр не повторяется три раза подряд (нет куска вида XXX).

28. Формулу вида терминал|формула знак формула|

i. знак - + - * /
ii. терминал - 0 1 2 3 4 5 6 7 8 9
iii. можно представить в виде двоичного дерева
iv. 1.вычислить значение дерева
v. 2.по формуле из текстового файла f построить дерево
vi. 3.напечатать дерево в виде соответствующей формулы
vii. 4.определить высоту заданного дерева

29. Напишите программу вычисления арифметических выражений при обходе дерева

a. в прямом порядке;
b. в обратном порядке.

30. Дано дерево глубины *N*, каждая внутренняя вершина которого имеет *K* (< 10) непосредственных потомков (нумеруются от 1 до *K*). Корень дерева имеет номер 0. Записать в текстовый файл с данным именем все возможные

пути, ведущие от корня к листьям. Перебирать пути, начиная с «самого левого» и заканчивая «самым правым» (при этом первыми заменять конечные элементы пути).

31. Дано дерево глубины *N*, каждая внутренняя вершина которого имеет *K* (< 10) непосредственных потомков (нумеруются от 1 до *K*). Корень дерева имеет номер 0. Записать в текстовый файл с данным именем все пути, ведущие от корня к листьям и удовлетворяющие следующему условию: никакие соседние элементы пути не нумеруются одной и той же цифрой. Дано дерево глубины *N* (*N* — четное), каждая внутренняя вершина которого имеет 2 непосредственных потомка: *A* с весом 1 и *B* с весом –1. Корень дерева *C* имеет вес 0. Записать в текстовый файл с данным именем все пути от корня к листьям, удовлетворяющие следующему условию: суммарный вес элементов пути равен 0. Дано дерево глубины *N* того же типа, что и в задании Recur27. Записать в текстовый файл с данным именем все пути от корня к листьям, удовлетворяющие следующему условию: суммарный вес элементов для любого начального отрезка пути неотрицателен.

32. Дано дерево глубины *N*, каждая внутренняя вершина которого имеет 3 непосредственных потомка: *A* с весом 1, *B* с весом 0 и *C* с весом –1. Корень дерева *D* имеет вес 0. Записать в текстовый файл с данным именем все пути от корня к листьям, удовлетворяющие следующим условиям: суммарный вес элементов для любого начального отрезка пути неположителен, а суммарный вес всех элементов пути равен 0.

33. Дано дерево глубины *N* того же типа, что и в предыдущей задаче. Записать в текстовый файл с данным именем все пути от корня к листьям, удовлетворяющие следующим условиям: никакие соседние элементы пути не обозначаются одной и той же буквой, а суммарный вес всех элементов пути равен 0.

Практическое занятие - 4.

Тема: Структуры основных операторов множеств. Анализ специальных методов представления множеств.

Цель работы: изучение АТД «Множество» и его применения для решения задач.

В результате выполнения практической работы студенты должны:

- *знать* понятие множества и основные операции над множествами, методы реализации множеств при помощи различных структур данных;
- *уметь* применять АТД «Множество» при решении задач.

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

Линейно упорядоченное множество S удовлетворяет следующим двум условиям.

– Для любых элементов a и b из множества S может быть справедливым только одно из следующих утверждений: $a < b$, $a = b$ или $b < a$.

– Для любых элементов a, b и c из множества S таких, что $a < b$ и $b <$ c, следует $a < c$ (свойство транзитивности).

Система обозначений для множеств:

Задание множества путём перечисления элементов: {1, 2, ..., 1000}

Задание множества с использованием шаблона: $\{x \mid$ для произвольного целого y, $x = y^2\}$

Принадлежность элемента x множеству A: $x \in A$

Пустое множество (не содержащее элементов): $\varnothing$

Включение множеств (A является подмножеством B): $A \subseteq B$

Объединением множеств A и B (обозначается $A \cup B$) называется множество, состоящее из элементов, принадлежащих хотя бы одному из множеств A и B.

Пересечением множеств A и B (обозначается $A \cap B$) называется множество, состоящее только из тех элементов, которые принадлежат и множеству A, и множеству B.

Разностью множеств A и B (обозначается $A \setminus B$) называется множество, состоящее только из тех элементов множества A, которые не принадлежат множеству B.

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 C, 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.

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

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

10. 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.

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

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

13. EMPTY(A) – возвращает true, если А – пустое множество, и false в противном случае.

4.1. 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.

4.2. Реализация множеств посредством связанных списков

Элементы списка являются элементами множества. 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.

4.3. Словари

Абстрактный тип множеств с операторами INSERT, DELETE и MEMBER называется DICTIONARY (Словарь).

Словари можно представить следующими способами:

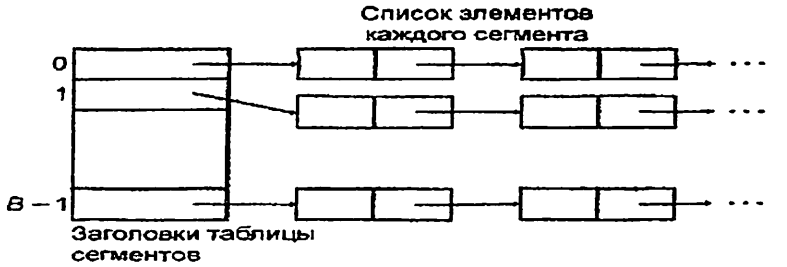
- сортированные или несортированные связанные списки.
- двоичные векторы (предполагая, что элементы данного множества являются целыми числами 1, ..., N для некоторого N или элементы множества можно сопоставить с таким множеством целых чисел).
- массив фиксированной длины с указателем на последнюю заполненную ячейку этого массива.

Существует еще один полезный и широко используемый метод реализации словарей, который называется *хешированием*. Этот метод требует фиксированного времени (в среднем) на выполнение операторов и снимает ограничение, что множества должны быть подмножествами некоторого конечного универсального множества. *Открытое*, или *внешнее, хеширование* позволяет хранить множества в потенциально бесконечном пространстве, снимая тем самым ограничения на размер множеств. *Закрытое*, или *внутреннее, хеширование* использует ограниченное пространство для хранения данных, ограничивая таким образом размер множеств.

4.4. Открытое хеширование

Основная идея заключается в том, что потенциальное множество (возможно, бесконечное) разбивается на конечное число классов. Для B классов, пронумерованных от 0 до B - 1, строится *хеш-функция* h такая, что для любого элемента x исходного множества функция $h(x)$ принимает целочисленное значение из интервала 0, ..., B - 1, которое, естественно, соответствует классу, которому принадлежит элемент x. Элемент x часто называют *ключом*, $h(x)$ — *хеш-значением* x, а "классы" — *сегментами*. Мы будем говорить, что элемент x принадлежит сегменту $h(x)$.

Массив, называемый *таблицей сегментов* и проиндексированный *номерами сегментов* 0, 1,...,B - 1, содержит заголовки для B списков. Элемент x *i-го* списка — это элемент исходного множества, для которого $h(x) = i$.



Организация данных при открытом хешировании

4.5. Закрытое хеширование

При закрытом хешировании в таблице сегментов хранятся непосредственно элементы словаря, а не заголовки списков. Поэтому в каждом сегменте может храниться только один элемент словаря. При закрытом хешировании применяется методика *повторного хеширования*. Если мы попытаемся поместить элемент x в сегмент с номером $h(x)$, который уже занят другим элементом (такая ситуация называется *коллизией*), то в соответствии с методикой повторного хеширования выбирается последовательность других номеров сегментов $h_1(x)$, $h_2(x)$, ..., куда можно поместить элемент x. Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное. Если свободных сегментов нет, то, следовательно, таблица заполнена и элемент x вставить нельзя.

4.6. Реализация АТД для отображений

Отображение (MAPPING) - функция, ставящая в соответствие элементам области определения соответствующие элементы из области значений. Для этого АТД определим 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. Операторы ASSIGN и COMPUTE реализуются точно так же, как операторы INSERT и MEMBER для словарей.

4.7. Очереди с приоритетами

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. Таким образом, оправдывая свое название, DELETEMIN является комбинацией операторов DELETE и MIN, которые были описаны выше.

За исключением хеш-таблиц, те реализации множеств, которые мы рассмотрели ранее, можно применить к очередям с приоритетами.

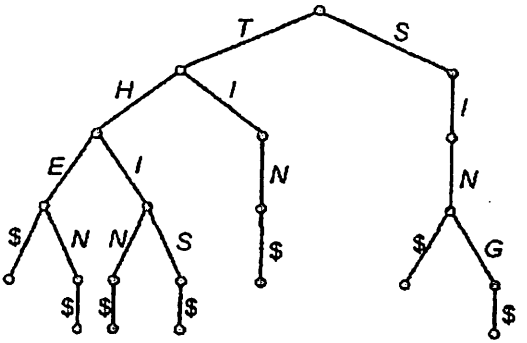
4.8. Структуры мультисписков

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.

Дерево двоичного поиска — это двоичное дерево, узлы которого помечены элементами множеств (мы также будем говорить, что узлы дерева содержат или хранят элементы множества). Определяющее свойство дерева двоичного поиска заключается в том, что все элементы, хранящиеся в узлах левого поддерева любого узла x, меньше элемента, содержащегося в узле x, а все элементы, хранящиеся в узлах правого поддерева узла x, больше элемента, содержащегося в узле x. Это свойство называется *характеристическим свойством дерева двоичного поиска* и выполняется для любого узла дерева двоичного поиска, включая его корень.

4.9. Нагруженные деревья

В этом разделе мы рассмотрим специальную структуру для представления множеств, состоящих из символьных строк. Некоторые из описанных здесь методов могут работать и со строками объектов другого типа, например со строками целых чисел. Эта структура называется *нагруженными деревьями* (tries).



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.

1. Процедура *ASSIGN(node, c, p)*, которая задает значение *p* (указатель на узел) символу *c* в узле *node*.
2. Функция *VALUEOF(node, c)* — возвращает значение (указатель на узел), ассоциированное с символом *c* в узле *node*.
3. Процедура *GETNEW(node, c)* делает значение узла *node* с символом *c* указателем на новый узел.

Нам также будет необходима процедура MAKENULL(node), делающая узел *node* пустым отображением.

Самой простой реализацией узлов нагруженного дерева будет массив *node* указателей на узлы с индексным множеством {A, B, ..., Z, $}. Таким образом, АТД TRIENODE можно определить следующим образом:

```
type
    chars = {'A','B', ...,'Z', '$'};
    TRIENODE = array[chars] of ↑TRIENODE;
```

В принципе, можно применить любую реализацию отображений, но мы хотим найти наиболее подходящую, область определения которых сравнительно мала. При таком условии наилучшим выбором будет реализация отображения посредством связанных списков. Здесь представлением отображения, т.е. узла нагруженного дерева, будет связанный список символов, для которых соответствующие значения не являются указателями **nil**.

Существуют другие подходы к реализации словарей и очередей с приоритетами, где даже в самом худшем случае время выполнения операторов имеет порядок $O(\log n)$. Одна из таких реализаций, которая называется *2-3 дерево*, имеет следующие свойства.

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

Будем считать, что пустое дерево и дерево с одним узлом также являются 2-3 деревьями.

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

4.10. Множества с операторами MERGE и FIND

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.

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.*

Другой подход к реализации АТД MFSET применяет деревья с указателями на родителей. Основная идея здесь заключается в том, что узлы деревьев соответствуют элементам множеств и есть реализация, посредством массивов или какая-либо другая, отображения множества элементов в эти узлы. Каждый узел, за исключением корней деревьев, имеет указатель на своего родителя. Корни содержат как имя компонента-множества, так и элемент этого компонента. Отображение из множества имен к корням деревьев позволяет получить доступ к любому компоненту.

4.11. 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).

Пример: **используя АТД SET, найти простые числа от 2 до 1000 методом решета Эратосфена.**

```
const max=1000;
var
      sieve, primes: SET; {элемент представляет собой целое число}
      i, n1, next: integer;
begin
      for i := 2 to max do
            INSERT(sieve, i);

      next := 2;
      while not EMPTY(sieve) do begin
            n1 := next;
            while n1 <= max do begin
                  DELETE(sieve, n1);
                  n1:=n1+next;
            end;
            INSERT(primes, next);
            repeat
                  next:=next+1;
            until MEMBER(next,sieve) or (next > max);
      end;
      print_set(primes);
end.
```

Задание.

1. Вводятся размерность прямоугольной матрицы и значения ее элементов. Элементы матрицы представляют собой целые положительные числа не более 200. Требуется написать программу, в которой составить множество элементов матрицы, которые меньше всех своих соседей (координаты ±1) и кратны 3. Вводится строка символов, состоящая из строчных латинских букв. Написать программу составления множества всех символов, которые встречаются в строке один раз.

2. Построить алгоритм, выдающий без повторений все перестановки N чисел.

3. Сгенерировать все подмножества данного n-элементного множества {0,..., n-1}.

4. Сгенерировать все k-элементные подмножества множества A из N чисел, A={1, 2, ..., N}. Пример: N=3, k=2, подмножества {1,2}, {1,3}, {2,3}.

5. Из числового множества A мощности n выбрать все подмножества, сумма элементов каждого из которых лежит в пределах от a до b.

6. **Перетекание массы.** На плоскости заданы n материальных точек. С некоторого момента точка с наименьшей массой исчезает, передавая свою массу ближайшей к ней точке. Так продолжается до тех пор, пока не останется одна точка. Реализовать этот процесс и найти оставшуюся точку.

7. Дано 3n точек на плоскости, причем никакие три из них не лежат на одной прямой. Построить множество n треугольников с вершинами в этих точках так, чтобы никакие два треугольника не пересекались и не содержали

друг друга.

8. Задано *n* произвольных натуральных чисел. Найти все группы по *k* чисел, сумма которых равна заданному числу *m*.

9. Задано множество точек. Стянуть это множество к его центру тяжести, уменьшая расстояние между точками в *a*> 1 раз. Центр тяжести должен оставаться неподвижным.

10. Из множества n-мерных векторов, заданных своими целочисленными координатами, найти пары ортогональных либо коллинеарных векторов.

11. Даны два массива числовых множеств: *A(m)* и *B{n)*. Найти все пары множеств A_i и B_i таких, что $card(A_i \bigcap B_i) \le k$ (k задано).

12. *Медианой множества точек на плоскости* назовем прямую, которая делит множество на два подмножества одинаковой мощности. Найти горизонтальную и вертикальную медианы заданного множества, у которого никакие две точки не лежат на одной горизонтальной или вертикальной прямой.

13. Для заданного множества материальных точек найти прямую, которая делит это множество на два подмножества одинаковой мощности и относительно которой эти подмножества находятся в статическом равновесии. (Для равновесия достаточно прохождения прямой через центр тяжести всего множества.)

14. В числовом множестве *A* мощности *n* найти подмножество *B* мощности *k* такое, модуль суммы элементов которого минимален (числа в А имеют разные знаки).

15. Круглая оболочка. Для заданного множества точек на плоскости найти круг минимального радиуса, содержащий все эти точки. **Примечание.** Либо диаметр такого круга совпадает с диаметром множества, либо соответствующая окружность проходит через некоторые 3 точки множества.

16. Треугольная оболочка. Среди заданного множества точек на плоскости выбрать 3 таких, чтобы треугольник, вершинами которого эти точки являются, содержал бы наибольшее количество точек множества.

17. Даны две целочисленных таблицы A [1:10] и B[1:15]. Разработать алгоритм и написать программу, которая проверяет, являются ли эти таблицы похожими. Две таблицы называются похожими, если совпадают множества чисел, встречающихся в этих таблицах.

18. Задается словарь. Найти в нем все анаграммы (слова, составленные из одних и тех же букв).

19. Задано семейство множеств букв. Найти такое k, для которого можно построить множество, состоящее из k букв, причем каждая из них принадлежит ровно k множествам заданного семейства.

20. Построить максимальное множество, состоящее из попарно не сравнимых векторов v. Векторы v определяются парами чисел, выбираемые из данной последовательности чисел a1, ...an , n>=1. Два вектора v=(a,в) и v'=(a',в') называются сравнимыми, если a<=a' и в<=в' или a>=a' и в>= в', в противном случае не сравнимыми.

Практическое занятие - 5.

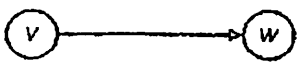
Тема: Представление ориентированных и неориентированных графов.

Цель работы: изучение способов представления ориентированных и неориентированных графов и решение задач при помощи АТД «Граф».

В результате выполнения практической работы студенты должны:

- *знать* основные понятия ориентированных и неориентированных графов и алгоритмов решения типичных задач;
- *уметь* применять АТД «Граф» при решении различных задач.

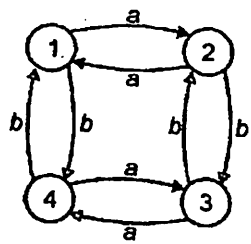
Ориентированный граф (или сокращенно *орграф*) $G = (V, E)$ состоит из множества вершин V и множества дуг E. Вершины также называют *узлами*, а дуги — *ориентированными ребрами*. Дуга представима в виде упорядоченной пары вершин (u, w), где вершина v называется *началом*, а w — *концом* дуги. Дугу (v, w) часто записывают как $v \longrightarrow w$ и изображают в виде



Говорят также, что дуга $v \longrightarrow w$ *ведет от вершины* v *к вершине* w, а вершина w *смежная* с вершиной u.

Путем в орграфе называется последовательность вершин $v_1, v_2, \ldots, v_n$ для которой существуют дуги $v_1 \longrightarrow v_2$, $v_2 \longrightarrow v_3 \ \ldots \ v_{n-1} \longrightarrow v_n$. Этот путь начинается в вершине v_1 и, проходя через вершины $v_2, v_3, \ldots, v_{n-1}$, заканчивается в вершине v_n.

Длина пути — количество дуг, составляющих путь, в данном случае длина пути равна n - 1. Путь называется *простым*, если все вершины на нем, за исключением, может быть, первой и последней, различны. *Цикл* — это простой путь длины не менее 1, который начинается и заканчивается в одной и той же вершине.



Помеченный орграф

Эксцентриситет вершины v определяется как

max(минимальная длина пути от вершины w до вершины v)

Центром орграфа G называется вершина с минимальным эксцентриситетом. Другими словами, центром орграфа является вершина, для которой максимальное расстояние (длина пути) до других вершин минимально.

5.1. Представления ориентированных графов

Матрица смежности для орграфа G — это матрица A размера n x n со значениями булевого типа, где $A[i, j]$ = true тогда и только тогда, когда существует дуга из вершины i в вершину j. Представление орграфа в виде матрицы смежности удобно применять в тех алгоритмах, в которых надо часто проверять существование данной дуги.

	1	2	3	4
1		*a*		*b*
2	*a*		*b*	
3		*b*		*a*
4	*b*		*a*	

Списком смежности для вершины i называется список всех вершин, смежных с вершиной i, причем определенным образом упорядоченный. Таким образом, орграф G можно представить посредством массива *HEAD* (Заголовок), чей элемент *HEAD[i]* является указателем на список смежности вершины i.

5.2. АТД для ориентированных графов

Наиболее общие операторы, выполняемые над ориентированными графами, включают операторы чтения меток вершин и дуг, вставки и удаления вершин и дуг и оператор перемещения по последовательностям дуг.

Для просмотра множества смежных вершин необходимы следующие три оператора.

1. FIRST(v) возвращает индекс первой вершины, смежной с вершиной v. Если вершина v не имеет смежных вершин, то возвращается "нулевая" вершина Λ.
2. NEXT(v, i) возвращает индекс вершины, смежной с вершиной v, следующий за индексом i. Если i — это индекс последней вершины, смежной с вершиной v, то возвращается Λ.
3. VERTEX(v, i) возвращает вершину с индексом i из множества вершин, смежных с v.

5.3. Задача нахождения кратчайшего пути

Пусть есть ориентированный граф $G = (V, E)$, у которого все дуги имеют неотрицательные метки (стоимости дуг), а одна вершина определена как *источник*. Задача состоит в нахождении стоимости кратчайших путей от источника ко всем другим вершинам графа G (здесь *длина пути* определяется как сумма стоимостей дуг, составляющих путь).

Алгоритм Дейкстры (Dijkstra). Алгоритм строит множество S вершин, для которых кратчайшие пути от источника уже известны. На каждом шаге к множеству S добавляется та из оставшихся вершин, расстояние до которой от источника меньше, чем для других оставшихся вершин. Если стоимости всех дуг неотрицательны, то можно быть уверенным, что кратчайший путь от источника к конкретной вершине проходит только через вершины множества S.

Назовем такой путь *особым*. На каждом шаге алгоритма используется также массив D, в который записываются длины кратчайших особых путей для каждой вершины. Когда множество S будет содержать все вершины орграфа, т.е. для всех вершин будут найдены "особые" пути, тогда массив D будет содержать длины кратчайших путей от источника к каждой вершине.

Алгоритм Флойда (R. W. Floyd). Положим, что вершины графа последовательно пронумерованы от 1 до n. Алгоритм Флойда использует матрицу A размера n x n, в которой вычисляются длины кратчайших путей. Вначале $A[i, j] = C[i, j]$ для всех $i \neq j$. Если дуга $i \longrightarrow j$ отсутствует, то $C[i, j] = \infty$. Каждый диагональный элемент матрицы A равен 0.

Над матрицей A выполняется n итераций. После k-й итерации $A[i, j]$ содержит значение наименьшей длины путей из вершины i в вершину j, которые не проходят через вершины с номером, большим k. Другими словами, между концевыми вершинами пути i и j могут находиться только вершины, номера которых меньше или равны k.

На k-й итерации для вычисления матрицы A применяется следующая формула:

$$A_k[i,j] = \min(A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j]).$$

Для вычисления $A_k[i, j]$ проводится сравнение величины $A_{k-1}[i, j]$ (т.е. стоимость пути от вершины i к вершине j без участия вершины k или другой вершины с более высоким номером) с величиной $A_{k-1}[i, k] + A_{k-1}[k, j]$ (стоимость пути от вершины i до вершины k плюс стоимость пути от вершины k до вершины j). Если путь через вершину k дешевле, чем $A_{k-1}[i, j]$, то величина $A_k[i, j]$ изменяется.

Во многих задачах интерес представляет только сам факт существования пути, длиной не меньше единицы, от вершины i до вершины j. Для этого алгоритм разработал *Уоршелл* (S. Warshall).

Предположим, что матрица стоимостей C совпадает с матрицей смежности для данного орграфа G, т.е. $C[i,j] = 1$ только в том случае, если есть дуга $i \longrightarrow j$, и $C[i, j] = 0$, если такой дуги не существует. Мы хотим вычислить матрицу A такую, что $A[i,j]=1$ тогда и только тогда, когда существует путь от вершины i до вершины j длиной не менее 1 и $A[i, j] = 0$ — в противном случае. Такую матрицу A часто называют *транзитивным замыканием* матрицы смежности.

Транзитивное замыкание можно вычислить, применяя на k-м шаге следующую формулу к булевой матрице A:

$$A_k[i, j] = A_{k-1}[i,j] \textbf{ or } (A_{k-1}[i,k] \textbf{ and } A_{k-1}[k,j]).$$

Эта формула устанавливает, что существует путь от вершины i до вершины j, проходящий через вершины с номерами, не превышающими k, только в следующих случаях.

1. Уже существует путь от вершины i до вершины j, который проходит через вершины с номерами, не превышающими k - 1.

2. Существует путь от вершины i до вершины k, проходящий через вершины с номерами, не превышающими k - 1, и путь от вершины k до вершины j, который также проходит через вершины с номерами, не превышающими k - 1.

5.4. 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*.

Неориентированный граф G — (V, E) состоит из конечного множества вершин V и множества ребер E. В отличие от ориентированного графа, здесь каждое ребро (v, w) соответствует *неупорядоченной* паре вершин: если (v, w) — неориентированное ребро, то $(v, w) = (w, v)$. Далее неориентированный граф мы будем называть просто графом.

Путем называется такая последовательность вершин $v_1, v_2, \ldots, v_n$, что для всех i, $1 < i < n$, существуют ребра (v_i, v_{i-1}). Путь называется *простым*, если все вершины пути различны, за исключением, возможно, вершин v_1 и v_n. Длина пути равна количеству ребер, составляющих путь, т.е. длина равна n - 1 для пути из n вершин. Если для вершин v_1 и v_n существует путь $v_1, v_2, \ldots, v_n$, то эти вершины называются *связанными*. Граф называется *связным*, если в нем любая пара вершин связанная.

Пусть есть граф G = (V, E) с множеством вершин V и множеством ребер E. Граф $G'=(V', E')$ называется подграфом графа G, если

1. множество V' является подмножеством множества V,
2. множество E' состоит из ребер (v, w) множества E таких, что обе вершины v и w принадлежат V'.

Если множество E' состоит из *всех* ребер (v, w) множества E таких, что обе вершины v и w принадлежат V', то в этом случае граф G' называется *индуцированным подграфом* графа G.

Связной компонентой графа G называется максимальный связный индуцированный подграф графа G.

Циклом (простым) называется путь (простой) длины не менее 3 от какой-либо вершины до нее самой. Мы не считаем циклами пути длиной 0, длиной 1 (петля от вершины v к ней самой) и длиной 2 (путь вида v, w, v). Граф называется *циклическим*, если имеет хотя бы один цикл. Связный ациклический граф, представляющий собой "дерево без корня", называют *свободным деревом*.

1. Каждое свободное дерево с числом вершин n, $n \geq 1$, имеет в точности n - 1 ребер.

2. Если в свободное дерево добавить новое ребро, то обязательно получится цикл.

Для представления неориентированных графов можно применять те же методы, что и для представления ориентированных графов, если неориентированное ребро между вершинами v и w рассматривать как две ориентированных дуги от вершины v к вершине w и от вершины w к вершине v.

5.5. 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 (с 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: **Задача поиска пути с минимальной стоимостью.**

Пусть имеется n городов, пронумерованных числами от 1 до n.

Для каждой пары городов с номерами i, j в таблице a[i][j] хранится целое число - цена прямого авиабилета из города i в город j. Считается, что рейсы существуют между любыми городами, a[i,i] = 0 при всех i, a[i][j] может отличаться от a[j,i]. Наименьшей стоимостью проезда из i в j считается минимально возможная сумма цен билетов для маршрутов (в том числе с пересадками), ведущих из i в j.

Найти наименьшую стоимость проезда из 1-го города во все остальные.

Обозначим через МинСт(1,s,к) наименьшую стоимость проезда из 1 в s менее чем с k пересадками. Тогда выполняется такое соотношение:

МинСт (1,s,k+1) = наименьшему из чисел МинСт(1,s,k) и

МинСт(1,i,k) + a[i][s] (i=1..n)

Как отмечалось выше, искомым ответом является МинСт(1,i,n) для всех i=1..n.

Алгоритм опишем следующим образом:

```
k:= 1;
 for i := 1 to n do begin x[i] := a[1][i]; end;
 {инвариант: x[i] := МинСт(1,i,k)}
 while k <> n do begin
        for s := 1 to n do begin
           y[s] := x[s];
              for i := 1 to n do begin
                  if y[s] > x[i]+a[i][s] then begin
                     y[s] := x[i]+a[i][s];
                  end;
               end
               {y[s] = МинСт(1,s,k+1)}
               for i := 1 to n do begin x[s] := y[s]; end;
        end;
     k := k + 1;
 end;
```

Приведенный алгоритм называют алгоритмом динамического программирования, или алгоритмом Форда - Беллмана.

Задание.

1. Задан граф в виде количества ребер n<=10 и списка ребер. Необходимо проверить, есть ли в графе вершина, смежная со всеми другими вершинами. Матрица задается при помощи матрицы смежности.

2. Задан набор неповторяющихся пар (Ai,Aj), Ai, Aj принадлежат множеству A={A1, A2, ..., An}. Необходимо составить цепочку максимальной длины по правилу (Ai,Aj)+(Aj,Ak)=(Ai,Aj,Ak). При образовании этой цепочки любая пара может быть использована не более одного раза.

3. Найти все возможные пути между двумя вершинами в графе не пересекающиеся по ребрам

4. Найти все возможные пути между двумя вершинами в графе не пересекающиеся по вершинам.

5. Пометить ребра куба числами от 1 до 12 так, чтобы для всех вершин сумма пометок входящих ребер была одинакова.

6. N колец сцеплены между собой (задана матрица A(n*n), A(i,j)=1 в случае, если кольца i и j сцеплены друг с другом и A(i,j)=0 иначе). Удалить минимальное количество колец так, чтобы получилась цепочка.

7. N различных станков один за другим объединены в конвейер. Имеется N рабочих. Задана матрица C[N , N], где C[i,j] производительность i-ого рабочего на j-ом станке. Определить, на каком станке должен работать каждый из рабочих, чтобы производительность была максимальной.

8. N различных станков расположены параллельно и выполняют однородные операции. Задана матрица C[N , N], где C[i,j] производительность i-ого рабочего на j-ом станке. Определить, на каком станке должен работать каждый из рабочих, чтобы производительность была максимальной.

9. В заданном графе необходимо определить, существует ли цикл, проходящий по каждому ребру графа ровно один раз.

10. Имеется N человек и прямоугольная таблица A[1:N,1:N];элемент A[i,j] равен 1, если человек i знаком с человеком j, A[i,j] = =A[j,i]. Можно ли разбить людей на 2 группы, чтобы в каждой группе были только незнакомые люди.

11. Есть некий граф, состоящий из 1 или более подграфов, каждый из которых связан, в то время как сам граф может быть несвязан. Требуется выявитиь в данном графе все связанные подграфы.

12. Задан граф.Для пары его вершин определить:

а)Кратчайший путь между ними

б)Сколько между ними существует путей менее заданного n

в)Сколько между ними существует простых путей

13. Дан ориентированный граф с N вершинами (N<50). Вершины и дуги окрашены в цвета с номерами от 1 до M (M? 6). Указаны две вершины, в которых находятся фишки игрока и конечная вершина. Правило перемещения фишек: игрок может передвигать фишку по дуге, если ее цвет совпадает с цветом вершины, в которой находится другая фишка; ходы можно делать только в направлении дуг графа; поочередность ходов необязательна. Игра заканчивается если одна из фишек достигает конечной вершины. Написать программу поиска кратчайшего пути до конечной вершины, если он существует.

14. Задан неориентированный граф. При прохождении по некоторым ребрам некоторые (определенные заранее) ребра могут исчезать или появляться. Найти кратчайший путь из вершины с номером q в вершину с номером w.

15. Заданы два числа N и M (20<=M<=N<=150), где N - количество точек на плоскости. Требуется построить дерево из M точек так, чтобы оно было оптимальным. Дерево называется оптимальным, если сумма всех его ребер минимальна. Все ребра - это расстояния между вершинами, заданными координатами точек на плоскости.

16.Даны два числа N и M. Построить граф из N вершин и M ребер. Каждой вершине ставится в соответствие число ребер, входящих в нее. Граф должен быть таким, чтобы сумма квадратов этих чисел была минимальна.

17.Задан ориентированный граф с N вершинами, каждому ребру которого приписан неотрицательный вес. Требуется найти простой цикл, для которого среднее геометрическое весов его ребер было бы минимально. **Примечание.** Цикл называется простым, если через каждую вершину он проходит не более одного раза, петли в графе отсутствуют.

18.Внутрь квадрата с координатами левого нижнего угла (0,0) и координатами верхнего угла (100, 100) поместили N (1<=N<=30) квадратиков. Необходимо найти кратчайший путь из точки (0,0) в точку (100,100), который бы не пересекал ни одного из этих квадратиков.

Ограничения:

- длина стороны каждого квадратика равна 5;
- стороны квадратиков параллельны осям координат;
- координаты всех углов квадратиков - целочисленные;
- квадратики не имеют общих точек.

19.Задан неориентированный граф с N вершинами, пронумерованными целыми числами от 1 до N. Написать программу, которая последовательно решает следующие задачи:

- выясняет количество компонент связности графа;
- находит и выдает все такие ребра, что удаление любого из них ведет к увеличению числа компонент связности;
- определяет, можно ли ориентировать все ребра графа таким образом, чтобы получившийся граф оказался сильно связным;
- ориентирует максимальное количество ребер, чтобы получившийся граф оказался сильно связным;
- определяет минимальное количество ребер, которые следует добавить в граф, чтобы ответ на третий пункт был утвердительным.

20.Задан ориентированный граф с N (1<=N<=33) вершинами, пронумерованными целыми числами от 1 до N. Напишите программу, которая подсчитывает количество различных путей между всеми парами вершин графа.

21.Ребенок нарисовал кружки и некоторые из них соединил отрезками. Кружки он пометил целыми числами от 1 до N (1<=N<=30), а на каждом отрезке поставил стрелочку. Затем он приписал каждому кружочку вес в виде некоторого целого числа и определил начальный и конечный кружочки. Из первого он должен выйти, а во второй попасть.

Ребенок решил для себя следующее:

- набрать максимально возможное суммарное количество очков;
- по каждому отрезку пройти ровно один раз;
- если в кружок он попадает при движении по направлению стрелки, то к суммарному количеству очков вес этого кружка прибавляется;
- если в кружок он попадает при движении против направления стрелки, то из суммарного количества очков вес этого кружка вычитается.

Написать программу, которая бы помогла ребенку построить путь, удовлетворяющий всем этим требованиям.

22.На площади n точек заданы своими координатами. Составить программу нахождения "дерева", которое соединяет все точки и имеет минимальную суммарную длину веток.

23.Почтальону, находящемуся в городе K, нужно забрать почту из N городов. Расстояния между городами заданы в матрице A[1..N,1..N], где A[x,y] - расстояние между городами x и y. Составить программу для нахождения оптимального маршрута почтальона.

24.Имеется N городов. Для каждой пары городов (I,J) можно построить дорогу, соединяющую эти два города и не заходящие в другие города. Стоимость такой дороги A(I,J). Вне городов дороги не пересекаются. Написать алгоритм для нахождения самой дешевой системы дорог, позволяющей попасть из любого города в любой другой. Результаты задавать таблицей B[1:N,1:N], где B[I,J]=1 тогда и только тогда, когда дорогу, соединяющую города I и J, следует строить.

25.Даны декартовы координаты N перекрестков города, которые пронумерованы от 1 до N. На каждом перекрестке имеется светофор. Некоторые из перекрестков соединены дорогами с двусторонним (правосторонним) движением, которые пересекаются только на перекрестках. Для каждой дороги известно время, которое требуется для проезда по ней от одного перекрестка до другого.

26.Необходимо проехать от перекрестка с номером A до перекрестка с номером B за минимальное время. Время проезда зависит от набора проезжаемых дорог и от времени ожидания на перекрестках. Так, если вы подъехали от перекрестка X к перекрестку C по дороге X—>C и хотите ехать дальше по дороге C—>Y, то время ожидания на перекрестке C зависит от того, поворачиваете ли вы налево или нет. Если вы поворачиваете налево, то время ожидания равно D * K, где D равно количеству дорог, пересекающихся на перекрестке C,а K — некоторая константа. Если вы не поворачиваете налево, то время ожидания равно нулю. Написать программу, которая определяет самый быстрый маршрут.

27.Есть N карточек. На каждой из них черными чернилами написан ее уникальный номер — число от 1 до N. Также на каждой карточке красными чернилами написано еще одно целое число, лежащее в промежутке от 1 до N (некоторыми одинаковыми «красными» числами могут помечаться несколько карточек). Необходимо выбрать из данных N карточек максимальное число карточек таким образом, чтобы множества «красных» и «черных» чисел на них совпадали.

28.(Задача о коммивояжере) Дан полный ориентированный симметрический граф с вершинами x1, x2,...xn.Вес дуги xixj задан элементами Vij матрицы весов. Используя алгоритм метода ветвей и границ, найти Гамильтонов контур минимального (максимального) веса.

29.*Треугольником* в графе называется всякая тройка различных и попарно смежных вершин этого графа. *Склеиванием* треугольника называется замена

треугольника одной вершиной с сохранением связности его с остальным графом. Последовательно применяя склеивание, преобразовать данный граф в граф, в котором нет треугольников.

30.Известен возраст каждого из n женихов и каждой из m невест. Сформировать из них возможно большее число пар так, чтобы разница в возрасте между молодоженами в каждой паре была не больше заданного k.

Практическое занятие - 6.

Тема: Анализ алгоритмов внутренней и внешней сортировки.

Цель работы: изучение методов сортировки и их применения для решения задач.

В результате выполнения практической работы студенты должны:

- *знать* различные методы внутренней и внешней сортировки;
- *уметь* выбирать подходящие алгоритмы сортировки и применять их при решении задач.

Сортировкой, или упорядочиванием списка объектов, называется расположение этих объектов по возрастанию или убыванию согласно определенному линейному отношению порядка, такому как отношение "≤" для чисел. При внутренней сортировке все сортируемые данные помещаются в оперативную память компьютера, где можно получить доступ к данным в любом порядке (т.е. используется модель памяти с произвольным доступом). Внешняя сортировка применяется тогда, когда объем упорядочиваемых данных слишком большой, чтобы все данные можно было поместить в оперативную память.

6.1. Метод "пузырька".

Чтобы описать основную идею этого метода, представим, что записи, подлежащие сортировке, хранятся в массиве, расположенном вертикально. Записи с малыми значениями ключевого поля более "легкие" и "всплывают" вверх наподобие пузырька. При первом проходе вдоль массива, начиная проход снизу, берется первая запись массива и ее ключ поочередно сравнивается с ключами последующих записей. Если встречается запись с более "тяжелым" ключом, то эти записи меняются местами.

Алгоритм „пузырька"

```
for i:= 1 to n- 1 do
    for j:=1 downto i+1 do
        if A[j].key<A[j-1].key then
            swap(A[j],A[j-1])
```

6.2. Сортировка вставками

Второй метод, который мы рассмотрим, называется сортировкой вставками, так как на i-м этапе мы "вставляем" i-й элемент $A[i]$ в нужную позицию среди элементов A[1], A[2], ..., $A[i-1]$, которые уже упорядочены. После этой вставки первые i элементов будут упорядочены. Сказанное можно записать в виде следующей псевдопрограммы:

```
for i:=2 to n do
```

переместить *A[i]* на позицию *j* <*L i* такую, что *A[i]* < *A[к]* для *j* < *к* < i и либо *A [i]* > *A[j*-1], либо *j* = 1

Сортировка посредством выбора. Идея сортировки посредством выбора также элементарна, как и те два метода сортировки, которые мы уже рассмотрели. На *i*-м этапе сортировки выбирается запись с наименьшим ключом среди записей A[*i*], ..., *A[n]* и меняется местами с записью *A[i]*. В результате после *i*-го этапа все записи A[1], ..., *A[i]* будут упорядочены. Сортировку посредством выбора можно описать следующим образом:

for i:= 1 to *n* -1 **do**
выбрать среди *A [i]*, ...,A[*n*]элемент с наименьшим ключом
и поменять его местами с *A [i]*;

Быстрая сортировка. Алгоритм с временем выполнения *O(n* log*n)* является, по-видимому, самым эффективным методом внутренней сортировки и поэтому имеет название "быстрая сортировка". В этом алгоритме для сортировки элементов массива *A[1], ..., A[n]* из этих элементов выбирается некоторое значение ключа *v* в качестве *опорного элемента,* относительно которого переупорядочиваются элементы массива. Желательно выбрать опорный элемент близким к значению медианы распределения значений ключей так, чтобы опорный элемент разбивал множество значений ключей на две примерно равные части. Далее элементы массива переставляются так, чтофы для некоторого индекса *j* все переставленные элементы *A*[1], ..., *A[j]* имели значения ключей, меньшие чем и, а все элементы *A[j*+1], ..., *A*[*n*] — значения ключей, большие или равные и. Затем процедура быстрой сортировки рекурсивно применяется к множествам элементов *A*[1], ..., *A[j]* и *A[j*+1], ..., *A[n]* для упорядочивания этих множеств по отдельности. Поскольку все значения ключей в первом множестве меньше, чем значения ключей во втором множестве, то исходный массив будет отсортирован правильно.

Процедура быстрой сортировки

if A[i],...,A[j] имеют не менее двух различных ключей
then begin
пусть v — наибольший из первых двух найденных различных ключей;
переставляются элементы A[i],..., A[j] так, чтобы
для некоторого k, i+l<k<j, A[i],...,A[k-1] имели ключи,
меньшие, чем *v*, а A[k],...,A[j] — большие или равные v;
quicksort(i, k-1);
quicksort(k, j)
end

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
```

6.3. „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.

Общая поразрядная сортировка. Предположим, что тип данных ключей keytype является записями со следующими полями:

```
type
```

```
keytype =
        record
        day:
        1..31;
        month:  (jan,  ...,  dec);
        year:  1900..1999
    end;
```

либо массивом элементов какого-нибудь типа, например

```
type
  keytype  = array[1..10]  of char;
```

Далее будем предполагать, что данные типа keytype состоят из k компонент f_1 , f_2 ..., f_k типа t_1, t_2, ..., t_k. Например, в (8.11) t_1 — 1..31, t_2 = (jan, ..., dec), а t_3 = 1900..1999. В (8.12) k = 10, а $t_1 = t_2 = ... = t_k$ = char.

Предположим также, что мы хотим сортировать записи в *лексикографическом порядке* их ключей. В соответствии с этим порядком ключевое значение $(a_1, a_2, ..., a_k)$ меньше ключевого значения $(b_1, b_2, ..., b_k)$, где a_i и b_i— значения из поля f_i $(i = 1, 2, ..., k)$, если выполняется одно из следующих условий:

1. $a_1 < b_1$ или
2. $a_1 = b_1$ и $a_2 < b_2$, или

.....

k. $a_1 = b_1$, $a_2 = b_2$, ..., $a_{k-1} = b_{k-1}$ и $a_k < b_k$.

Другими словами, ключевое значение $(a_1, a_2, ..., a_k)$ меньше ключевого значения $(b_1, b_2, ..., b_k)$, если существует такой номер j $(1 < j < k - 1)$, что $a_1 = b_1, a_2 = b_2, ..., a_j = b_j$ и $a_{j+1} < b_{j+1}$.

Если принять сделанные выше определения ключей, то в этом случае значения ключей можно рассматривать как целочисленные выражения в некоторой системе счисления. Например, определение (8.12), где каждое поле ключа содержит какой-либо символ, позволяет считать эти символы целочисленными выражениями по основанию 128 (если использовать только латинские буквы) или, в зависимости от используемого набора символов, по какому-то другому основанию. В определении типов (8.11) значения поля *year* (год) можно считать целыми числами по основанию 100 (так как здесь значения поля изменяются от 1900 до 1999, т.е. могут изменяться только последние две цифры), значения поля *month* (месяц) очевидно можно интерпретировать как целочисленные выражения по основанию 12, а значения третьего поля *day* (день) — как целочисленные выражения по основанию 31. Предельный случай такого подхода: любые целые числа (из конечного фиксированного множества) рассматриваются как массивы цифр по основанию 2 или другому подходящему основанию. Обобщение "карманной" сортировки, использующее такое видение значений ключевых полей, называется *поразрядной сортировкой*.

Основная идея поразрядной сортировки заключается в следующем: сначала проводится "карманная" сортировка всех записей по полю f_k (как бы по "наименьшей значащей цифре"), затем отсортированные по "карманам" записи объединяются (наименьшие значения идут первыми), затем к полученному объединенному списку применяется "карманная" сортировка по полю f_{k-1} снова проводится конкатенация содержимого "карманов", к объединенному списку применяется "карманная" сортировка по полю f_{k-2}, и т.д. При сортировке записей по "карманам" новая запись, вставляемая в карман, присоединяется в конец списка элементов этого "кармана", а не в начало, как было в "чистой карманной" сортировке. После выполнения "карманной" сортировки по всем ключам f_k, f_{k-1} ,..., f_1 и последнего объединения записей получим их результирующий список, где все записи будут упорядочены в лексикографическом порядке в соответствии с этими ключами. Эскиз описанного алгоритма в виде процедуры *radixsort* (поразрядная сортировка) приведен в листинге

```
    procedure radixsort;
        { radixsort сортирует список A из л записей по ключевым полям
          f1 , f2 ...,fk типов t1, t2, ..., tk соответственно. В качестве
          "карманов" используются массивы Bi типа array[ti] of listtype,
          1 < i < k, где listtype — тип данных связанных списков записей }
     begin
(1)      for i:= k downto 1 do begin
(2)          for для каждого значения v типа ti do
                 { очистка "карманов" }
(3)              сделать Bi[v] пустым;
(4)          for для каждой записи r из списка A do
(5)              переместить запись r в конец списка "кармана"
                     Bi[v],
                 где v — значение ключевого поля fi записи r;
(6)          for для каждого значения v типа ti
                     в порядке возрастания v do
(7)      конкатенация Bi[v] в конец списка A
         end
     end; { radixsort }
```

Пример: **Реализация сортировки Шелла.**

```
procedure Shell(var item: DataArray; count:integer);
 const
  t = 5;
 var
  i, j, k, s, m: integer;
  h: array[1..t] of integer;
  x: DataItem;
 begin
```

```
h[1]:=9; h[2]:=5; h[3]:=3; h[4]:=2; h[5]:=1;
for m := 1 to t do
 begin

k:=h[m];
s:=-k;
for i := k+1 to count do
begin
 x := item[i];
 j := i-k;
 if s=0 then
 begin
  s := -k;
  s := s+1;
  item[s] := x;
 end;
 while (x<item[j]) and (j<count) do
  begin
   item[j+k] := item[j];
   j := j-k;
  end;
  item[j+k] := x;
 end;
end;
end; { конец сортировки Шелла }
```

Задание.

1. 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). переставить команды (то есть соответствующие строки и столбцы таблицы) в соответствии с занятым местом, указанным на диагонали.

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

3. 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*.

4. 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*.

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

6. 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*.

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

8. 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.

9. 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*.

10. 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.

11. 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*.

12. **Горе-летописец.** Первое слово каждой строки текстового файла (первое поле каждой записи в массиве записей) содержит дату события в формате Д/М/ГГГГ, где месяц представлен римскими цифрами, например 13/IV/2001; остальные слова или поля — описание события. Упорядочить события хронологически (по возрастанию дат).

13. Требуется выполнить сортировку временных моментов, заданных в часах, минутах и секундах.

14. В массиве (файле) хранятся данные о членах некоторого коллектива: фамилия, имя (как мужского, так и женского пола), телефон. Среди представленных персон немало однофамильцев. Упорядочить список по невозрастанию количества однофамильцев.

15. Имеется *n* деталей, каждая из которых проходит обработку сначала на одном станке, затем на другом (например, токарный и шлифовальный). На каждом станке одновременно обрабатывается только одна деталь; время на

переналадку не требуется. Известно время обработки каждой детали на каждом станке. Упорядочить детали так, чтобы суммарное время обработки партии деталей было минимально.

16. Несколько арифметических прогрессий заданы своими параметрами: $a^{(i)}$— первый член; $d^{(i)}$— разность; $n^{(i)}$ — количество членов; $d^{(i)} > 0$, $i = 1,..., m$. Не находя самих прогрессий (в виде массивов или файлов), произвести их слияние в один неубывающий массив.

17. Даны N натуральных чисел. Найти минимальное натуральное число, не представимое суммой никаких этих чисел, если в эту сумму каждое исходное число может входить не более одного раза. Ограничения: $1 \le N \le 10000$, значения исходных чисел от 1 до 1000000000, время 1 с.

18. Вывести в порядке возрастания все несократимые дроби, заключённые между 0 и 1, знаменатели которых не превышают N. Ограничения: $2 \le N \le 255$, время 1 с.

19. На числовой прямой окрасили N отрезков. Известны координаты левого и правого концов каждого отрезка (L_i и R_i). Найти длину окрашенной части числовой прямой. Ограничения: $-1000000000 \le L_i \le R_i \le 1000000000$, $1 \le N \le 15000$, время 1 с.

20. На окружности окрасили N дуг. Известны угловая координата L[I] начала и R[I] конца I-ой дуги (от начала к концу двигались, закрашивая дугу, против часовой стрелки). Какая доля окружности окрашена?

21. Имеется N камней веса A1,A2,...,AN. Необходимо разбить их на две кучи таким образом, чтобы веса куч отличались не более чем в 2 раза. Если этого сделать нельзя, то указать это.

22. Даны две целочисленных таблицы A [1:10] и B[1:15]. Разработать алгоритм и написать программу, которая проверяет, являются ли эти таблицы похожими. Две таблицы называются похожими, если совпадают множества чисел, встречающихся в этих таблицах.

23. Задаются число n>1 - размерность пространства и размеры M n-мерных параллелепипедов (ai1 , ..., ain), i=1,...,M. Параллелепипед может располагаться в пространстве любым из способов, при которых его ребра параллельны осям координат. Найти максимальную последовательность вкладываемых друг в друге параллелепипедов.

24. Ожерелье представляет собой нитку, на которую надето N различных бусинок. У нитки два конца: левый и правый. С бусинками на нитке можно делать следующие операции:

- ML (move left) - снять крайнюю левую бусинку и надеть ее на правый конец нитки. По сути при этом происходит циклический сдвиг ожерелья.
- MR (move right) - снять правую бусинку и надеть ее на левый конец.
- RL (remove left) - снять крайнюю левую бусинку и положить ее на стол (не одевать на нитку).
- RR (remove right) - снять крайнюю правую бусинку и положить ее на стол.
- PL (put left) - взять со стола бусинку и надеть ее на левый конец нитки.

• PR (put right) - взять со стола бусинку и надеть ее на правый конец нитки.

На столе может лежать не более одной снятой бусинки, то есть после операции RL или RR нельзя использовать другую операцию RL или RR, пока не будет выполнена операция PL или PR.

Ваша задача используя только перечисленные операции отсортировать исходное ожерелье

25. Упорядочить строки матрицы *K(m,n)*, содержащей натуральные числа, по возрастанию суммы цифр в десятичной системе счисления, используемых для записи элементов строки.

26. Массив чисел, заданных в шестнадцатеричной системе счисления в текстовом файле (по одному числу в строке, цифры разделены пробелами), упорядочить по неубыванию, не переводя числа в другую систему.

27. В результате m экспериментов разной продолжительности получены наборы значений функций $y^{(k)} = y^{(k)}(x)$, $k = 1, 2, ..., m$ 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. 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): $y = y_{i-1} + \frac{x - x_{i-1}}{h}(y_i - y_{i-1})$.

29. К подготовленной статье автор приложил список использованной литературы, но расположил издания в порядке появления ссылок на них в тексте. Редактор потребовал расположить источники по алфавиту. Упорядочить список литературы по требованию редактора (каждое название — с нового абзаца: номер п/п, автор(-ы), название работы и т. д. Текст статьи со ссылками прилагается; после сортировки внести соответствующие исправления в текст. Для справки: каждая ссылка — это (в квадратных скобках) номер из списка или несколько номеров. Проверить корректность ссылок (наличие в списке источника с соответствующим номером).

30. Пополнить упорядоченный по возрастанию список A элементами неупорядоченного списка B, сохранив упорядоченность (совпадающие элементы включать единожды).

Практическое занятие - 7.

Тема: Изучение методов анализа алгоритмов. Сравнительный анализ методов разработки алгоритмов.

Цель работы: изучение методов анализа временной эффективности алгоритмов и основных способов разработки алгоритмов.

В результате выполнения практической работы студенты должны:

- *знать* методы анализа алгоритмов; методы декомпозиции, динамического программирования, поиска с возвратом, «жадные» алгоритмы;
- *уметь* оценивать временную эффективность алгоритмов и выбирать эффективные методы решения задач.

7.1. Анализ временной эффективности алгоритмов.

Одной из объективных оценок алгоритма может служить временная эффективность алгоритма. Чтобы повысить объективность оценок алгоритмов, ученые приняли асимптотическую временную сложность как основную меру эффективности выполнения алгоритма.

Говорят, что алгоритм имеет эффективность (т.е. временную сложность в самом худшем случае) $O(f(n))$, или просто $f(n)$, если функция от n, равная максимуму числу шагов, выполняемых алгоритмом, имеет порядок роста $O(f(n))$, причем максимум берется по всем входным данным длины n. Можно сказать по-другому: существует константа c, такая, что для достаточно больших n величина $cf(n)$ является верхней границей количества шагов, выполняемых алгоритмом для любых входных данных длины n.

Существуют три различных подхода к решению рекуррентных соотношений.

1. 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 $T(n) \leq f(n)$.
2. 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 $T(n)$.
3. Третий подход заключается в использовании общих решений определенных рекуррентных соотношений, приведенных в этой главе или взятых из других источников.

Рассмотрим решение рекуррентных соотношений, когда исходную задачу размера n можно разделить на a подзадач каждую размера n/b. Для

определенности будем считать, что выполнение задачи размера 1 требует одну единицу времени и что время "сборки" подзадач, составляющих исходную задачу размера n, требует $d(n)$ единиц времени. Например, пусть $a = b = 2$ и $d(n) = c_2n/c_1$ в единицах c_1. Тогда, если обозначить через $T(n)$ время выполнения задачи размера n, будем иметь

$$T(1) = 1,$$
$$T(n) = aT(n/b) + d(n). \qquad (7.1)$$

Подставим вместо n выражение n/b^i:

$$T\left(\frac{n}{b^i}\right) = aT\left(\frac{n}{b^{i+1}}\right) + d\left(\frac{n}{b^i}\right). \qquad (7.2)$$

Подставляя в (7.1) равенства (7.2) последовательно для i = 1, 2, ..., получим

$$T(n) = aT\left(\frac{n}{b}\right) + d(n) = a\left(aT\left(\frac{n}{b^2}\right) + d\left(\frac{n}{b}\right)\right) + d(n) = a^2T\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) =$$
$$= a^2\left(aT\left(\frac{n}{b^3}\right) + d\left(\frac{n}{b^2}\right)\right) + ad\left(\frac{n}{b}\right) + d(n) = a^3T\left(\frac{n}{b^3}\right) + a^2d\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) =$$
$$= \ldots = a^iT\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^jd\left(\frac{n}{b^j}\right).$$

Если, как мы предположили, $n = b^k$, тогда мы получаем формулу

$$T(n) = a^k + \sum_{j=0}^{k} a^jd(b^{k-j}). \qquad (7.3)$$

Так как $k = log_bn$, то первое выражение в (7.3) можно записать как a^{logbn} или, что эквивалентно, как n^{logba}. Таким образом, это выражение обозначает n в степени, которая зависит от a и b. В общем случае, чем больше a (т.е. надо решить большее количество подзадач), тем больший показатель степени n. При больших значениях b (чем больше b, тем меньше размер подзадач) показатель степени n будет меньше.

Первое выражение a^k или n^{logba} называется *однородным решением* (по аналогии с дифференциальными уравнениями). Однородное решение — это точное решение уравнения (7.1), когда функция $d(n)$, называемая *управляющей функцией*, равна 0 для всех n. Другими словами, однородное решение соответствует стоимости выполнения всех подзадач, если их можно объединить "бесплатно".

С другой стороны, второе выражение формулы (7.3) соответствует стоимости создания подзадач и комбинирования их результатов. Назовем это выражение *частным решением* (также по аналогии с теорией дифференциальных уравнений). Частное решение зависит как от управляющей функции, так и от количества и размера подзадач. Существует практическое правило: если однородное решение больше управляющей функции, то частное решение имеет тот же порядок роста, что и однородное решение. Если же управляющая функция растет на порядок n^ε (для некоторого $\varepsilon > 0$) быстрее

однородного решения, тогда частное решение имеет такой же порядок роста, как и управляющая функция. Когда управляющая функция имеет с однородным решением одинаковый порядок роста или растет не медленнее, чем $log^k n$ для некоторого k, тогда частное решение растет как управляющая функция, умноженная на $\log n$.

Будем говорить, что функция f целочисленного аргумента называется *мультипликативной,* если для всех положительных целых чисел x и y справедливо равенство $f(xy) = f(x)f(y)$. В этом случае можно найти точное решение.

7.2. Алгоритмы „разделяй и властвуй"

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.

7.3. Динамическое программирование.

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 про исходит из теории управления). Динамическим программированием (в наиболее общей форме) называют процесс пошагового решения задач, когда на каждом шаге выбирается одно решение из множества допустимых (на этом шаге) решений, причем такое, которое оптимизирует заданную целевую функцию или функцию критерия. (В приведенных далее примерах процесс пошагового решения задачи сводится к пошаговому заполнению таблиц.) В основе теории динамического программирования лежит *принцип оптимальности Беллмана.*

7.4. „Жадные" алгоритмы

На каждой отдельной стадии "жадный" алгоритм выбирает тот вариант, который является *локально оптимальным* в том или ином смысле. Следует подчеркнуть, что не каждый "жадный" алгоритм позволяет получить оптимальный результат в целом. Как нередко бывает в жизни, "жадная стратегия" подчас обеспечивает лишь сиюминутную выгоду, в то время как в целом результат может оказаться неблагоприятным. Существуют задачи, для которых ни один из известных "жадных" алгоритмов не позволяет получить оптимального решения; тем не менее имеются "жадные" алгоритмы, которые с большой вероятностью позволяют получать "хорошие" решения. Нередко

вполне удовлетворительным можно считать "почти оптимальное" решение характеризующееся стоимостью, которая лишь на несколько процентов превышает оптимальную. В таких случаях "жадный" алгоритм зачастую оказывается самым быстрым способом получить "хорошее" решение. Вообще говоря, если рассматриваемая задача такова, что единственным способом получить оптимальное решение является использование метода полного поиска, тогда "жадный" алгоритм или другой эвристический метод получения хорошего (хотя и необязательно оптимального) решения может оказаться единственным реальным средством достижения результата.

7.5. 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.

Алгоритмы локального поиска

Описанная ниже стратегия нередко приводит к оптимальному решению задачи.

1. Начните с произвольного решения.
2. Для улучшения текущего решения примените к нему какое-либо преобразование из некоторой заданной совокупности преобразований. Это улучшенное решение становится новым "текущим" решением.
3. 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.

Пример: **применение метода «разделяй и властвуй» для вычисления определителя матрицы.**

```
Const
  max_n = 4;
Type
  matrix = Array[1 .. max_n, 1 .. max_n] Of real;
{ Матрица, для которой будет вычисляться определитель }
Const
```

```
 a: matrix =  ((2, 9, 9, 4),  (2, -3, 12, 8), (4, 8, 3, -5),   (1, 2, 6, 4));

function minusOne(n: integer): integer;
 begin
  minusOne := (1 - 2*Byte(Odd(n)));
 end;
function get_addr(i, j: integer;
      const n: integer): integer;
 begin
  get_addr := pred(i) * n + j
 end;
{ Рекурсивное определение определителя }
Function det(Var p; Const n: integer): real;
 Type
  matrix = Array[1 .. max_n * max_n] Of real;
 Var
  my_p: matrix Absolute p;
  pp: ^matrix;
  s: real;
  i, j, curr: integer;
 Begin
  s := 0.0;
  If n = 2 Then
   Begin
    det := my_p[1]*my_p[4] - my_p[2]*my_p[3]; exit
   End;
  For i := 1 To n Do
   Begin
    GetMem(pp, Sqr(Pred(n)) * SizeOf(real));
    curr := 1;
    For j := 1 To n Do
     If j <> i Then
      Begin
       move(my_p[get_addr(j,2,n)],pp^[get_addr(curr,1,Pred(n))],
       pred(n) * SizeOf(real));
       inc(curr);
      End;
    s := s + minusOne(Succ(i)) * my_p[get_addr(i, 1, n)] *
          det(pp^, Pred(n));
    FreeMem(pp, Sqr(Pred(n)) * SizeOf(real))
   End;
  det := s
 End;
begin
 WriteLn( det(a, 4):0:0 );
```

end.

Задание.

1. Вывести значение целочисленного выражения, заданного в виде строки *S*. Выражение определяется следующим образом:

```
<выражение>   ::=   <терм> | <выражение> + <терм> |
                    <выражение> – <терм>
<терм>        ::=   <цифра> | <терм> * <цифра>
```

2. Вывести значение целочисленного выражения, заданного в виде строки *S*. Выражение определяется следующим образом:

```
<выражение>   ::=   <терм> | <выражение> + <терм> |
                    <выражение> – <терм>
<терм>        ::=   <элемент> | <терм> * <элемент>
<элемент>     ::=   <цифра> | (<выражение>)
```

3. Вывести значение целочисленного выражения, заданного в виде строки *S*. Выражение определяется следующим образом:

```
<выражение>   ::=   <цифра> |
                    (<выражение><знак><выражение>)
<знак>        ::=   + | – | *
```

4. Вывести значение целочисленного выражения, заданного в виде строки *S*. Выражение определяется следующим образом (функция M возвращает максимальный из своих параметров, а функция m — минимальный):

```
<выражение>   ::=   <цифра> | M(<выражение> , <выражение>) |
                    m(<выражение> , <выражение>)
```

5. Вывести значение логического выражения, заданного в виде строки *S*. Выражение определяется следующим образом («T» — True, «F» — False):

```
<выражение>   ::=   T | F | And(<выражение> , <выражение>) |
                    Or(<выражение> , <выражение>)
```

6. Вывести значение логического выражения, заданного в виде строки *S*. Выражение определяется следующим образом («T» — True, «F» — False):

7.

```
<выражение>   ::=   T | F | And(<параметры>) |
                    Or(<параметры>) | Not(<выражение>)
<параметры>   ::=   <выражение> | <выражение> , <параметры>
```

7. Ввести в символьной форме два многочлена от x с целыми коэффициентами и вывести их произведение в порядке убывания степеней (также в символьной форме).

8. В заданной последовательности целых чисел найти максимально длинную подпоследовательность чисел такую, что каждый последующий элемент подпоследовательности делился нацело на предыдущий.

9. Пусть x и y - две бинарных последовательности (т.е. элементы последовательностей - нули и единицы); x и y можно рассматривать как запись в двоичной форме некоторых двух натуральных чисел. Найти максимальное число z, двоичную запись которого можно получить вычеркиванием цифр как из x, так и из y. Ответ выдать в виде бинарной последовательности.

10. Задана матрица натуральных чисел A(n,m). За каждый проход через клетку (i,j) взымается штраф A(i,j). Необходимо минимизировать штраф и а) Пройти из какой-либо клетки 1-ой строки в n-ую строчку, при этом из текущей клетки можно перейти 1) в любую из 3-х соседних, стоящих в строке с номером на 1-цу большем; 2) в любую из 8 соседних клеток; б) Реализовать пункт а) для перехода из клетки (1,1) в (n,m).

11. Покупатель имеет купюры достоинством A(1), ...,A(n), а продавец - B(1), .. ,B(m). Необходимо найти максимальную стоимость товара P, которую покупатель не может купить, потому что нет возможности точно рассчитаться за этот товар с продавцом, хотя денег на покупку этого товара достаточно.

12. У покупателя есть n монет достоинством H(1),..., H(n). У продавца есть m монет достоинством B(1),...,B(l). Может ли купить покупатель вещь стоимости S так, чтобы у продавца нашлась точная сдача (если она необходима).

13. Задан массив M [1:N] натуральных чисел, упорядоченный по неубыванию, т.е.: M[1]<=M[2]<=...<=M[N]. Написать алгоритм выплаты заданной суммы S минимальным количеством купюр достоинством M(1), ..., M(N).

14. Фермер хочет построить на своей земле как можно больший по площади сарай. Но на его участке есть деревья и хозяйственные постройки, которые он не хочет никуда переносить. Для простоты представим ферму сеткой размера MxN. Каждое из деревьев и построек размещается в одном или нескольких узлах сетки. Прямоугольный сарай не должен ни с чем соприкасаться (т.е. в соседних с ним узлах сетки не может ничего быть). Найти максимально возможную площадь сарая и где он может размещаться.

15. Пусть известно, что для перемножения матрицы размера n*m на матрицу размера m*k требуется n*m*k операций. Необходимо определить, какое

минимальное число операций потребуется для перемножения n матриц A1,...An, заданных своими размерами n(i)*m(i). При этом можно перемножать любые две рядом стоящие матрицы, в результате чего получается матрица нужного размера. Замечание: n(i) - число строк в матрице Ai m(i) - число столбцов в матрице Ai n(i)=m(i)+1.

16. Дан набор переменных x1,x2,...,xN. Каждая переменная xi может принимать значение только -1, 0 или 1. Для данного целого числа S требуется определить количество способов присвоить переменным xi значения так, чтобы сумма всех возможных произведений xixj была равна S, где i<j и i,j=1,2,...,N. Два способа считаются различными, если они содержат различное число xi=0.

17. Определим правильные скобочные выражения так:

- Пустое выражение – правильное.
- Если выражение S правильное, то (S) и [S] также правильные
- Если выражения A и B правильные, то выражение AB тоже правильное.

Дана последовательность скобок (,),[и]. Требуется найти самое короткое правильное выражение, в котором данная последовательность является подпоследовательностью, то есть такое, из которого можно вычеркнуть некоторые символы (возможно, ноль) и получить исходную последовательность, не меняя порядок оставшихся.

18. Непустая строка, содержащая некоторое слово, называется палиндромом, если это слово читается как слева направо, так и справа налево. Пусть дана строка, в которой записано слово S, состоящее из N прописных букв латинского алфавита. Вычеркиванием из этого слова некоторого набора символов можно получить строку, которая будет палиндромом. Требуется найти количество способов вычеркивания из данного слова некоторого (возможно, пустого) набора таких символов, что полученная в результате строка является палиндромом. Способы, различающиеся только порядком вычеркивания символов, считаются одинаковыми.

19. Строку Фибоначчи F(K) для натуральных чисел K определим так: F(1)='A', F(2)='B', F(K)=F(K-1)+F(K-2), где «+» означает конкатенацию строк. Требуется найти количество вхождений строки S, состоящей из символов A и B, в строку Фибоначчи F(N), N≤45 (длина F(45) равна 1134903170). Длина S до 25 символов.

20. Кубик, грани которого помечены цифрами от 1 до 6, бросают N раз. Найти вероятность того, что сумма выпавших чисел будет равна Q.

21. В сообщении, состоящем из одних русских букв и пробелов, каждую букву заменили ее порядковым номером в русском алфавите, а пробел – нулем. Требуется по заданной последовательности цифр найти количество исходных

сообщений, из которых она могла получиться. Ограничения: цифр не более 100, время 1 с.

22. Заданы вес E пустой копилки и вес F копилки с монетами. В копилке могут находиться монеты N видов; известны ценность Pi каждого вида монет и вес Wi одной монеты. Найти минимальную и максимальную суммы денег, которые могут находиться в копилке.

23. В таблице из N строк и N столбцов клетки заполнены цифрами от 1 до 9. Требуется найти такой путь из клетки (1,1) в клетку (N,N), чтобы сумма цифр в клетках, через которые он пролегает, была минимальной; из каждой клетки ходить можно только вниз или вправо. $2 \le N \le 250$, время 1 с.

24. вершинным покрытием неориентированного графа G=(V, E) мы называем некоторое семейство его вершин V' с таким свойством: для всякого ребра (u, v) графа G хотя бы один из его концов u или v содержится в V'. Размером вершинного покрытия считаем количество входящих в него вершин. Найти вершинное покрытие минимального размера, используя «жадный» алгоритм.

25. Известно, что плоская фигура может быть обведена за один прием, «не отрывая карандаша от бумаги», если она содержит две или ни одной точки, в которой сходится нечетное число линий. Фигура задана множеством пар номеров вершин $< i, j >$, соединенных линиями. Найти путь обхода фигуры или показать, что его не существует.

26. Задача о ранце. Из n предметов, обладающих каждый весом v_i и стоимостью p_i, $i = 1, 2, ..., n$, выбрать такие, что при суммарном весе не более V суммарная стоимость максимальна.

27. Пусть n красных и n синих точек на плоскости заданы своими координатами. Построить n отрезков с разноцветными концами, суммарная длина которых минимальна (каждая точка является концом только одного отрезка).

28. Минимальное дерево-остов. На плоскости своими координатами задано n точек. Построить связный граф с вершинами во всех этих точках так, чтобы суммарная длина его ребер была наименьшей. **Указание.** Для решения задачи достаточно начиная с любой точки на каждом шаге присоединять к связной части графа ближайшую к ней несвязную точку.

29. В массиве $Z(n)$ найти наибольшую по количеству элементов арифметическую прогрессию (элементы прогрессии стоят в массиве в произвольном порядке).

30. Задача о раскрое. Из прямоугольника размером $a \times b$ требуется вырезать возможно большее число прямоугольников размером $c \times d$. Найти оптимальный вариант раскроя (возможно, таких вариантов несколько).

Практическое занятие - 8.

Тема: Структуры данных и алгоритмы внешней памяти. Алгоритмы управления памятью.

Цель работы: изучение файлового типа данных, эффективных алгоритмов работы со внешней памятью и управлению ею.

В результате выполнения практической работы студенты должны:

- *знать* особенности внешней памяти и работы с ней; понятие файла и основные операции для работы с файлами; алгоритмы эффективного распределения памяти;
- *уметь* использовать файловый тип данных при решении задач; применять различные алгоритмы управления памятью.

Файл можно рассматривать как связанный список блоков.

Базовой операцией, выполняемой по отношению к файлам, является перенос одного блока в *буфера* находящийся в основной памяти. Буфер представляет собой зарезервированную область в основной памяти, размер которой соответствует размеру блока. Типичная операционная система обеспечивает чтение блоков в том порядке, в каком они появляются в списке блоков, который содержит соответствующий файл.

Время, необходимое для поиска блока и чтения его в основную память, достаточно велико в сравнении со временем, которое требуется для относительно простой обработки данных, содержащихся в этом блоке.

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

8.1. Сортировка слиянием.

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 ,... , 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 $r_{k(i-1)+1}$, $r_{k(i-1)+2}$, ..., 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. 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).

Итак, начинаем с разделения всех n записей на два файла f_1 и f_2 (желательно, чтобы записей в этих файлах было поровну). Можно считать, что любой файл состоит из серий длины 1. Затем мы можем объединить серии длины 1 и распределить их по файлам gj и g_2, организованным в виде серий длины 2. Мы делаем f_1 и f_2 пустыми и объединяем g_1 и g_2 в f_1 и f_2, которые затем можно организовать в виде серий длины 4. Затем мы объединяем f_1 и f_2, создавая g_1 и g_2, организованные в виде серий длиной 8, и т.д.

После выполнения i подобного рода проходов у нас получатся два файла, состоящие из серий длины 2^i. Если $2^i \geq n$, тогда один из этих двух файлов будет пустым, а другой будет содержать единственную серию длиной n, т.е. будет отсортирован.

Если "узким местом" является обмен данными между основной и вторичной памятью, возможно, удалось бы сэкономить время за счет увеличения числа каналов обмена данными. Мы могли бы разместить на m дисководах m файлов $(f_1, f_2, \ldots, f_m)$, организованных в виде серий длины k. Тогда можно прочитать m серий, по одной из каждого файла, и объединить их в одну серию длиной mk. Эта серия помещается в один из m выходных файлов $(g_1, g_2, \ldots, g_m)$, каждый из которых получает по очереди ту или иную серию.

Многоканальную (m-канальную) сортировку слиянием можно выполнить с помощью лишь $m + 1$ файлов (в отличие от описанной выше $2m$-файловой стратегии). При этом выполняется ряд проходов с объединением серий из m файлов в более длинные серии в $(m + 1)$-м файле. Вот последовательные шаги такой процедуры.

1. 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.
2. 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. (Если выполнено менее m проходов, можно считать, что гипотетические проходы, выполненные до первого прохода, создавали серии длины 1.)

Подобный процесс сортировки слиянием называется *многофазной сортировкой.*

Когда "узким местом" является считывание файлов, необходимо очень тщательно выбирать блок, который должен считываться следующим.

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.

В соответствии с нашими предположениями, эти действия занимают одинаковое время. Для обеспечения максимальной эффективности их следует выполнять параллельно. Это можно делать, если выбор записей с наименьшими ключами не включает записи, считываемые в данный момент.[1] Следовательно, 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).

8.2. 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*. операторы для работы с файлами.

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

Способы хранения данных во внешней памяти:

- Простая организация данных
- Хешированные файлы
- Индексированные файлы
- Несортированные файлы с плотным индексом
- Вторичные индексы

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. Оно представляет собой обобщение 2-3 дерева, которое обсуждалось в разделе 5.4. 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 $\lceil m/2 \rceil$ 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.

8.3. Управление памятью

В работе компьютерных систем нередко возникают ситуации, когда ограниченными ресурсами основной памяти приходится *управлять*, т.е. разделять между несколькими совместно использующими ее "конкурентами".

		Утилизация неиспользуемого пространства: явная	Утилизация неиспользуемого пространства: "сборка мусора"
Размер блока	фиксированный	Файловая система	Lisp
	переменный	Мультипрограммная система	Snobol

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.

8.4. Выделение памяти для объектов разного размера

Рассмотрим теперь управление динамической памятью (кучей), когда существуют указатели на выделенные блоки (как показано на рис. 12.1). 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);

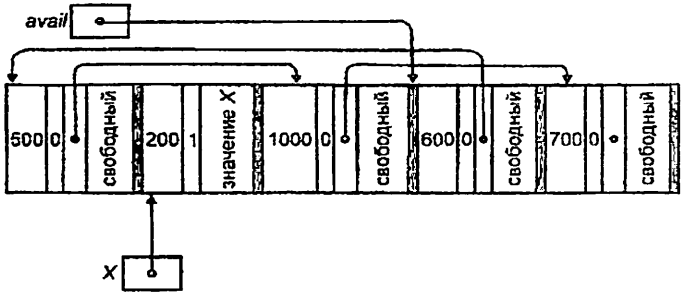
б) указателя (для связи данного блока со свободным пространством);

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.

1. 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.
2. 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.

8.5. Выбор свободных блоков

Две стратегии, которые представляют реализацию противоположных требований, называются "первый подходящий" (first-fit) и "самый подходящий" (best-fit). Описание этих стратегий приведено ниже.

1. *Первый подходящий.* В соответствии с этой стратегией, если требуется блок размера d, нужно просматривать список блоков свободного пространства с самого начала и до тех пор, пока не встретится блок размера $c \geq d$. Затем нужно использовать последние d байт (или слов) этого блока, как было описано выше.
2. *Самый подходящий.* В соответствии с этой стратегией, если требуется блок размера d, нужно проанализировать весь список блоков свободного пространства и найти блок размера не менее d, причем размер этого блока должен как можно меньше превышать величину d. Затем нужно использовать последние d байт этого блока.

8.6. Методы близнецов

Разработано целое семейство стратегий управления динамической памятью, которое позволяет частично решить проблему фрагментации и неудачного распределения размеров пустых блоков. Эти стратегии, называемые *методами близнецов* (buddy systems), на практике затрачивают очень мало времени на объединение смежных пустых блоков. Недостаток метода близнецов заключается в том, что блоки имеют весьма ограниченный набор размеров, поэтому, возможно, придется нерационально расходовать память, помещая элемент данных в блок большего размера, чем требуется.

Главная идея, лежащая в основе всех методов близнецов, заключается в том, что все блоки имеют лишь строго определенные размеры $s_1 < s_2 < s_3 < \ldots < s_k$. Характерными вариантами последовательности чисел s_1, s_2, ... являются числа 1, 2, 4, 8, ... *(метод близнецов экспоненциального типа)* и 1, 2, 3, 5, 8, 13, ... *(метод близнецов с числами Фибоначчи,* где $s_{i+1} = s_i + s_{i-1}$). Все пустые блоки размера s_i связаны в список; кроме того, существует массив заголовков списков свободных блоков, по одному для каждого допустимого размера s_i.[1] Если для нового элемента данных требуется блок размером d, то выбирается свободный блок такого размера s_i чтобы $s_i \geq d$, однако $s_{i-1} < d$, т.е. наименьший допустимый размер, в который умещается новый элемент данных.

Если допустить, что $j = i - k$ для некоторого $k \geq 0$, тогда, поскольку $s_{i+1} - s_i = s_{i-k}$, следует, что $s_{i+1} = s_i + s_{i-k}$ Какое бы значение k в уравнении (12.1) мы ни выбрали, получим *метод близнецов k-го порядка.* При использовании метода близнецов k-го порядка можно считать, что каждый блок размером s_{i+1} состоит из блока размером s_i и блока размером s_{i-k}. Допустим для определенности, что блок размером s_i находится слева (в позициях с меньшими значениями номеров) от блока размером s_{i-k}.[3] Если динамическую память рассматривать как единый блок размером s_n (при некотором большом значении n), тогда позиции, с которых могут начинаться блоки размером s_i будут полностью определены.

8.7. Уплотнение памяти

Существуют два общих подхода к решению этой проблемы.

– 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). Если бы, например, мы хранили данные, размер которых чаще всего невелик, то можно было бы выбрать блоки по 16 байт (4 — для указателя и 12 — для данных). Если же элементы данных обычно имеют большие размеры, следовало бы выбрать блоки размером несколько сотен байт (по-прежнему 4 байта для указателя, а остальное для данных).

– 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.

Пример. Написать программу, позволяющую работать с хранимыми в последовательном файле данными об абонентах АТС. Запись об абоненте должна содержать номер телефона, фамилию, адрес (номер дома и квартиры), данные о том, заблокирован ли абонент, и размер его задолженности. Реализовать возможность добавления, удаления, модификации и поиска записей.

```
uses crt;
type abon=record
num:integer;
fam:string[20];
house:integer;
flat:integer;
blk:boolean;
debt:real;
end;

var db:file of abon;
   fname:string;

procedure add(a:abon);
begin
seek(db,filesize(db));
write(db,a);
end;

procedure inpname(var a:abon);
begin
a.fam:='';
repeat
writeln('name:');
readln(a.fam);
until a.fam<>'';
end;

procedure inphouse(var a:abon);
```

```
var s1:string;n:integer;
begin
a.house:=0;
repeat
writeln('house');
readln(s1);
val(s1,a.house,n);
if (n<>0) then writeln('ne chislo');
if a.house<=0 then writeln('<0');
until (n=0) and (a.house>0);
end;
procedure inpflat(var a:abon);
var s1:string;n:integer;
begin
a.flat:=0;
repeat
writeln('flat');
readln(s1);
val(s1,a.flat,n);
if (n<>0) then writeln('ne chislo');
if a.flat<=0 then writeln('<0');
until (n=0) and (a.flat>0);
end;

procedure inpblk(var a:abon);
var s1:string:
begin
a.blk:=false;
repeat
writeln ('blocked? y/n');
readln(s1);
if (s1='y') or (s1='Y')  then a.blk:=true;
until (s1='n') or (s1='N') or a.blk;
end;

procedure inpdebt(var a:abon);
var s1:string;n:integer;
begin
a.debt:=0;
repeat
writeln('debt');
readln(s1);
val(s1,a.debt,n);
if (n<>0) then writeln('ne chislo');
until (n=0);
```

```
end;

procedure addnewabon;
var a:abon;
s1:string;n:integer;
i,j:integer;
rep:boolean;
begin
repeat
repeat
writeln('ID');
readln(s1);
val(s1,i,n);
if (n<>0) then writeln('ne chislo');
until (n=0);
rep:=false;
reset(db);
for j:=0 to filesize(db)-1 do
 begin
 read(db,a);
 if a.num=i then begin
              rep:=true;
              break;
              end;
 end;
if rep then writeln('Povtor')
     else begin
          a.num:=i;
          inpname(a);
          inphouse(a);
          inpflat(a);
          inpblk(a);
          inpdebt(a);
          add(a);
          end;
until not(rep);
end;

procedure printabon(a:abon);
begin
with a do
writeln(num:5,fam:20,house:4,flat:4,blk:7,debt:11:2);
end;

function findabonfam(s:string):integer;
```

```
 var i:integer;
 a:abon;
 begin
 reset(db);
 for i:=0 to filesize(db)-1 do
  begin
  read(db,a);
  if a.fam=s then begin
              printabon(a);
              findabonfam:=i;
              exit;
              end;
  end;
 writeln('Not found');
 findabonfam:=-1;
 end;

 function findabonnum(n:integer):integer;
 var i:integer;
 a:abon;
 begin
 reset(db);
 for i:=0 to filesize(db)-1 do
  begin
  read(db,a);
  if a.num=n then begin
              printabon(a);
              findabonnum:=i;
              exit;
              end;
  end;
 writeln('Not found');
 findabonnum:=-1;
 end;
 procedure modifyabon(n:integer);
 var a:abon;
    c:char;
 begin
 seek(db,n);
 read(db,a);
 with a do
 begin
  writeln('ID:',a.num);
  writeln('Familiya:',a.fam);
  repeat
```

```
 writeln('izmenit? (y/n)');
 c:=readkey;
 if c='y' then inpname(a);
 until (c='y')or(c='n');
 writeln('Dom:',a.house);
 repeat
 writeln('izmenit? (y/n)');
 c:=readkey;
 if c='y' then inphouse(a);
 until (c='y')or(c='n');
 writeln('Kvartira:',a.flat);
 repeat
 writeln('izmenit? (y/n)');
 c:=readkey;
 if c='y' then inpflat(a);
 until (c='y')or(c='n');
 writeln('Blokirovan:',a.blk);
 repeat
 writeln('izmenit? (y/n)');
 c:=readkey;
 if c='y' then inpblk(a);
 until (c='y')or(c='n');
 writeln('Zadoljennost:',a.debt);
 repeat
 writeln('izmenit? (y/n)');
 c:=readkey;
 if c='y' then inpdebt(a);
 until (c='y')or(c='n');
end;
seek(db,n);
write(db,a);
end;

procedure delabon(n:integer);
var i:integer;
a:abon;c:char;
begin
seek(db,n);
read(db,a);
 repeat
 writeln('udalit? (y/n)');
 c:=readkey;
 if c='y' then begin

for i:=n to filesize(db)-2 do
```

```
 begin
 seek(db,i+1);
 read(db,a);
 seek(db,i);
 write(db,a);
 end;
 seek(db,filesize(db)-1);
 truncate(db);end;
 until (c='y')or(c='n');
 end;

 procedure showall;
 var a:abon;
 begin
 reset(db);
 while not eof(db) do
  begin
  read(db,a);
  printabon(a);
  end;
 readkey;
 end;


 var i,n:integer;
    ans:char;
    s:string;
 begin
 assign(db,'aaa');
 {$I-}reset(db);{$i+}
 if ioresult<>0 then  rewrite(db);

 repeat
 clrscr;
 writeln('Baza abonentov ATS');
 writeln;
 writeln('1 - prosmotret bazu');
 writeln('2 - dobavit zapis');
 writeln('3 - nayti po nomeru');
 writeln('4 - nayti po familii');
 writeln('5 - udalit (po nomeru)');
 writeln('6 - izmenit (po nomeru)');
 writeln('7 - vihod');

 ans:=readkey;
```

```
case ans of
'1': begin
    clrscr;
    showall;
    end;
'2': addnewabon;
'3': begin
    clrscr;
    repeat
    writeln('Vvedite nomer');
    readln(s);
    val(s,i,n);
    if n<>0 then writeln('Oshibka;vvedtite snova');
    until (n=0);
    findabonnum(i);
    readkey;
    end;
'4': begin
    clrscr;
    writeln('Vvedite familiyu');
    readln(s);
    findabonfam(s);
    readkey;
    end;
'5': begin
    clrscr;
    repeat
    writeln('Vvedite nomer');
    readln(s);
    val(s,i,n);
    if n<>0 then writeln('Oshibka;vvedtite snova');
    until (n=0);
    n:=findabonnum(i);
    if n>=   0 then
    delabon(n) else readkey;
    end;
'6': begin
    clrscr;
    repeat
    writeln('Vvedite nomer');
    readln(s);
    val(s,i,n);
    if n<>0 then writeln('Oshibka;vvedtite snova');
    until (n=0);
    n:=findabonnum(i);
```

```
    if n>=0 then modifyabon(n)else readkey;
    end;
end;
until ans='7';

close(db);
end.
```

Задание.

1. Подготовить список из N наименований товаров. Информация о каждом товаре содержит:
   - Название товара.
   - Цену.
   - Год выпуска.
   - Количество.

Товары упорядочены по невозрастанию года выпуска.

Разработать программу, которая заносит во внешний файл записи упорядоченного списка, и программу, которая добавляет в сформированный внешний файл данные об M товарах, при этом, не нарушая упорядоченности исходного файла. Если среди добавляемых товаров встречается товар, сведения о котором в файле уже есть, то необходимо их обновить, т. е. старую запись исключить.

2. Рассмотрим массив размера n. Разработайте алгоритм циклического сдвига против часовой стрелки всех элементов в таком массиве на k позиций, используя только фиксированную дополнительную память, объем которой не зависит от k и n. *Совет.* Проанализируйте, что произойдет, если переставить в обратном порядке первые k элементов, последние $n - k$ элементов и, наконец, весь массив.

3. Дан указатель P_1 на начало непустой цепочки элементов-записей типа TNode, связанных между собой с помощью поля Next. Преобразовать исходную (*односвязную*) цепочку в *двусвязную*, в которой каждый элемент связан не только с последующим элементом, но и с предыдущим. Поле Prev первого элемента положить равным nil. Вывести указатель на последний элемент преобразованной цепочки.

4. Разработайте алгоритм замены вложенной строки символов y, принадлежащей строке xyz, на другую вложенную строку y' используя для этого как можно меньшее количество дополнительной памяти. Какова временная сложность (время выполнения) этого алгоритма и сколько для него необходимо памяти?

5. Напишите программу получения копии заданного списка. Какова временная сложность этого алгоритма и сколько для него необходимо памяти?

6. Напишите программу, которая проверяла бы идентичность двух списков. Какова временная сложность этого алгоритма и сколько для него необходимо памяти?

7. Реализуйте алгоритм Морриса для уплотнения динамической памяти.

8. Разработайте схему выделения памяти применительно к ситуации, когда

память выделяется и освобождается блоками длиной 1 и 2. Перечислите преимущества и недостатки такого алгоритма.

9. Составьте программу *concatenate* (конкатенация), которая принимает последовательность имен файлов в качестве аргументов и переписывает содержимое этих файлов на стандартное устройство вывода, осуществляя таким образом конкатенацию этих файлов.

10. Составьте программу *include* (включить в себя), которая копирует свой вход на свой выход за исключением случая, когда ей встречается строка в форме #include *имя_файла;* в этом случае программа должна заменить такую строку на содержимое названного файла. Обратите внимание, что включенные файлы также могут содержать операторы #include.

11. Составьте программу *compare* (сравнение), которая сравнивает два файла, запись за записью, чтобы выяснить, идентичны ли эти файлы.

12. Составьте программу *find* (поиск), которая имеет два аргумента, состоящих из строки шаблона и имени файла, и распечатывает все строки файла, содержащие указанную строку шаблона в качестве вложенной строки. Если, например, строкой шаблона является "ufa", а файл представляет собой список слов, тогда *find* распечатывает все слова, содержащие указанные три буквы.

13. Допустим, имеется внешний файл, содержащий ориентированные дуги x —> y, которые образуют ориентированный ациклический граф. Допустим, что не хватает свободного объема оперативной памяти для одновременного хранения всей совокупности вершин или дуг. Составьте программу внешней топологической сортировки, которая распечатывает такую линейно упорядоченную последовательность вершин, что если x —> y представляет собой дугу, то в данной последовательности вершина *x* появляется до вершины *y*.

14. Допустим, что есть внешний файл записей, каждая из которых представляет ребро графа *G* и стоимость этого ребра. Напишите программу построения остовного дерева минимальной стоимостью для графа *G*, полагая, что объем основной памяти достаточен для хранения всех вершин графа, но не достаточен для хранения всех его ребер.

15. Допустим, что имеется файл, содержащий последовательность положительных и отрицательных чисел $a_1, a_2, \ldots, a_n$. Напишите программу с временем выполнения $O(n)$, которая находила бы непрерывную вложенную подпоследовательность $a_i, a_{i+1}, \ldots, a_j$, которая имела бы наибольшую сумму $a_i + a_{i+1} + \ldots + a_j$ среди всех вложенных подпоследовательностей такого рода.

16. Прямоугольный садовый участок шириной N и длиной M метров разбит на квадраты со стороной 1 м. На этом участке вскопаны грядки. Грядкой называется совокупность квадратов, удовлетворяющая таким условиям:

- из любого квадрата этой грядки можно попасть в любой другой квадрат этой же грядки, последовательно переходя по грядке из квадрата в квадрат через их общую сторону

- никакие две грядки не пересекаются и не касаются друг друга ни по вертикальной, ни по горизонтальной сторонам квадратов (касание грядок углами квадратов допускается).

Входной файл содержит числа N и M через пробел, а затем N строк по M символов. # означает территорию грядки, . – невскопанную территорию. Подсчитать количество грядок.

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 с.

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 практических 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 № 451
Тираж. 100
Отпечатано в типографии ТУИТ
Ташкент 700084, ул.А.Тимура – 108