

Оглавление

Введение.....	6
Основные понятия и определения.....	8
Лабораторная работа №1. Встроенные структуры данных (Pascal/C).....	11
Задание.....	11
Содержание отчета.....	15
Теоретические сведения.....	15
Простые типы данных в Pascal.....	15
Целые типы.....	15
Символьный тип (char).....	15
Логический тип (boolean).....	16
Перечисляемый тип.....	16
Тип-диапазон.....	16
Вещественные типы.....	16
Сложный тип.....	18
Простые типы данных в C.....	18
Целые типы.....	18
Символьный тип.....	19
Перечисляемый тип.....	19
Вещественные типы.....	20
Структурированные типы данных в Pascal.....	21
Массив.....	21
Структура данных типа «запись».....	21
Структура данных типа «множество».....	22
Структурированные типы данных в C.....	23
Структура данных типа «массив».....	23
Структура данных типа «структура».....	24
Контрольные вопросы.....	25
Лабораторная работа № 2. Производные структуры данных. Структура данных «строка» (Pascal/C).....	26
Задание.....	26
Содержание отчета.....	42
Теоретические сведения.....	42
Контрольные вопросы.....	47
Лабораторная работа № 3. Сравнительный анализ методов сортировки (Pascal/C).....	47
Задание.....	47
Содержание отчета.....	49
Теоретические сведения.....	49
Сортировка включением.....	50
Анализ сортировки включением.....	51
Сортировка выбором.....	51
Анализ сортировки простым выбором.....	52
Сортировка обменом.....	53
Анализ сортировки обменом.....	53
Улучшенная сортировка обменом 1.....	54
Анализ улучшенной сортировки обменом 1.....	54
Улучшенная сортировка обменом 2.....	54
Анализ улучшенной сортировки обменом 2.....	54
Сортировка Шелла.....	54

Анализ сортировки Шелла.....	55
Сортировка Хоара.....	56
Анализ сортировки Хоара.....	56
Пирамидальная сортировка.....	57
Анализ пирамидальной сортировки.....	60
Примеры программной реализации алгоритмов сортировки на языке Pascal.....	60
Примеры программной реализации алгоритмов сортировки на языке C.....	65
Контрольные вопросы.....	69
Лабораторная работа № 4. Сравнительный анализ алгоритмов поиска (Pascal/C).....	70
Задание.....	70
Содержание отчета.....	70
Теоретические сведения.....	72
Алгоритмы поиска в неупорядоченных массивах.....	72
Алгоритм линейного поиска.....	72
Алгоритм быстрого линейного поиска.....	72
Анализ алгоритмов линейного поиска.....	72
Алгоритмы поиска в упорядоченных массивах.....	73
Алгоритм быстрого линейного поиска.....	73
Алгоритм бинарного поиска.....	73
Анализ алгоритма бинарного поиска.....	74
Алгоритм блочного поиска.....	74
Анализ алгоритма блочного поиска.....	74
Контрольные вопросы.....	75
Лабораторная работа №5. Структуры данных «линейные списки» (Pascal/C).....	76
Задание.....	76
Содержание отчета.....	93
Теоретические сведения.....	93
Контрольные вопросы.....	97
Лабораторная работа №6. Структуры данных «стек» и «очередь» (Pascal/C).....	98
Задание.....	98
Содержание отчета.....	137
Теоретические сведения.....	137
Стек.....	137
Очередь.....	140
Контрольные вопросы.....	143
Лабораторная работа №7. Структуры данных типа «дерево» (Pascal/C).....	144
Задание.....	144
Содержание отчета.....	161
Теоретические сведения.....	161
Принципы размещения бинарного дерева в памяти ЭВМ.....	163
Алгоритмы обхода бинарного дерева.....	165
Обход бинарного дерева «в глубину» (в прямом порядке).....	165
Обход бинарного дерева «в ширину» (по уровням).....	166
Обход бинарного дерева в симметричном порядке.....	166

Обход бинарного дерева в обратном порядке.....	167
Алгоритмы формирования бинарного дерева.....	167
Рекурсивный алгоритм формирования бинарного дерева	
«в глубину».....	168
Итеративный алгоритм формирования бинарного дерева	
«в глубину».....	168
Алгоритм формирования бинарного дерева «в ширину»	169
Алгоритм формирования бинарного дерева «снизу вверх»	169
Рекурсивный алгоритм формирования бинарного дерева	170
Итеративный алгоритм формирования бинарного дерева	170
Алгоритм формирования бинарного дерева минимальной	
высоты.....	171
Итеративный алгоритм формирования сбалансированного	
бинарного дерева.....	172
Представление алгебраических выражений бинарными	
деревьями.....	172
Алгоритм формирования бинарного дерева по прямой	
польской записи.....	173
Алгоритм формирования бинарного дерева по обратной	
польской записи.....	173
Контрольные вопросы.....	173
Лабораторная работа №8. Структуры данных типа «таблица» (Pascal/C).....	174
Задание	174
Содержание отчета	197
Теоретические сведения	198
Контрольные вопросы.....	202
Библиографический список.....	203

Введение

Структуры данных являются неотъемлемой частью любого программного продукта. При разработке программы необходимо определить множество данных, описывающих конкретную задачу и составить алгоритм решения задачи. В зависимости от назначения программы данные могут иметь разный уровень сложности или организованности, начиная с простых типов, представляющих собой числа или символы, и заканчивая файлами и системами файлов достаточно сложной структуры. Изучение структур данных, правильный выбор их в зависимости от выполняемых операций, размера требуемой для структур памяти, частоты использования структур при выполнении программы позволяет повысить эффективность разрабатываемых программ, уменьшить время и стоимость программной разработки. Знание теории структур данных и методов представления данных на логическом и физическом уровне необходимо для изучения таких дисциплин направления «Программная инженерия», как «Операционные системы и сети», «Базы данных», «Конструирование программного обеспечения».

В этом лабораторном практикуме подробно рассмотрены структуры данных и алгоритмы, которые являются фундаментом современной методологии разработки программ. Особое внимание посвящено методам анализа и построения алгоритмов.

Учебное пособие состоит из описания восьми лабораторных работ. Описание каждой лабораторной работы включает цели и постановку задачи, необходимые теоретические сведения, индивидуальные варианты заданий. Тематика лабораторных работ соответствует требованиям, определенным государственными образовательными стандартами по данной дисциплине.

Лабораторная работа №1 посвящена изучению встроенным структурам данных языков программирования Pascal и C. Особое внимание уделяется изучению физического уровня представления этих структур.

Тематика лабораторной работы №2 связана с производными структурами данных, то есть такими структурами, которые реализует сам программист. Ему предлагается создать собственную структуру строкового типа, в соответствии с вариантом задания, и, используя ее, решить указанную задачу.

В лабораторной работе №3 изучаются базовые и улучшенные методы сортировки в основной памяти. Для сравнительного анализа алгоритмов сортировки необходимо реализовать вычислительный эксперимент для построения функции временной сложности и уметь сделать выбор для этих алгоритмов вид порядка функции временной сложности.

Лабораторная работа №4 посвящена сравнительному анализу функции временной сложности и ее порядка алгоритмов поиска.

В лабораторной работе №5 изучается производная структура данных «односвязный линейный список». Созданная студентом эта структура данных используется для решения задачи в соответствии с вариантом.

Лабораторная работа №6 посвящена классическим структурам данных «стек» и «очередь». Они реализуются студентом, причем как отображение на массив, либо как отображение на односвязный линейный список, и затем используются для моделирования модельной вычислительной системы согласно варианту.

В лабораторной работе №7 изучается нелинейная структура данных «бинарное дерево». Эта структура реализуется как в последовательной, так и в связной памяти. Затем она используется для решения задачи, которая указана в варианте заданий.

Лабораторная работа №8 посвящена широко используемой структуре данных «таблица». Эта структура данных реализуется как в линейном, так и в нелинейном варианте. С помощью созданной структуры данных также решается конкретная задача.

Чтобы еще раз подчеркнуть важность правильного выбора структуры данных можно воспользоваться следующей формулой:

Алгоритм = метод + структуры данных

Термин «метод» будет обозначать множество алгоритмов, объединенных главной идеей. Появление нового метода – это событие и событие крайне редкое, ведущее обычно к радикальному ускорению обработки данных. Внедрение новых композиций структур данных, напротив, ярким событием не является, но зачастую в несколько раз уменьшает время обработки данных без изменения метода.

Основные понятия и определения

Структура данных (СД) — общее свойство информационного объекта, с которым взаимодействует та или иная программа. Это общее свойство характеризуется:

- 1) множеством допустимых значений данной структуры;
- 2) набором допустимых операций;
- 3) характером организованности.

Различают следующие уровни описания СД:

- 1) абстрактный (математический) уровень;
- 2) логический уровень;
- 3) физический уровень.

Абстрактный уровень определяет характер организованности СД. Организованность СД может быть представлена множеством элементов, каждый из которых представляет собой СД, и отношениями между ними, свойства которых определяют различные типы СД.

Основные типы СД:

1. *Множество* (рис.1) — совокупность независимых элементов, отношения между которыми не заданы.

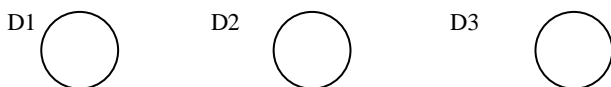


Рис. 1. СД типа «множество»

2. *Последовательность* (рис.2) — множество элементов, над которыми определены отношения линейного порядка, т.е. для каждого элемента, может быть за исключением первого и последнего, имеется один предыдущий и один последующий.

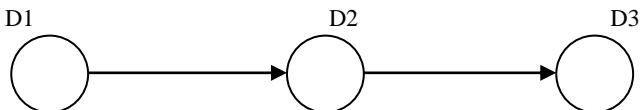


Рис. 2. СД типа «последовательность»

3. *Дерево* (рис.3) — множество элементов, над которыми определены отношения иерархического порядка, т.е. «один ко многим».

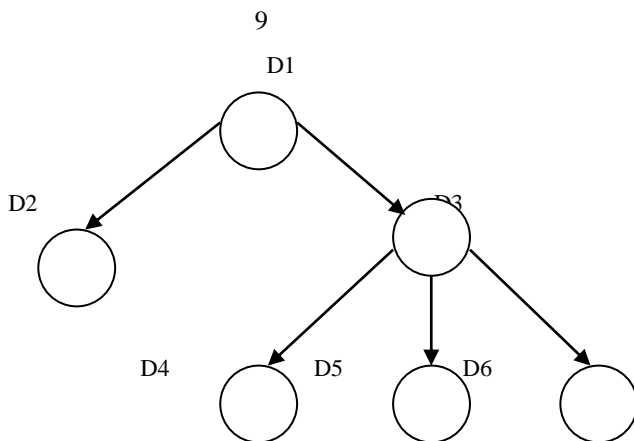


Рис.3. СД типа «дерево»

4. *Граф* (рис.4) — множество элементов, на которых определены отношения бинарного порядка, т.е. “многие ко многим”.

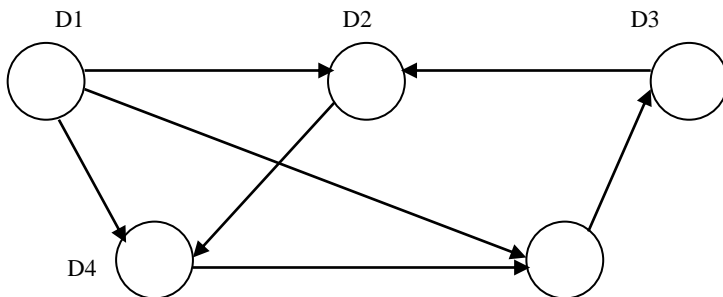


Рис.4. СД типа «граф»

Важным свойством СД является ее изменчивость во время выполнения программы. Если количество элементов и/или отношения между ними изменяются, то такие СД называют *динамическими*, иначе — *статическими*.

Логический уровень — представление СД на языке программирования. Простейшие СД, представляющие собой один элемент, определяются простыми типами. В языке *Pascal* для переменных простого типа определены множества допустимых значений и набор допустимых операций. Характер организованности — простейший.

Структурные СД, представляющие собой совокупность простейших СД и отношения между ними, могут быть определены программистом в виде структурного типа: множество, массив, запись. В этом случае набор допу-

стимых значений зависит от простых типов, на основе которых построена СД и структурного типа, представляющего собой СД. Множество допустимых операций и характер организованности определяется структурным типом. Т.к. рассмотренные СД полностью определяются типами данных языка программирования, то их можно назвать *встроенными*. Для других СД, например, список, стек, очередь, дерево, таблица и др., нет соответствующих типов, определяющих организованность этих данных и допустимые операции. Такие СД называются *производными* и реализуются непосредственно программистами. Одна и та же производная СД может быть реализована различными способами, но множество допустимых значений, набор допустимых операций, характер организованности и изменчивости остаются неизменными, т.к. это свойство самой СД, а не способа ее реализации.

Физический уровень — отображение на память ЭВМ информационного объекта в соответствии с логическим описанием. На этом уровне определяются область и объем памяти, необходимый для хранения экземпляра СД, форматы и интерпретация внутреннего представления. На физическом уровне, в памяти ЭВМ, СД могут иметь последовательную (прямоугольную) или связанную схему хранения, располагаться в статической или динамической памяти. При *последовательной* схеме хранения данные располагаются в соседних, последовательно расположенных ячейках памяти. Физический порядок следования данных полностью соответствует логическому. При *связной* схеме хранения каждый элемент СД имеет ссылку на другой (другие) элементы. Физический порядок следования данных может не соответствовать логическому.

При последовательной схеме хранения доступ к элементам СД может быть прямым, т.е. возможно обращение непосредственно к любому элементу независимо от состояния СД (от текущего обрабатываемого элемента). К СД с прямым доступом к элементам можно отнести массив, запись, последовательный список, хеш-таблицу.

При связанной схеме хранения доступ к элементам СД последовательный, т.е. в зависимости от текущего обрабатываемого элемента доступ возможен к одному или некоторому ограниченному числу элементов. К СД с последовательным доступом к элементам можно отнести связанные линейные списки и СД, построенные на их основе: стек, дек, очередь, таблицы (за исключением хеш-таблиц с прямым доступом), деревья, графы.

Объем памяти, занимаемый экземпляром СД, зависит от объема памяти, занимаемой одним элементом, и характера изменчивости СД. Объем памяти, занимаемый экземпляром статической СД (массив, запись) не изменяется в процессе выполнения программы. Объем памяти, занимаемый экземпляром динамической СД может быть постоянным в процессе выполнения программы (например СД типа «строка» или динамические СД, реализо-

ванные на основе массива), или изменяться в соответствии с количеством элементов СД, если при включении/исключении элемента выделяется/освобождается достаточный для его хранения объем динамической памяти.

Память (статическая или динамическая), используемая для хранения экземпляра СД, и схема хранения СД определяется реализацией СД и не является свойством СД.

Лабораторная работа № 1

Встроенные структуры данных(Pascal/C)

Цель работы: изучение базовых типов данных языка Pascal/C как структур данных (СД).

З а д а н и е

1. Для типов данных (см. Варианты заданий в таблицах 1,2) определить:
 - 1.1. Абстрактный уровень представления СД:
 - 1.1.1. Характер организованности и изменчивости.
 - 1.1.2. Набор допустимых операций.
 - 1.2. Физический уровень представления СД:
 - 1.2.1. Схему хранения.
 - 1.2.2. Объем памяти, занимаемый экземпляром СД.
 - 1.2.3. Формат внутреннего представления СД и способ его интерпретации.
 - 1.2.4. Характеристику допустимых значений.
 - 1.2.5. Тип доступа к элементам.
 - 1.3. Логический уровень представления СД.
 Способ описания СД и экземпляра СД на языке программирования.
2. Для заданных типов данных определить набор значений, необходимый для изучения физического уровня представления СД.
3. Преобразовать значения в двоичный код.
4. Преобразовать двоичный код в значение.
5. Разработать и отладить программу, выдающую двоичное представление значений, заданных СД.

В программе использовать процедуры *PrintByte* и *PrintVar*.

Спецификация процедуры *PrintByte*:

1. Заголовок: *procedure PrintByte(a:byte)/void PrintByte(unsigned char a)*.
2. Назначение: выводит на экран монитора двоичное представление переменной *a* типа *byte/unsigned char*.
3. Входные параметры: *a*.
4. Выходные параметры: нет.

Рекомендации: использовать побитовые операции сдвига и логического умножения.

Спецификация процедуры *PrintVar*:

1. Заголовок: *procedure PrintVar(var a; size:word)/ void PrintVar(void a, unsigned int size)*.
2. Назначение: выводит на экран монитора двоичное представление переменной *a* произвольного типа размером *size* байт.
3. Входные параметры: *a* — переменная произвольного типа, значение которой выводится на экран в двоичном представлении (нетипизованный параметр); *size* — объем памяти (в байтах) занимаемый переменной *a*.
4. Выходные параметры: нет.

Рекомендации: нетипизованную переменную *a* привести к типу «массив байт», значение каждого элемента которого выводить на экран в двоичном представлении процедурой *PrintByte*.

6. Обработать программой значения, полученные в результате выполнения пункта 3 задания. Сделать выводы.

7. Разработать и отладить программу, определяющую значение переменной по ее двоичному представлению по следующему алгоритму:

1. Ввести двоичный код в переменную *S* строкового типа.
2. Преобразовать *S* в вектор *B* типа «массив байт».
3. Привести *B* к заданному типу. Вывести значение.
4. Конец.

8. Обработать программой значения, полученные в результате выполнения пункта 4 задания. Сделать выводы.

Таблица 1

Варианты индивидуальных заданий на Pascal

номер варианта	тип 1	тип 2	тип 3
1	integer	real	(red, yellow, green)
2	longint	real	массив [1..3,1..3] char
3	word	double	set of byte
4	byte	single	1000..2000
5	integer	extended	'A'..'Z'
6	shortint	single	массив [1..5] real
7	longint	comp	(cat, dog, mouse, tiger)
8	byte	real	-300..300
9	word	double	(winter, spring, summer, autumn)
10	shortint	real	set of 'a'..'h'
11	byte	single	-1..1
12	word	comp	1..256
13	integer	double	set of 50..100
14	byte	extended	'a'..'h'
15	longint	double	-256..-1
16	shortint	extended	(one, two, three, four, five, six)
17	word	real	256..511
18	integer	single	1..50
19	longint	extended	(a, b, c, d, e, f, g, h)
20	byte	double	массив [1..8] char
21	shortint	comp	-10..10
22	word	single	запись
23	integer	extended	массив [1..6] boolean
24	byte	comp	(winter, spring, summer, autumn)
25	longint	double	35000..35001
26	shortint	real	set of 200..250
27	word	extended	запись
28	integer	comp	1..256
29	byte	single	массив [1..2,1..4] integer
30	shortint	double	(day, night)

Таблица 2

Варианты индивидуальных заданий на C

номер ва- рианта	тип 1	тип 2	тип 3
1	int	float	{red, yellow, green} colors

Окончание табл.2

2	longint	float	char массив[3][3]
3	unsigned short	double	{ winter,spring, summer,autumn }season
4	byte	float	float массив[10][10]
5	int	long double	структура
6	signed char	float	float массив[5]
7	longint	comp	{ cat, dog,mouse,tiger }animal
8	byte	float	{ a, b, c, d, e, f, g, h }lettre
9	unsigned short	double	{ winter,spring, summer,autumn }season
10	signed char	float	{ red, yellow, green }colors
11	byte	float	char массив[3][3]
12	unsigned short	comp	{ winter,spring, summer,autumn }season
13	int	double	double массив[3][3][3]
14	byte	long double	int массив [5][5]
15	longint	double	структура
16	signed char	long double	{ one,two,three,four, five,six }number
17	unsigned short	float	{ cat, dog,mouse,tiger }animal
18	int	float	char Массив [4][4]
19	longint	long double	{ a, b, c, d, e, f, g, h }lettre
20	byte	double	char массив [8]
21	signed char	comp	float массив[3]
22	unsigned short	float	структура
23	int	long double	int массив [6]
24	byte	comp	{ winter,spring, summer,autumn }season
25	longint	double	структура
26	signed char	float	float массив[10]
27	unsigned short	long double	структура
28	int	comp	{ winter,spring, summer,autumn }season
29	byte	float	int массив[2][4]
30	signed char	double	{ red, yellow, green }colors

Содержание отчета

1. Тема лабораторной работы.
2. Цель работы.
3. Индивидуальное задание.
4. Характеристика каждого заданного типа данных как СД в соответствии с пунктом 1 задания.
5. Набор значений заданных типов, порядок их преобразования в двоичное представление, двоичное представление значений.
6. Набор двоичных векторов, порядок их преобразования в значения заданных типов, значения заданных типов.
7. Спецификация алгоритма, текст программы, результаты работы программы, выводы (пункты 5, 6 задания).
8. Спецификация алгоритма, текст программы, результаты работы программы, выводы (пункты 7, 8 задания).

Теоретические сведения

Простые типы данных в Pascal

Целые типы

Диапазон возможных значений целых типов зависит от их внутреннего представления. В таблице 3 приводится название целых типов, длина их внутреннего представления в байтах и диапазон возможных значений.

Таблица 3

Целые типы языка Pascal

Название типа	Диапазон значений	Длина, байт
shortint	−128..127	1
integer	−32768..32767	2
longint	−2147483648..2147483647	4
byte	0..255	1
word	0..65535	2

В первом байте типа *integer*, *word* располагается младшая часть числа, во втором — старшая.

Символьный тип(char)

Диапазон возможных значений этого типа представляет собой множество всех символов. Каждому символу приписывается целое число от 0 до 255. Для кодировки используют код *ASCII*. Символьный тип занимает в памяти один байт.

Логический тип(boolean)

Множество значений: *true*(1) и *false*(0). Логический тип занимает в памяти один байт. Тип упорядочен.

Перечисляемый тип

Задается перечислением тех значений, которые он может получать. Каждое значение именуется некоторым идентификатором и располагается в списке, обрамленном круглыми скобками, например:

```
type
  colors = (red, yellow, green);
```

Соответствие между значениями перечисляемого типа и порядковыми номерами этих значений устанавливается порядком перечисления: первое значение в списке получает порядковый номер 0, второе — 1 и т.д. Максимальная мощность перечисляемого типа составляет 65536 значений, поэтому фактически перечисляемый тип является подмножеством целого типа *word*.

Тип-диапазон

Тип-диапазон есть подмножество своего базового типа, в качестве которого может выступать любой порядковый тип, кроме типа-диапазона. Он задается границами своего базового типа, например:

```
type
  date=1..31;
  month=1..12;
```

Вещественные типы

Значения этих типов определяют произвольное число лишь с некоторой конечной точностью, зависящей от внутреннего формата вещественного числа. В таблице 4 приводятся названия вещественных типов, диапазон возможных значений и длина внутреннего представления.

Таблица 4

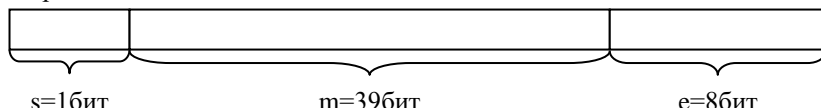
Вещественные типы языка Pascal

Название типа	Диапазон значений	Длина, байт
Real	2.9e-39..1.7e+38	6
single	1.5e-45..3.4e+38	4
double	5.0e-324..1.7e+308	8
extended	3.4e-4932..1.1e+4932	10

Внутримашинное представление вещественных типов.

Real

Формат:



Где s — знак, m — мантисса, e — порядок числа.

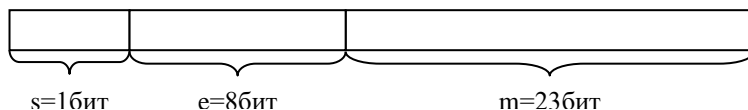
Формула для вычисления значения, хранящегося в памяти:

$$v = (-1)^s \cdot 2^{e-129} \cdot 1.m, \text{ если } 0 < e \leq 255 \text{ или } v = 0, \text{ если } e = 0.$$

Точность 11 — 12 знаков.

Single

Формат:



Формула для вычисления значения, хранящегося в памяти:

$$v = (-1)^s \cdot 2^{e-127} \cdot 1.m, \text{ если } 0 < e < 255$$

$$v = (-1)^s \cdot 2^{126} \cdot 0.m, \text{ если } e = 0 \text{ и } m \neq 0$$

$$v = (-1)^s, \text{ если } e = 0 \text{ и } m = 0$$

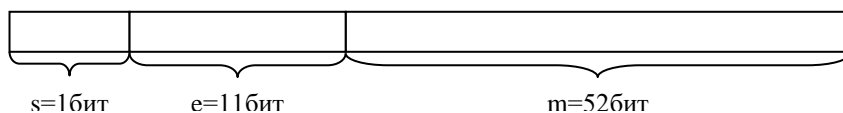
$$v = (-1)^s \cdot \text{Inf}, \text{ если } e = 255 \text{ и } m = 0$$

$$v = \text{NaN}, \text{ если } e = 255 \text{ и } m \neq 0$$

Точность 7 — 8 знаков.

Double

Формат:



Формула для вычисления значения, хранящегося в памяти:

$$v = (-1)^s \cdot 2^{e-1023} \cdot 1.m, \text{ если } 0 < e < 2047$$

$$v = (-1)^s \cdot 2^{1022} \cdot 0.m, \text{ если } e = 0 \text{ и } m \neq 0$$

$$v = (-1)^s, \text{ если } e = 0 \text{ и } m = 0$$

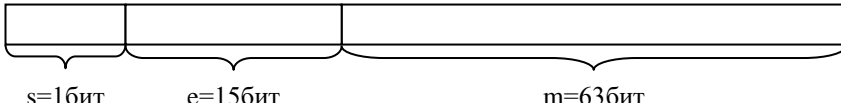
$$v = (-1)^s \cdot \text{Inf}, \text{ если } e = 2047 \text{ и } m = 0$$

$$v = \text{NaN}, \text{ если } e = 2047 \text{ и } m \neq 0$$

Точность 15 — 16 знаков.

Extended

Формат:



Формула для вычисления значения, хранящегося в памяти:

$$v = (-1)^s \cdot 2^{e-16383} \cdot 1.m, \text{ если } 0 < e < 32767$$

$$v = (-1)^s \cdot 2^{16382} \cdot 0.m, \text{ если } e = 0 \text{ и } m \neq 0$$

$$v = (-1)^s, \text{ если } e = 0 \text{ и } m = 0$$

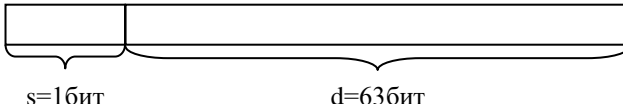
$$v = (-1)^s \cdot \text{Inf}, \text{ если } e = 32767 \text{ и } m = 0$$

$$v = \text{NaN}, \text{ если } e = 32767 \text{ и } m \neq 0$$

Точность 19 — 20 знаков.

Сложный тип*Comp*

Формат:



Значение $v = \text{NaN}$, если $s = 1$ и $d = 0$, иначе v представляет собой 64-битовое значение, являющееся дополнением до двух.

Простые типы данных в С**Целые типы**

Диапазон возможных значений целых типов зависит от их внутреннего представления. В таблице 5 приводится название основных целых типов, длина их внутреннего представления в байтах и диапазон возможных значений.

Таблица 5

Целые типы языка С

Имя типа	Диапазон значений	Размер, байт
Char	0...255	1
Int	-32768...32767	2 (или 4)

В языке C/C++ существует возможность использовать модификаторы *short* и *long*. Целью этих модификаторов было разграничить длины двух типов целых чисел для практических потребностей.

Модификаторы *signed* (знаковый) и *unsigned* (без знака) применимы к любым целым типам.

Диапазоны значений целых типов языка C приведены в таблице 6:

Таблица 6

Диапазоны значений целых типов языка C

Имя типа	Диапазон значений	Размер, байт
Long int	−2147483648...2147483647	4
Signed char	−128...127	1
Unsigned char	0...255	1
Short int	−32768...32767	2
Unsigned int	0...65535	2
Unsigned long int	0...4294967295	4
Unsigned short int	0...65535	2

Используя модификаторы *short*, *long*, *unsigned* со спецификатором *int*, спецификатор *int* может быть опущен, тогда модификаторы рассматриваются как спецификаторы:

Unsigned int = *unsigned*

Long int = *long*

Short int = *short*

Символьный тип

Диапазоном возможных значений этого типа является множество всех символов. Каждому символу приписывается целое число от 0..255. Для кодировки используют код *ASCII*. Символьный тип занимает в памяти один байт.

Перечисляемый тип

Он задается перечислением тех значений, которые он может получать. Каждое значение именуется некоторым идентификатором и располагается в списке, обрамленном фигурными скобками, например:

```
Typedef enum {red, yellow, green} colors;
```

Соответствие между значениями перечисляемого типа и порядковыми номерами этих значений устанавливается порядком перечисления: первое значение в списке получает порядковый номер 0, второе — 1 и т.д.

Вещественные типы

Значения этих типов определяют произвольное число лишь с некоторой конечной точностью, зависящей от внутреннего формата вещественно-

го числа. В таблице 7 приводятся названия вещественных типов, диапазон возможных значений и длина внутреннего представления.

Таблица 7

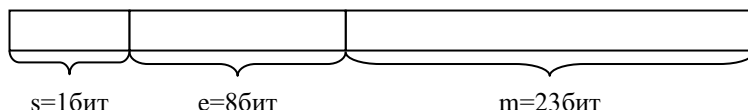
Вещественные типы языка C

Имя типа	Диапазон значений	Размер, байт
Float	1.1754943E-38...1.175494E+38	4
Double	2.22507E-308... 2.225E+308	8
Long double (в некоторых реализациях языка СИ отсутствует)	3.4E-4932...3.4E+4932	10

Внутримашинное представление вещественных типов:

Float

Формат:



Формула для вычисления значения, хранящегося в памяти:

$$v = (-1)^s \cdot 2^{e-127} \cdot 1.m, \text{ если } 0 < e < 255$$

$$v = (-1)^s \cdot 2^{126} \cdot 0.m, \text{ если } e = 0 \text{ и } m \neq 0$$

$$v = (-1)^s, \text{ если } e = 0 \text{ и } m = 0$$

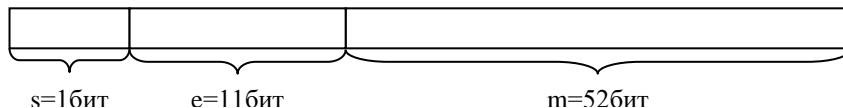
$$v = (-1)^s \cdot \text{Inf}, \text{ если } e = 255 \text{ и } m = 0$$

$$v = \text{NaN}, \text{ если } e = 255 \text{ и } m \neq 0$$

Точность 6 — 7 знаков.

Double

Формат:



Формула для вычисления значения, хранящегося в памяти:

$$v = (-1)^s \cdot 2^{e-1023} \cdot 1.m, \text{ если } 0 < e < 2047$$

$$v = (-1)^s \cdot 2^{1022} \cdot 0.m, \text{ если } e = 0 \text{ и } m \neq 0$$

$$v = (-1)^s, \text{ если } e = 0 \text{ и } m = 0$$

$$v = (-1)^s \cdot \text{Inf}, \text{ если } e = 2047 \text{ и } m = 0$$

$$v = \text{NaN}, \text{ если } e = 2047 \text{ и } m \neq 0$$

Точность 15-16 знаков.

Структурированные типы данных в Pascal

Массив

Массив — последовательность элементов одного типа, называемого *базовым*.

На *абстрактном* уровне массив представляет собой линейную структуру. На *физическом* уровне массив реализован последовательной (прямоугольной) схемой хранения. Располагаться он может в статической или динамической памяти. Размер памяти, выделяемый под массив, зависит от базового типа элемента массива и от количества элементов в массиве и определяется формулой $V_{\text{мас}} = V_{\text{эл}} \cdot k$; где $V_{\text{мас}}$ — объем памяти для массива, $V_{\text{эл}}$ — объем памяти одного элемента (слот), k — количество элементов.

На *логическом* уровне СД типа массив можно описать следующим образом:

1. Type $T_ar = \text{array } [T1] \text{ of } T2$; { $T1$ — тип индекса }

Var $Ar : T_ar$; { $T2$ — тип элемента }

Массив Ar типа T_ar располагается в статической памяти.

2. Type $TP_ar = ^T_ar$;

Var $P_ar : TP_ar$;

Массив типа T_ar будет располагаться в динамической памяти после обращения к процедуре $\text{new}(P_ar)$. Адрес массива запишется в переменную P_ar .

Массив — это статическая структура. В процессе выполнения программы количество элементов массива не изменяется. Доступ к элементу массива прямой, осуществляется через индекс элемента, например $Ar[i]$ или $P_ar^{\wedge}[i]$.

Кардинальное число для массива T_ar :

$$Car(T_ar) = [Car(T2)]^{Car(T1)}$$

Набор допустимых операций для СД типа массив:

1. Операция доступа (доступ к элементам массива — прямой; от размера структуры время выполнения операции не зависит).

2. Операция присваивания.

Структура данных типа «запись»

Запись — это множество элементов (полей), которые могут быть различных типов.

На *абстрактном* уровне запись представляет собой структуру множество — отношения между элементами отсутствуют.

На *физическом* уровне запись реализована последовательной схемой хранения. Располагаться она может в статической или в динамической памяти. Размер памяти, выделяемый под запись, зависит от типов полей и от их количества и определяется формулой $V_{\text{зап}} = \sum V_i \mid i=1, k$; где $V_{\text{зап}}$ — объем памяти для записи, k — количество полей, V_i — объем памяти для i -го поля. На *логическом* уровне СД типа запись можно записать следующим образом:

```
Type T_rec = Record
    S1: T1;
    S2: T2;
    .....
    Sn: Tn;
End;
Var Rec: T_rec;
```

Здесь: S_1, \dots, S_n — идентификаторы полей; T_1, \dots, T_n — типы полей; Rec — идентификатор записи; T_{rec} — тип записи.

Если D_{T1} — множество значений элементов типа T_1 , D_{T2} — множество значений элементов типа T_2 , \dots , D_{Tn} — множество значений элементов типа T_n , то $D_{T_{rec}}$ — множество значений элементов типа T_{rec} будет определяться с помощью прямого декартова произведения:

$$D_{T_{rec}} = D_{T1} \times D_{T2} \times \dots \times D_{Tn}.$$

Кардинальное число для записи T_{rec} :

$$Car(T_{rec}) = \prod Car(T_i) \mid i=1, n$$

Допустимые операции для СД типа запись аналогичны операциям для СД типа массив.

По характеру изменчивости запись — это статическая структура. Доступ к элементам записи прямой, осуществляется по имени поля.

Структура данных типа «множество»

Множество на *физическом* уровне — это массив битов, в котором каждый бит указывает, является ли элемент принадлежащим множеству или нет. Максимальное число элементов множества — 256, так что множество никогда не может занимать более 32 байтов.

Число байтов, занимаемых отдельным множеством, вычисляется как $ByteSize = (Max \div 8) - (Min \div 8) + 1$, где Min и Max — нижняя и верхняя граница базового типа этого множества.

Номер байта для конкретного элемента E вычисляется по формуле $Byte\ Number = (E \div 8) - (Min \div 8)$, а номер бита внутри этого байта — по формуле $BitNumber = E \bmod 8$.

На логическом уровне множество можно описать так:

$Type\ T_set = set\ of\ T; \{T — базовый тип множества\}$

Значениями типа множество являются все подмножества базового типа. Мощность множественного типа T_set (кардинальное число) равна $Car(T_set) = 2^{Car(T)}$. Число элементов базового типа ограничено и не должно превышать 256. Базовый тип является перечисляемым типом или поддиапазоном перечисляемого типа. Кроме того, объем памяти, занимаемый значением базового типа — один байт.

Допустимыми операциями для множества являются:

1. Пересечение (*).
2. Объединение (+).
3. Вычитание (–).
4. Операции отношения:
 - 4.1. Равенство множеств (=).
 - 4.2. Неравенство множеств (\neq).
 - 4.3. Левый операнд-подмножество правого (\subseteq).
 - 4.4. Правый операнд-подмножество левого (\supseteq).
 - 4.5. Принадлежность элемента множеству (in).

Структурированные типы данных в С

Структура данных типа «массив»

Массив — последовательность элементов одного типа, называемого *базовым*. На *абстрактном* уровне массив представляет собой линейную структуру.

На *физическом* уровне массив реализован последовательной (прямоугольной) схемой хранения. Располагаться он может в статической или динамической памяти. Размер памяти, выделяемый под массив, зависит от базового типа элемента массива и от количества элементов в массиве и определяется формулой $V_{\text{мас}} = V_{\text{эл}} * k$, где $V_{\text{мас}}$ — объем памяти для массива, $V_{\text{эл}}$ — объем памяти одного элемента (слот), k — количество элементов.

На логическом уровне СД типа массив можно описать следующим образом:

1. *typedef T2 t_arr[T1]*, где $T1$ — тип индекса, $T2$ — тип элемента.
t_arr ar;

Массив *ar* типа *t_arr* располагается в статической памяти.

```
2. typedef t_arr *tp_ar;
   tp_ar p_ar;
```

Массив типа *tp_ar* будет располагаться в динамической памяти после обращения к одной из функций выделения памяти (*calloc()*, *malloc()*).

Адрес массива запишется в переменную *p_ar*.

Массив — это статическая структура. В процессе выполнения программы количество элементов массива не изменяется. Доступ к элементу массива прямой, осуществляется через индекс элемента, например *Ar[i]* или **P_ar[i]*.

Кардинальное число для массива *T_ar*:

$$Car(T_ar) = [Car(T2)]^{Car(T1)}.$$

Набор допустимых операций для СД типа массив:

1. Операция доступа (доступ к элементам массива — прямой, от размера структуры время выполнения операции не зависит).

2. Операция присваивания.

Структура данных типа «структура»

Структура — это множество элементов (полей), которые могут быть различных типов (аналогом в языке Pascal является запись).

На *абстрактном* уровне структура представляет собой множество — отношения между элементами отсутствуют.

На *физическом* уровне структура реализована последовательной схемой хранения. Располагаться она может в статической или в динамической памяти. Размер памяти, выделяемый под структуру, зависит от типов полей и от их количества и определяется формулой $V_{\text{зап}} = \sum V_i \mid i=1, k$, где $V_{\text{зап}}$ — объем памяти для структуры, k — количество полей, V_i — объем памяти для i -го поля.

На *логическом* уровне СД типа структура можно записать следующим образом:

```
typedef struct t_struct {S1: T1;
S2: T2;
.....
Sn: Tn;
};
t_struct str;
```

Здесь: S_1, \dots, S_n — идентификаторы полей; T_1, \dots, T_n — типы полей; *str* — идентификатор записи; *t_struct* — тип записи.

Если D_{T_1} — множество значений элементов типа T_1 , D_{T_2} — множество значений элементов типа T_2 , ..., D_{T_n} — множество значений элементов типа T_n , то D_{t_str} — множество значений элементов типа t_str будет определяться с помощью прямого декартова произведения:

$$D_{t_str} = D_{T_1} \cdot D_{T_2} \cdot \dots \cdot D_{T_n}.$$

Кардинальное число для структуры t_str :

$$Car(t_str) = \prod Car(T_i) \mid i=1, n.$$

Допустимые операции для СД типа структура аналогичны операциям для СД типа массив. По характеру изменчивости структура — это статическая структура данных. Доступ к элементам структуры прямой, осуществляется по имени поля.

Контрольные вопросы

1. Что такое структура данных?
2. Приведите примеры различных уровней описания структур данных.
3. Приведите примеры структур данных с различным характером организации.
4. Какие структуры данных называют динамическими, а какие — статическими?
5. Чем различаются последовательная и связанная схемы хранения данных.
6. От чего зависит диапазон значений целых типов.
7. Приведите примеры целых типов, имеющих различный диапазон значений и одинаковый объем памяти.
8. Чем определяется точность представления вещественных значений?
9. Приведите примеры форматов машинного представления вещественных значений.
10. Как определяется объем памяти, занимаемый множеством?
11. Сколько памяти занимает пустое множество?
12. Определите характер изменчивости массива.
13. Чем различаются структуры данных массив и запись на абстрактном уровне?
14. Как осуществляется доступ к элементам массива и элементам записи?
15. Определите множество значений структурированного типа данных.

Лабораторная работа № 2

Производные структуры данных.

Структура данных типа «строка» (Pascal/C)

Цель работы: изучение встроенной структуры данных типа «строка», разработка и использование производных структур данных строкового типа.

Задание

1. Для СД типа строка определить:
 - 1.1. Абстрактный уровень представления СД:
 - 1.1.1 Характер организованности и изменчивости.
 - 1.1.2 Набор допустимых операций.
 - 1.2. Физический уровень представления СД:
 - 1.2.1. Схему хранения.
 - 1.2.2. Объем памяти, занимаемый экземпляром СД.
 - 1.2.3. Формат внутреннего представления СД и способ его интерпретации.
 - 1.2.4. Характеристику допустимых значений.
 - 1.2.5. Тип доступа к элементам.
 - 1.3. Логический уровень представления СД.
 Способ описания СД и экземпляра СД на языке программирования.
2. Реализовать СД строкового типа в соответствии с вариантом индивидуального задания (см. табл.8) в виде модуля. Определить и обработать исключительные ситуации.
3. Разработать программу для решения задачи в соответствии с вариантом индивидуального задания (см. табл.8) с использованием модуля, полученного в результате выполнения пункта 2.

Таблица 8

Варианты индивидуальных заданий

Номер варианта	Номер формата	Задача
1	1	1
2	2	2
3	3	3

Окончание табл.8

4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9
10	1	10
11	2	11
12	3	12
13	4	13
14	5	14
15	6	15
16	7	1
17	8	2
18	9	3
19	1	4
20	2	5
21	3	6
22	4	7
23	5	8
24	6	9
25	7	10
26	8	11
27	9	12
28	1	13
29	2	14
30	3	15

Варианты задач

1. Заголовок: *procedure Copies(var s1,s2:string; n:byte)/ void Copies(string1 s1, string1 s2, int n).*
Назначение: копирование строки *s* в строку *s1* *n* раз.
Входные параметры: *s1,n*.
Выходные параметры: *s2*.
2. Заголовок: *function Word(s:string):word/ unsigned Word(string1 s).*
Назначение: подсчет числа слов в строке *s*.
Входные параметры: *s*.
Выходные параметры: нет.

3. Заголовок: *procedure Center(s1,s2:string;l:word)/ void Center(string1 s1, string1 s2, unsigned l).*

Назначение: центрирование расположения строки *s1* в середине строки *s2* длины *l*.

Входные параметры: *s1, l*.

Выходные параметры: *s2*.

4. Заголовок: *function LastPost(s1,s2:string):word/ unsigned LastPost(string1 s1, string1 s2).*

Назначение: поиск последнего вхождения подстроки *s2* в строку *s1*.

Входные параметры: *s1, s2*.

Выходные параметры: нет.

5. Заголовок: *function WordLength(s:string;n:word):word/ unsigned WordLength(string1 s, unsigned n).*

Назначение: определение длины слова с номером *n*.

Входные параметры: *s, n*.

Выходные параметры: нет.

6. Заголовок: *function WordIndex(s:string;n:word):word/ unsigned WordIndex(string1 s, unsigned n).*

Назначение: возврат позиции начала в строке *s* слова с номером *n*.

Входные параметры: *s, n*.

Выходные параметры: нет.

7. Заголовок: *function SudWord(s:string;n:word):string/ string1 *SudWord(char *s, unsigned n).*

Назначение: выделение из строки *s* слов, начиная с номера *n*.

Входные параметры: *s, n*.

Выходные параметры: нет.

8. Заголовок: *function WordCmp(s1,s2:string):boolean/ int WordCmp(string1 s1, string1 s2).*

Назначение: сравнение строк(с игнорированием множественных пробелов).

Входные параметры: *s1, s2*.

Выходные параметры: нет.

9. Заголовок: *function StrSpn(s1,s2:string):word/ unsigned StrSpn(string1 s1, string1 s2).*

Назначение: нахождение длины той части строки *s1*, которая содержит только символы из строки *s2*.

Входные параметры: $s1, s2$.

Выходные параметры: нет.

10. Заголовок: *function StrCSpn(s,s1:string):word/ unsigned StrCSpn(string1 s, string1 s1).*

Назначение: нахождение длины той части строки s , которая не содержит символы из строки $s1$.

Входные параметры: $s, s1$.

Выходные параметры: нет.

11. Заголовок: *procedure Overlay(var s,s1:string;n:word)/ void Overlay(string1 s, string1 s1, unsigned n).*

Назначение: перекрытие части строки s , начиная с позиции n , строкой $s1$.

Входные параметры: $s, s1, n$.

Выходные параметры: s .

12. Заголовок: *procedure Compress(var s:string;c:char)/ void Compress(string1 s, char c).*

Назначение: замена в строке s множественных вхождений символа c на единственное.

Входные параметры: s, c .

Выходные параметры: s .

13. Заголовок: *procedure Trail(var s:string)/ void Trail(string1 s).*

Назначение: удаление головных и хвостовых пробелов.

Входные параметры: s .

Выходные параметры: s .

14. Заголовок: *procedure SrtSet(var s:string;n,l:word;c:char)/ void SrtSet(string1 s, unsigned n, unsigned l, char c).*

Назначение: установка l символов строки s , начиная с позиции n , в значение c .

Входные параметры: s, c, l, n .

Выходные параметры: s .

15. Заголовок: *procedure Space(s:string;l:word)/ void Space(string1 s, unsigned l).*

Назначение: доведение строки s до длины l путем вставки пробелов между словами.

Входные параметры: s, l .

Выходные параметры: s .

Варианты форматов

Формат 1

Реализация на языке Pascal:

```
Unit form1;
Interface
  Const {определение исключительных ситуаций}
  Type string1=array[1..256] of char;
  {признак конца строки—символ с кодом 0}
  Procedure WriteToStr(var st:string1;s:string);
  Procedure WriteFromStr(var s:string;st:string1);
  Procedure InputStr(var st:string1);
  Procedure OutputStr(const st:string1);
  Function Comp(s1,s2:string1;var fl:shortint):boolean;
  Procedure Delete(var S:String1;Index,Count:Word);
  Procedure Insert(Subs:String1;var S:String1;Index:Word);
  Procedure Concat( const S1, S2:string1;var srez: string1);
  Procedure Copy(S:String1;Index,Count:Word; var Subs:string1);
  Function Length(S: String1): word;
  Function Pos(SubS, S: String1): word;
  Var StrError: {тип переменной ошибки}
```

Реализация на языке C:

```
#if !defined(__FORM1_H)
#define __FORM1_H
const ...; // Определение исключительных ситуаций
typedef char string1[256];
// Признак конца строки - символ '\0'
void WriteToStr(string1 st, char *s);
void WriteFromStr(char *s, string1 st);
void InputStr(string1 s);
void OutputStr(string1 s);
int Comp(string1 s1, string1 s2);
void Delete(string1 s, unsigned Index, unsigned Count);
void Insert(string1 Subs, string1 s, unsigned Index);
void Concat(string1 s1, string1 s2, string1 srez);
void Copy(string1 s, unsigned Index, unsigned Count, string1 Subs);
unsigned Length(string1 s);
unsigned Pos(string1 SubS, string1 s);
```

```

    int StrError; // Переменная ошибок
    //...
#endif

```

Формат 2

Реализация на языке *Pascal*:

```

Unit form2;
Interface
    Const {определение исключительных ситуаций}
    Type
        str=array[1..65520] of char; {признак конца строки – символ с кодом
0}
        p_str=^str;
        string1=record
            st:p_str;
            n:word {количество символов в строке, определяется при
инициализации}
        end;

```

```

    Procedure InitStr(var st:string1; n:word);
    Procedure WriteToStr(var st:string1;s:string);
    Procedure WriteFromStr(var s:string;st:string1);
    Procedure InputStr(var st:string1);
    Procedure OutputStr(const st:string1);
    Function Comp(s1,s2:string1;var fl:shortint):boolean;
    Procedure Delete(var S:String1;Index,Count:Word);
    Procedure Insert(Subs:String1;var S:String1;Index:Word);
    Procedure Concat( const S1, S2:string1;var srez:string1);
    Procedure Copy(S:String1;Index,Count:Word; var Subs:string1);
    Function Length(S: String1): word;
    Function Pos(SubS, S: String1): word;
    Var StrError: {тип переменной ошибки}

```

Реализация на языке *C*:

```

#ifndef __FORM2_H
#define __FORM2_H
    const ...; // Определение исключительных ситуаций
    typedef struct str
    {

```

```

char* s; // Признак конца строки - символ '\0'
unsigned max; /* Максимальное количество символов в стро-
ке, определяющееся при инициализации */
};
typedef str *string1;
void InitStr(string1 st, unsigned n);
void WriteToStr(string1 st, char *s);
void WriteFromStr(char *s, string1 st);
void InputStr(string1 st);
void OutputStr(string1 st);
int Comp(string1 s1, string1 s2);
void Delete(string1 s, unsigned Index, unsigned Count);
void Insert(string1 Subs, string1 s, unsigned Index);
void Concat(string1 s1, string1 s2, string1 srez);
void Copy(string1 s, unsigned Index, unsigned Count, string1 Subs);
unsigned Length(string1 s);
unsigned Pos(string1 SubS, string1 s);
void DoneStr(string1 s)
int StrError; // Переменная ошибок
//...
#endif

```

Формат 3

Реализация на языке Pascal:

```

Unit form3;
Interface
Const {определение исключительных ситуаций}
Type string1=array[-1..1024] of char;
{первые два байта содержат динамическую длину строки}
Procedure WriteToStr(var st:string1;s:string);
Procedure WriteFromStr(var s:string;st:string1);
Procedure InputStr(var st:string1);
Procedure OutputStr(const st:string1);
Function Comp(s1,s2:string1;var fl:shortint):boolean;
Procedure Delete(var S:String1;Index,Count:Word);
Procedure Insert(Subs:String1;var S:String1;Index:Word);
Procedure Concat( const S1, S2:string1;var srez:string1);
Procedure Copy(S:String1;Index,Count:Word; var Subs:string1);
Function Length(S: String1): word;
Function Pos(SubS, S: String1): word;
Var StrError: {тип переменной ошибки}

```

Реализация на языке C:

```

#if !defined(__FORM3_H)
#define __FORM3_H
    const ...; // Определение исключительных ситуаций
    typedef char string1[1024]; /* Первые два байта содержат динамиче-
    скую длину строки */
    void WriteToStr(string1 st, char *s);
    void WriteFromStr(char *s, string1 st);
    void InputStr(string1 st);
    void OutputStr(string1 st);
    int Comp(string1 s1, string1 s2);
    void Delete(string1 s, unsigned Index, unsigned Count);
    void Insert(string1 Subs, string1 s, unsigned Index);
    void Concat(string1 s1, string1 s2, string1 srez);
    void Copy(string1 s, unsigned Index, unsigned Count, string1 Subs);
    unsigned Length(string1 s);
    unsigned Pos(string1 SubS, string1 s);
    int StrError; // Переменная ошибок
    //...
#endif

```

Формат 4**Реализация на языке Pascal:**

```

Unit form4;
Interface
    Const {определение исключительных ситуаций}
    Type string1=record
        St:array[1..1024] of char;
        N:word{динамическая длина строки}
    End;
    Procedure WriteToStr(var st:string1;s:string);
    Procedure WriteFromStr(var s:string;st:string1);
    Procedure InputStr(var st:string1);
    Procedure OutputStr(const st:string1);
    Function Comp(s1,s2:string1;var fl:shortint):boolean;
    Procedure Delete(var S:String1;Index,Count:word);
    Procedure Insert(Subs:String1;var S:String1;Index:word);
    Procedure Concat( const S1, S2:string1;var srez:string1);
    Procedure Copy(S:String1;Index,Count:Word; var Subs:string1);
    Function Length(S: String1): word;
    Function Pos(SubS, S: String1): word;
    Var StrError: {тип переменной ошибки}

```

Реализация на языке C:

```

#if !defined(__FORM4_H)
#define __FORM4_H
    const ...; // Определение исключительных ситуаций
    typedef struct str
    {
        char s[1024];
        unsigned N; // Динамическая длина строки
    };
    typedef str *string1;
    void WriteToStr(string1 st, char *s);
    void WriteFromStr(char *s, string1 st);
    void InputStr(string1 st);
    void OutputStr(string1 st);
    int Comp(string1 s1, string1 s2);
    void Delete(string1 s, unsigned Index, unsigned Count);
    void Insert(string1 Subs, string1 s, unsigned Index);
    void Concat(string1 s1, string1 s2, string1 srez);
    void Copy(string1 s, unsigned Index, unsigned Count, string1 Subs);
    unsigned Length(string1 s);
    unsigned Pos(string1 SubS, string1 s);
    int StrError; // Переменная ошибок
    //...
#endif

```

Формат 5**Реализация на языке Pascal:**

```

Unit form5;
Interface
    Const {определение исключительных ситуаций}
    Type St=array[1..65520] of char;
         String1=record
             p_st:^st; {указатель на строку}
             max:word; {максимальное количество символов в стро-
ке, определяется при инициализации}
             N:word {динамическая длина строки}
         End;

    Procedure InitStr(var st:string1; n:word);

```



```

Procedure WriteToStr(var st:string1;s:string);
Procedure WriteFromStr(var s:string;st:string1);
Procedure InputStr(var st:string1);
Procedure OutputStr(const st:string1);
Function Comp(s1,s2:string1;var fl:shortint):boolean;
Procedure Delete(var S:String1;Index,Count:word);
Procedure Insert(Subs:String1;var S:String1;Index:word);
Procedure Concat( const S1, S2:string1;var srez:string1);
Procedure Copy(S:String1;Index,Count:Word; var Subs:string1);
Function Length(S: String1): word;
Function Pos(SubS, S: String1): word;
Var StrError: {тип переменной ошибки}

```

Реализация на языке C:

```

#ifndef __FORM5_H
#define __FORM5_H
const ...; // Определение исключительных ситуаций
typedef struct str
{
    char *s; // Указатель на строку
    unsigned max; /* Максимальное количество символов в
строке, определяющееся при инициализации */
    unsigned N; // Динамическая (текущая) длина строки
};
typedef str *string1;
void InitStr(string1 st, unsigned n);
void WriteToStr(string1 st, char *s);
void WriteFromStr(char *s, string1 st);
void InputStr(string1 st);
void OutputStr(string1 st);
int Comp(string1 s1, string1 s2);
void Delete(string1 s, unsigned Index, unsigned Count);
void Insert(string1 Subs, string1 s, unsigned Index);
void Concat(string1 s1, string1 s2, string1 srez);
void Copy(string1 s, unsigned Index, unsigned Count, string1 Subs);
unsigned Length(string1 s);
unsigned Pos(string1 SubS, string1 s);
void DoneStr(string1 s)
int StrError; // Переменная ошибок
//...
#endif

```

Формат 6**Реализация на языке Pascal:***Unit form6;**Interface**Const* {определение исключительных ситуаций}*Type St=array[1..65520] of char;* {первые два байта содержат максимальную длину строки, которая определяется при инициализации}*String1=record**p_st:^st;*{указатель на строку}*N:word* {динамическая длина строки}*End;**Procedure InitStr*(var st:string1; n:word);*Procedure WriteToStr*(var st:string1;s:string);*Procedure WriteFromStr*(var s:string;st:string1);*Procedure InputStr*(var st:string1);*Procedure OutputStr*(const st:string1);*Function Comp*(s1,s2:string1;var fl:shortint):boolean;*Procedure Delete*(var S:String1;Index,Count:word);*Procedure Insert*(Subs:String1;var S:String1;Index:word);*Procedure Concat*(const S1, S2:string1;var srez:string1);*Procedure Copy*(S:String1;Index,Count:Word; var Subs:string1);*Function Length*(S: String1): word;*Function Pos*(SubS, S: String1): word;*Var StrError:* {тип переменной ошибки}**Реализация на языке C:**

#if !defined(__FORM6_H)

#define __FORM6_H

const ...; // Определение исключительных ситуаций

typedef struct str

{

*char *s;* /* Указатель на строку. Первые два байта строки s содержат максимальную длину строки, которая определяется при инициализации */*unsigned N;* // Динамическая (текущая) длина строки

};

typedef str *string1;

void InitStr(string1 st, unsigned n);

void WriteToStr(string1 st, char *s);

```

void WriteFromStr(char *s, string1 st);
void InputStr(string1 st);
void OutputStr(string1 st);
    int Comp(string1 s1, string1 s2);
    void Delete(string1 s, unsigned Index, unsigned Count);
    void Insert(string1 Subs, string1 s, unsigned Index);
    void Concat(string1 s1, string1 s2, string1 srez);
    void Copy(string1 s, unsigned Index, unsigned Count, string1 Subs);
    unsigned Length(string1 s);
    unsigned Pos(string1 SubS, string1 s);
    void DoneStr(string1 s);
    int StrError; // Переменная ошибок //...
#endif

```

Формат 7

Реализация на языке Pascal:

```

Unit form7;
Interface
    Const {определение исключительных ситуаций}
    Type St=array[1..65520] of char;
    {первые два байта содержат динамическую длину строки}
    String1=record
        p_st:^st;{указатель на строку}
        max:word {максимальная длина строки, определяется
при инициализации}
    End;

    Procedure InitStr(var st:string1; n:word);
    Procedure WriteToStr(var st:string1;s:string);
    Procedure WriteFromStr(var s:string;st:string1);
    Procedure InputStr(var st:string1);
    Procedure OutputStr(const st:string1);
    Function Comp(s1,s2:string1;var fl:shortint):boolean;
    Procedure Delete(var S:String1;Index,Count:word);
    Procedure Insert(Subs:String1;var S:String1;Index:word);
    Procedure Concat( const S1, S2:string1;var srez:string1);
    Procedure Copy(S:String1;Index,Count:Word; var Subs:string1);
    Function Length(S: String1): word;
    Function Pos(SubS, S: String1): word;
    Var StrError: {тип переменной ошибки}

```

Реализация на языке C:

```

#if !defined(__FORM7_H)
#define __FORM7_H
    const ...; // Определение исключительных ситуаций
    typedef struct str
    {
        char *s; /* Указатель на строку. Первые два байта строки s
        содержат динамическую длину строки */
        unsigned max; /* Максимальное количество символов в стро-
        ке, определяющееся при инициализации */
    };
    typedef str *string1;
    void InitStr(string1 st, unsigned n);
    void WriteToStr(string1 st, char *s);
    void WriteFromStr(char *s, string1 st);
    void InputStr(string1 st);
    void OutputStr(string1 st);
    int Comp(string1 s1, string1 s2);
    void Delete(string1 s, unsigned Index, unsigned Count);
    void Insert(string1 Subs, string1 s, unsigned Index);
    void Concat(string1 s1, string1 s2, string1 srez);
    void Copy(string1 s, unsigned Index, unsigned Count, string1 Subs);
    unsigned Length(string1 s);
    unsigned Pos(string1 SubS, string1 s);
    void DoneStr(string1 s);
    int StrError; // Переменная ошибок//...
#endif

```

Формат 8**Реализация на языке Pascal:**

```

Unit form8;
Interface
    Const {определение исключительных ситуаций}
    Type St=array[1..65520] of char; {первые два байта содержат макси-
    мальную длину строки, которая определяется при инициализации; признак
    конца строки – символ с кодом 0}
    String1=^st; {указатель на строку}

    Procedure InitStr(var st:string1; n: word);

```

```

Procedure WriteToStr(var st:string1;s:string);
Procedure WriteFromStr(var s:string;st:string1);
Procedure InputStr(var st:string1);
Procedure OutputStr(const st:string1);
Function Comp(s1,s2:string1;var fl:shortint):boolean;
Procedure Delete(var S:String1;Index,Count:word);
Procedure Insert(Subs:String1;var S:String1;Index:word);
Procedure Concat( const S1, S2:string1;var srez:string1);
Procedure Copy(S:String1;Index,Count:Word; var Subs:string1);
Function Length(S: String1): word;
Function Pos(SubS, S: String1): word;
Var StrError: {тип переменной ошибки}

```

Реализация на языке C:

```

#ifdef __FORM8_H
#define __FORM8_H
const ...; // Определение исключительных ситуаций
typedef char *string1;
/* Первые два байта содержат максимальную длину строки, которая
определяется при инициализации. Признак конца строки - символ '\0' */
void InitStr(string1 *st, unsigned n);
void WriteToStr(string1 st, char *s);
void WriteFromStr(char *s, string1 st);
void InputStr(string1 st);
void OutputStr(string1 st);
int Comp(string1 s1, string1 s2);
void Delete(string1 s, unsigned Index, unsigned Count);
void Insert(string1 Subs, string1 s, unsigned Index);
void Concat(string1 s1, string1 s2, string1 srez);
void Copy(string1 s, unsigned Index, unsigned Count,string1 Subs);
unsigned Length(string1 s);
unsigned Pos(string1 SubS, string1 s);
void DoneStr(string1 s);
int StrError; // Переменная ошибок //...
#endif

```

Формат 9

Реализация на языке Pascal:

```

Unit form9;
Interface

```

```

Const {определение исключительных ситуаций}
Type St=array[1..65520] of char;
  {первые два байта содержат максимальную длину строки, которая
определяется при инициализации; вторые два байта содержат динамиче-
скую длину строки}
String1=^st; {указатель на строку}
Procedure InitStr(var st:string1; n:word);
Procedure WriteToStr(var st:string1;s:string);
Procedure WriteFromStr(var s:string;st:string1);
Procedure InputStr(var st:string1);
Procedure OutputStr(const st:string1);
Function Comp(s1,s2:string1;var fl:shortint):boolean;
Procedure Delete(var S:String1;Index,Count:word);
Procedure Insert(Subs:String1;var S:String1;Index:word);
Procedure Concat( const S1, S2:string1;var srez:string1);
Procedure Copy(S:String1;Index,Count:Word; var Subs:string1);
Function Length(S: String1): word;
Function Pos(SubS, S: String1): word.
Function Pos(SubS, S: String1): word;
Var StrError: {тип переменной ошибки}

```

Реализация на языке C:

```

#if !defined(__FORM9_H)
#define __FORM9_H
const ...; // Определение исключительных ситуаций
typedef char *string1;

/* Первые два байта содержат максимальную длину строки, которая
определяется при инициализации. Вторые два байта содержат динамиче-
скую длину строки */

void InitStr(string1 *st, unsigned n);
void WriteToStr(string1 st, char *s);
void WriteFromStr(char *s, string1 st);
void InputStr(string1 st);
void OutputStr(string1 st);
int Comp(string1 s1, string1 s2);
void Delete(string1 s, unsigned Index, unsigned Count);
void Insert(string1 Subs, string1 s, unsigned Index);
void Concat(string1 s1, string1 s2, string1 srez);
void Copy(string1 s, unsigned Index, unsigned Count, string1 Subs);

```

```

unsigned Length(string1 s);
unsigned Pos(string1 SubS, string1 s);
void DoneStr(string1 s);
int StrError; // Переменная ошибок
//...
#endif

```

Назначение процедур и функций в модулях реализации СД типа строка в Pascal

1. *Procedure InputStr*(var st:string1). Ввод строки *st* с клавиатуры.
2. *Procedure OutputStr*(const st:string1). Вывод строки *st* на экран монитора.
3. *Procedure InitStr*(var st:string1; n: word). Выделение динамической памяти под строку *st*, содержащую от 0 до *n* символов.
4. *Procedure WriteToStr*(var st:string1; s:string). Запись данных в строку *st* из строки *s*.
5. *Procedure WriteFromStr*(var s:string; st:string1). Запись данных в строку *s* из строки *st*.
6. *Function Comp*(s1,s2:string1; var fl:shortint):boolean. Сравнивает строки *s1* и *s2*. Возвращает true если *s1=s2* и *fl=0*, если *s1>s2* и *fl=1*, если *s1<s2* и *fl=-1*.
7. *Procedure Delete*(var S:String1; Index,Count:Word). Удаляет *Count* символов из строки *S*, начиная с позиции *Index*.
8. *Procedure Insert*(Subs:String1;var S:String1; Index:Word). Вставляет подстроку *SubS* в строку *S*, начиная с позиции *Index*.
9. *Procedure Concat*(const S1, S2:string1; var srez:string1). Выполняет конкатенацию строк *S1* и *S2*; результат помещает в *srez*.
10. *Procedure Copy* (S:String1;Index,Count:Word; var Subs: String1). Возвращает подстроку *Subs* из строки *S*, начиная с позиции *Index* и длиной *Count* символов.
11. *Function Length*(S: String1): Word. Возвращает текущую длину строки *S*.
12. *Function Pos*(SubS, S: String1): Word. Возвращает позицию, начиная с которой в строке *S* располагается подстрока *SubS*.

Назначение процедур и функций в модулях реализации СД типа «строка» в C

1. *void InputStr*(string1 s). Ввод строки *s* с клавиатуры.
2. *void OutputStr*(string1 s). Вывод строки *s* на экран монитора.
3. *void InitStr*(string1 *s, unsigned n). Выделение динамической памяти под строку *st*, содержащую от 0 до *n* символов. Значением *n* определяется

максимальное количество символов, которое может вместить строка (зависит от кол-ва выделенной памяти). Динамическая длина строки есть ее текущая длина.

4. *void WriteToStr(string1 st, char *s)*. Запись данных в строку *st* из строки *s*. Строка *s* заканчивается нулевым символом '\0'.

5. *void WriteFromStr(char *s, string1 st)*. Запись данных в строку *s* из строки *st*. Строка *s* заканчивается нулевым символом '\0'.

6. *int Comp(string1 s1, string1 s2)*. Сравнивает строки *s1* и *s2*. Возвращает 0 если $s1 = s2$; 1, если $s1 > s2$; -1, если $s1 < s2$.

7. *void Delete(string1 s, unsigned Index, unsigned Count)*. Удаляет *Count* символов из строки *s*, начиная с позиции *Index*.

8. *void Insert(string1 Subs, string1 s, unsigned Index)*. Вставляет подстроку *Subs* в строку *s*, начиная с позиции *Index*.

9. *void Concat(string1 s1, string1 s2, string1 srez)*. Выполняет конкатенацию строк *s1* и *s2*. Результат помещает в *srez*.

10. *void Copy(string1 s, unsigned Index, unsigned Count, string1 Subs)*. Записывает *Count* символов в строку *Subs* из строки *s*, начиная с позиции *Index*.

11. *unsigned Length(string1 s)*. Возвращает текущую длину строки *S*.

12. *unsigned Pos(string1 SubS, string1 s)*. Возвращает позицию, начиная с которой в строке *s* располагается подстрока *SubS*.

void DoneStr(string1 s). Удаляет строку *s* из динамической памяти.

Содержание отчета

1. Тема лабораторной работы.
2. Цель работы.
3. Характеристика СД «строка» (пункт 1 задания).
4. Индивидуальное задание.
5. Текст модуля для реализации СД типа «строка», текст программы для отладки модуля, тестовые данные, результат работы программы.
6. Текст программы для решения задачи с использованием модуля, тестовые данные, результат работы программы.

Теоретические сведения

Строкой называется последовательность символов.

На *абстрактном* уровне строка представляет собой линейную структуру — последовательность.

На *физическом* уровне строка реализована последовательной схемой хранения. Располагаться она может в статической или динамической памяти. Размер памяти, выделяемый под строку, зависит от максимального количества элементов (символов) в строке и определяется формулой:

$V_{стр} = K + 1$, где K — максимальное количество символов в строке.

Строка — это динамическая структура. В процессе выполнения программы количество элементов может изменяться от нуля до K , но размер памяти, выделенный под строку, не меняется. Практически, строка представляет собой массив символов из $K + 1$ элементов типа *char*. Нумеруются элементы от нуля до K . В Pascal в нулевом элементе хранится символ с кодом, равным динамической длине строки, т.е. количеству элементов в строке в текущий момент времени. Если там находится символ с кодом 0, то строка не содержит ни одного символа и называется пустой. Элементы, начиная с первого содержат символы строки, количество которых совпадает с динамической длиной строки. В C во внутреннем представлении этот массив (строка) заканчивается нулевым символом '\0', по которому программа может найти конец, т.е. элемент с индексом K равен нулю как признак конца строки. Доступ к элементам строки, также как и к элементам массива — прямой.

На *логическом* уровне СД типа строка может быть описана следующим образом.

В языке Pascal:

1. *Var s1: string;*

Экземпляр СД типа строка (*string*) *s1* располагается в статической памяти и занимает 256 байт, количество элементов строки может быть в пределах от 0 до 255.

2. *Type T_str10 = string[10];*

Var s2: T_str10;

Экземпляр СД типа строка (*T_str10*) *s2* располагается в статической памяти и занимает 11 байт, количество элементов строки может быть в пределах от 0 до 10.

3. *Type P_str = ^string;*

Var ps3: P_str;

Строка будет располагаться в динамической памяти после обращения к процедуре *new(ps3)*. Строка будет занимать в памяти 256 байт, адрес которой запишется в переменную *ps3*. Количество символов в строке может быть в пределах от 0 до 255.

```
Type T_str10 = string[10];
P_str10 = ^T_str10;
Var ps4: P_str10;
```

Строка будет располагаться в динамической памяти после обращения к процедуре *new(ps4)*. Строка будет занимать в памяти 11 байт, адрес которой запишется в переменную *ps4*. Количество символов в строке может быть в пределах от 0 до 10.

В языке C:

1. *#define STRLENGTH ...// Значение*

...

char s[STRLENGTH];

Экземпляр СД типа строка располагается в статической памяти и занимает *STRLENGTH* байт, количество элементов строки (символов) может быть в пределах от 0 до *STRLENGTH - 1*.

2. *typedef char t_str10[10];*

...

t_str10 s;

Экземпляр СД типа строка (*t_str10*) *s* располагается в статической памяти и занимает 10 байт, количество элементов строки может быть в пределах от 0 до 9.

3. *typedef char* p_str;*

...

p_str ps;

Строка будет располагаться в динамической памяти после обращения к процедуре выделения памяти (*malloc* или *calloc*). Адрес начала строки запишется в переменную *ps*.

4. *typedef char t_str10[10];*

typedef t_str10 p_str10;*

...

p_str10 ps;

Строка будет располагаться в динамической памяти после обращения:

ps = (p_str10) new p_str10;

Строка будет занимать в памяти 10 байт, адрес которой запишется в переменную *ps*. Количество символов в строке может быть в пределах от 0 до 9.

Количество допустимых значений СД типа «строка»:

$$CAR(string) = 1 + 256 + 256^2 + \dots + 256^K,$$

где K — максимальное количество элементов в строке.

Операции над строками:

1. Операция присваивания.

Операндами могут быть символы, строки, символьные массивы. Результатом операции является строка, равная значению операнда. Если строковой переменной присваивается значение, превышающее ее длину, то перед присваиванием происходит усечение присваиваемого значения.

2. Операция сравнения.

Переменные строкового типа можно сравнивать между собой. Из двух строк является та большей, у которой первый из неравных символов больше (по ASCII-коду). Иначе они равны.

3. Операция конкатенация.

Операндами могут быть символы, строки, символьные массивы. Результатом является строка, полученная дописыванием в конец первого операнда второго операнда.

В Pascal определены следующие стандартные процедуры и функции над строками:

1. *Procedure Delete(var S:String; Index,Count:Integer)*

Удаляет *Count* символов из строки *S*, начиная с позиции *Index*.

2. *Procedure Insert(Subs:String;var S:String; Index:Integer)*

Вставляет подстроку *SubS* в строку *S*, начиная с позиции *Index*.

3. *Procedure Str(X:real; var S:String)*

Преобразует численное значение *X* в его строковое представление *S*.

4. *Procedure Val(S:String; var X;var Code:Integer)*

Преобразует строковое значение *S* в его численное представление *X*. Параметр *Code* содержит признак ошибки преобразования (0 — нет ошибки).

5. *Function Concat(S1[,S2,...,SN]:string):string*

Выполняет конкатенацию последовательности строк.

6. *Function Copy(S: String; Index, Count: Integer):string*

Возвращает подстроку из строки *S*, начиная с позиции *Index* и длиной *Count* символов.

7. *Function Length(S: String): byte*

Возвращает текущую длину строки *S*.

8. *Function Pos(SubS, S: String): Byte*

Возвращает позицию, начиная с которой в строке *S* располагается подстрока *SubS* (0 — *S* не содержит *SubS*).

Операций над строками как с едиными целыми в языке С нет. Такие возможности, как конкатенация, сравнение строк, являющиеся в языке Pascal встроенными, в С доступны лишь через специальные функции. Одной из возможностей языка С является инициализация строковых констант, например:

```
char str[64] = "Символьные строки";
```

Следует отметить, что в языке С автоматически добавляется *NULL* к строковым константам.

В модуле *string.h* предусмотрен следующий ряд функций для работы со строками:

1. *int strcmp(const char *s1, const char*s2);*

Сравнивает *s1* и *s2* и возвращает отрицательное число, если *s1* < *s2*; ноль, если *s1* = *s2*; или положительное, если *s1* > *s2*.

2. *char *strcat(char *dest, const char *src);*

Выполняет конкатенацию строк (копирует строку с начальным адресом *src* в конец строки с адресом *dest*). Возвращает указатель на результирующую строку.

3. *size_t strlen(const char *s);*

Возвращает длину строки *s*.

4. *char *strcpy(char *dest, const char *src);*

Копирует строку *src* в *dest*, дополняя последнюю символом '\0'.

5. *char *strncpy(char *dest, const char *src, size_t len);*

Копирует не более *len* символов из строки *src* в *dest*. Дополняет результат символом '\0', если символов в *src* меньше *len*. Возвращает указатель на результирующую строку.

6. *char *strstr(const char *s1, const char *s2);*

Возвращает указатель на первое вхождение *s2* в *s1* или, если такового не оказалось, *NULL*.

7. *int sprintf(char *s, const char *format [argument, ...]);*

Действует так же, как и *printf*, только вывод осуществляет в строку *s*, завершая ее символом '\0'. Строка *s* должна быть достаточно большой, чтобы вмещать результат вывода. Возвращает количество записанных символов, в число которых символ '\0' не входит. Функция располагается в модуле *string.h*.

Следующие три функции, описанные в *stdlib.h*, предназначены для перевода строки *s* в числовое значение:

```
double atof(const char *s);
int atoi(const char *s);
long atol(const char *s);
```

К о н т р о л ь н ы е в о п р о с ы

1. Какие структуры данных называются встроенными, а какие — производными?
2. Что представляет собой структура данных «строка» на абстрактном уровне?
3. Каков характер изменчивости встроенной структуры данных «строка» в языке Pascal?
4. На каких уровнях представления структур данных различаются встроенные структуры данных «строка» в языках Pascal и C?
5. Как реализованы операции над строками в языках Pascal и C?

Л а б о р а т о р н а я р а б о т а № 3

Сравнительный анализ методов сортировки (Pascal/C)

Цель работы: изучение методов сортировки массивов и приобретение навыков в проведении сравнительного анализа различных методов сортировки.

З а д а н и е

1. Изучить временные характеристики алгоритмов.
2. Изучить методы сортировки:
 - 1) включением;
 - 2) выбором;
 - 3) обменом:
 - 3.1) улучшенная обменом 1;
 - 3.2) улучшенная обменом 2;
 - 4) Шелла;
 - 5) Хоара;
 - 6) пирамидальная.
3. Программно реализовать методы сортировки массивов.

4. Разработать и программно реализовать средство для проведения экспериментов по определению временных характеристик алгоритмов сортировки.

5. Провести эксперименты по определению временных характеристик алгоритмов сортировки. Результаты экспериментов представить в виде таблицы 9, клетки которой содержат количество операций сравнения при выполнении алгоритма сортировки массива с заданным количеством элементов. Провести эксперимент для упорядоченных, неупорядоченных и упорядоченных в обратном порядке массивов (для каждого типа массива заполнить отдельную таблицу).

6. Построить график зависимости количества операций сравнения от количества элементов в сортируемом массиве.

7. Определить аналитическое выражение функции зависимости количества операций сравнения от количества элементов в массиве.

8. Определить порядок функций временной сложности алгоритмов сортировки при сортировке упорядоченных, неупорядоченных и упорядоченных в обратном порядке массивов.

Таблица 9

Результаты экспериментов

[illegible]

Содержание отчета

1. Тема лабораторной работы.
2. Цель работы.
3. Листинг программы.
4. Результаты работы программы.
5. Графики зависимостей ФВС.
6. Выводы по работе.

Теоретические сведения

Временная сложность (ВС) алгоритма — это зависимость времени выполнения алгоритма от количества обрабатываемых входных данных. Здесь представляет интерес среднее и худшее время выполнения алгоритма. ВС можно установить с различной точностью. Наиболее точной оценкой является аналитическое выражение для функции: $t = t(N)$, где t — время, N — количество входных данных (размерность). Данная функция называется *функцией временной сложности* (ФВС).

Например: $t = 5N^2 + 6N + 1$. Такая оценка может быть сделана только для конкретной реализации алгоритма в конкретной вычислительной системе и не пригодна для оценки алгоритма. Для сравнения алгоритмов достаточно определить лишь порядок функции временной сложности $t(N)$. Две функции $f_1(N)$ и $f_2(N)$ одного порядка, если

$$f_1(N) = O(f_2(N))$$

Иначе это записывается в виде: $f_1(N) = O(f_2(N))$ (Читается «О большое»).

Порядок функции, заданной многочленом, определяется только тем членом, который растет быстрее других с увеличением N , причем коэффициент при нем не учитывается.

Для определения порядка функции ВС алгоритма достаточно найти зависимость числа выполнения того оператора от количества исходных данных, который выполняется в алгоритме чаще других.

Для оценки алгоритмов можно использовать функцию зависимости числа операций сравнения $C = C(N)$ от количества обрабатываемых данных, т.к. функции $t(N)$ и $C(N)$ — функции одного порядка.

Сортировка — это процесс перестановки элементов данного множества в определенном порядке. Сортировка является идеальным примером огромного разнообразия алгоритмов, выполняющих одну и ту же задачу, многие из которых в некотором смысле являются оптимальными, а большинство имеет какие-либо преимущества по сравнению с остальными. По-

этому на примере сортировки убеждаются в необходимости проведения сравнительного анализа алгоритмов.

Формулировка задачи может быть записана следующим образом. Если даны элементы a_1, a_2, \dots, a_n , то сортировка означает перестановку этих элементов в массив $a_{k1}, a_{k2}, \dots, a_{kn}$, где при заданной функции упорядочения f справедливы отношения $f(a_{k1}) \leq f(a_{k2}) \leq \dots \leq f(a_{kn})$. Обычно функция упорядочения f не вычисляется по какому-то специальному правилу, а является значением элемента.

Метод сортировки называется *устойчивым*, если относительный порядок элементов с одинаковыми значениями не меняется при сортировке; неустойчивым — в противном случае.

Методы сортировки, в зависимости от лежащего в их основе приема, можно разбить на три базовых класса:

- 1) сортировка включением;
- 2) сортировка выбором;
- 3) сортировка обменом.

Сортировка включением

Этот метод обычно используют игроки в карты. Элементы (карты) условно разделяются на готовую последовательность a_1, \dots, a_{i-1} и входную последовательность a_i, \dots, a_n . На каждом шаге, начиная с $i=2$ и увеличивая i на единицу, берут 1-й элемент входной последовательности и передают его в готовую последовательность, вставляя на подходящее место.

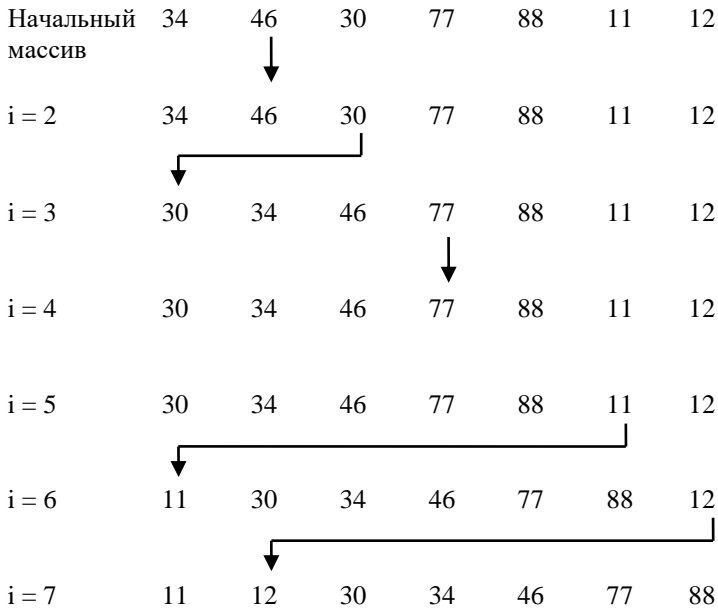
Алгоритм сортировки включением

1. Запоминаем элемент, подлежащий вставке.
2. Перебираем справа налево отсортированные элементы и сдвигаем каждый элемент вправо на одну позицию, пока не освободится место для вставляемого элемента.
3. Вставляем элемент на освободившееся место. Пункты 1-3 выполняем для всех элементов массива, кроме первого.

При поиске подходящего места для элемента x удобно чередовать сравнения и пересылки, т. е. как бы «просеивать» x , сравнивая его с очередным элементом $a[j]$ и либо вставляя x , либо пересылая $a[j]$ вправо. Просеивание может быть закончено при двух различных условиях:

- 1) найден элемент a_i с ключом меньшим, чем ключ x ;
- 2) достигнут левый конец готовой последовательности.

Пример.



Анализ сортировки включением

Если первоначальный массив отсортирован, то на каждом просмотре делается только одно сравнение, так что эта сортировка имеет порядок $O(N)$. Если массив первоначально отсортирован в обратном порядке, то данная сортировка имеет порядок $O(N^2)$, поскольку общее число сравнений равно $1 + 2 + 3 + \dots + (N - 2) + (N - 1) = (N - 1) \cdot N / 2$, что составляет $O(N^2)$. Чем ближе к отсортированному виду, тем более эффективной становится сортировка простыми вставками. Среднее число сравнений в сортировке простыми вставками также составляет $O(N^2)$.

Сортировка выбором

При сортировке выбором массив разбивается на две части: отсортированная и неотсортированная (вначале отсортированная часть пустая). На каждом шаге из неотсортированной части выбирают наименьший элемент и присоединяют его к отсортированной части.

Алгоритм сортировки выбором

1. Находим наименьший элемент в неупорядоченной части массива.
2. Меняем местами найденный элемент с тем, который соседствует с упорядоченной частью.
3. Пункты 1 и 2 выполняем, пока в неупорядоченной части имеется более одного элемента.

Пример.

Начальные значения	37	06	71	83	41	03	61
Проход 1	37	06	71	83	41	<u>03</u>	61
Проход 2	03	<u>06</u>	71	83	41	37	61
Проход 3	03	06	71	83	41	<u>37</u>	61
Проход 4	03	06	37	83	<u>41</u>	71	61
Проход 5	03	06	37	41	83	71	<u>61</u>
Проход 6	03	06	37	41	61	<u>71</u>	83
Проход 7	03	06	37	41	61	71	83

Алгоритм сортировки выбором в некотором смысле противоположен алгоритму сортировки вставками. При сортировке вставками на каждом шаге рассматривается только один очередной элемент входной последовательности и все элементы готового массива для нахождения места вставки. При сортировке простым выбором рассматриваются все элементы входного массива, и наименьший из них отправляется в готовую последовательность.

Анализ сортировки простым выбором

При сортировке простым выбором число сравнений ключей не зависит от их начального порядка. На первом просмотре выполняется $N - 1$ сравнение, на втором — $N - 2$ и т.д. Следовательно, общее число сравнений равно $(N - 1) + (N - 2) + (N - 3) + \dots + 1 = N(N - 1) / 2$, что составляет $O(N^2)$.

Порядок функции ВС не зависит от упорядоченности сортируемого массива, однако время сортировки упорядоченного массива будет минимальным, т.к. от упорядоченности массива зависит число перестановок элементов.

Сортировка обменом

Идея обменной сортировки заключается в том, что два элемента, нарушающие требуемый порядок, меняются местами.

Алгоритм сортировки обменом:

1. Перебираем поочередно все пары соседних элементов, начиная с последнего, и меняем местами элементы в парах, нарушающих порядок.
2. Пункт 1 повторяем $n - 1$ раз.

После первого прохода минимальный элемент массива занимает первую позицию и в дальнейшем в сортировке не участвует. После второго прохода образуется упорядоченная последовательность из первых двух элементов массива и т.д., пока все элементы массива не будут отсортированы.

Пример.

Начальные значения	37	06	71	83	41	03	61
Проход 1	03	37	06	71	83	41	61
Проход 2	03	06	37	41	71	83	61
Проход 3	03	06	37	41	61	71	83
Проход 4	03	06	37	41	61	71	83
Проход 5	03	06	37	41	61	71	83
Проход 6	03	06	37	41	61	71	83
Проход 7	03	06	37	41	61	71	83

Анализ сортировки обменом

При использовании сортировки обменом число сравнений элементов не зависит от их начального порядка. На первом просмотре выполняется $N - 1$ сравнение, на втором — $N - 2$ и т.д. Следовательно, общее число сравнений равно $(N - 1) + (N - 2) + (N - 3) + \dots + 1 = N(N - 1) / 2$, что составляет $O(N^2)$.

Перестановки в упорядоченном массиве не выполняются, а в массиве, упорядоченном в обратном порядке, их число равно числу сравнений элементов. Следовательно, время выполнения алгоритма зависит от упорядоченности массива.

Улучшенная сортировка обменом 1

После каждого прохода в сортировке обменом может быть сделана проверка, были ли совершены перестановки в течение данного прохода. Если перестановок не было, то это означает, что массив упорядочен и дальнейших проходов не требуется.

Анализ улучшенной сортировки обменом 1

Число сравнений для этого метода зависит от числа проходов, необходимых для сортировки. В худшем случае, когда массив упорядочен в обратном порядке, выполняется $N - 1$ проходов. На первом проходе выполняется $N - 1$ сравнение, на втором — $N - 2$ и т.д. Следовательно, общее число сравнений равно $(N - 1) + (N - 2) + (N - 3) + \dots + 1 = N \cdot (N - 1) / 2$, что составляет $O(N^2)$. В этом случае выполняется максимальное число перестановок, что увеличивает время выполнения алгоритма.

В лучшем случае, когда массив уже упорядочен, потребуется всего один проход и $N - 1$ сравнение, что составляет $O(N)$. Перестановки в этом случае не выполняются.

Улучшенная сортировка обменом 2

В отличие от улучшенной сортировки обменом 1 здесь в течение прохода фиксируется последний элемент, участвующий в обмене. В очередном проходе этот элемент и все предшествующие в сравнении не участвуют, т.к. все элементы до этой позиции уже отсортированы.

Анализ улучшенной сортировки обменом 2


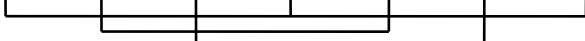

Анализ улучшенной сортировки обменом 2 аналогичен анализу улучшенной сортировки обменом 1. Порядок функций ВС этих алгоритмов в лучшем и худшем случаях одинаковый.

Сортировка Шелла

Сортировка Шелла — это улучшенный метод сортировки вставками. Был предложен Д.Л. Шеллом в 1959 г. Рассмотрим этот метод на примере (см. ниже). При первом проходе группируются и сортируются элементы, отстоящие друг от друга на 5 позиций: (X_1, X_6) , (X_2, X_7) , (X_3) , (X_4) , (X_5) , т.е. выполняется сортировка массива с шагом 5; при втором проходе — элементы, отстоящие друг от друга на три позиции: (X_1, X_4, X_7) , (X_2, X_5) , (X_3, X_6) ; при третьем — на 1 позицию.

Если некоторый массив частично отсортирован с использованием шага h , а затем сортируется с использованием меньшего шага, то массив остается частично отсортированным по шагу h , т.е. последующие частичные сортировки не нарушают результата предыдущих сортировок. Следующий шаг сортировки меньше предыдущего, а последний — равен 1.

Пример.

Начальные значения	25	57	82	63	90	75	80
Просмотр 1	25	57	82	63	90	75	80
Шаг 5							
Просмотр 2	25	57	82	63	90	75	80
Шаг 3							
Просмотр 3	25	57	75	63	90	82	80
Шаг 1							
Упорядоченный массив	25	57	63	75	80	82	90

Алгоритм сортировки Шелла

1. Определить количество проходов t и шаг сортировки на каждом проходе. Результат сохранить в массиве h .
2. На i -ом проходе ($i=1, \dots, t$) выполнить сортировку включением с шагом $h(i)$.

Анализ сортировки Шелла

Поскольку первый в сортировке Шелла используемый шаг является большим, отдельные подмассивы достаточно малы по отношению к исходному. Таким образом, сортировка включением над этими подмассивами работает достаточно быстро — $O(N^2)$ (эффективно при малом N). Сортировка каждого подмассива приводит к тому, что весь массив становится ближе к отсортированному виду, что также делает эффективным использование сортировки включением. Так как массив становится более упорядоченным, то $O(N) < \text{порядок ФВС} < O(N^2)$.

Анализ сортировки Шелла математически сложен. В случае правильного выбора шагов порядок ФВС будет выглядеть как $O(N^{1.2})$. Д. Кнут предложил выбирать шаги из следующего ряда: 1, 4, 13, 40, 121, ..., а вычислять их по формуле: $h_{i-1} = 3 \cdot h_i + 1$; $h_t = 1$; $t = \lceil \log_3 N \rceil - 1$ (количество просмотров), где N — размерность исходного массива. Т.е. элементы должны быть взаимно простыми числами. Исходя из этого порядок сортировки может быть аппроксимирован величиной $O(N(\log_2 N))$.

Сортировка Хоара

Сортировка Хоара — лучший из известных до сего времени метод сортировки массивов. Он обладает столь блестящими характеристиками, что его автор Ч. Хоар назвал эту сортировку быстрой. Быстрая сортировка основана на том факте, что для достижения наибольшей эффективности желательно разбить массив на подмассивы и сортировать подмассивы меньшего размера.

Алгоритм быстрой обменной сортировки:

1. Выбираем элемент массива в качестве разделителя (например, первый).
2. Располагаем элементы меньше разделителя в первой части массива, а большие — во второй.
3. Если число элементов в первой части массива больше 1, то применяем к ней алгоритм в целом, иначе конец алгоритма.
4. Если число элементов во второй части массива больше 1, то применяем к ней алгоритм в целом, иначе конец алгоритма.

Данный алгоритм основан на том утверждении, что ни один из элементов, расположенных в первой части массива не обменяется с каким либо элементом, расположенным во второй части.

Пункт 2 алгоритма выполняется следующим образом: просматриваем массив от краев к центру и меняем местами элементы, нарушающие порядок.

Пример.

(30	10	40	20	15	17	45	60)
(17	10	15	20)	(40	30	45	60)
(30	10	17	20	15)	(40	45	60)

Анализ сортировки Хоара

Пусть $m = \log_2 N$, где N — количество элементов. Будем считать, что количество элементов, меньших разделителя, равно количеству элементов, больших разделителя, т.е. массив разбивается пополам на две равные части. Определим количество сравнений в этом случае:

$$N + 2 \cdot (N/2) + \dots + m \cdot (N/m) = O(N \cdot m) = O(N \cdot \log_2 N).$$

Если каждый раз одна из частей массива содержит не более одного элемента, то порядок будет $O(N^2)$.

Характер разбиения массива, а, следовательно, и порядок функции ВС, зависит от соотношения делителя и остальных элементов массива. Для определения «хорошего» делителя имеются различные алгоритмы.

Пирамидальная сортировка

Пирамидальная сортировка является улучшенной сортировкой выбором. Из массива, состоящего из N элементов, выбирается максимальный и меняется местами с последним. Затем рассматривается массив из $N - 1$ элементов. Процесс повторяется до тех пор, пока количество рассматриваемых элементов больше одного. Пирамидальная сортировка отличается от сортировки выбором поиском максимального элемента. Чтобы понять пирамидальную сортировку, массив нужно интерпретировать как бинарное дерево, в корне которого находится первый элемент массива, на втором уровне — второй и третий, на третьем — с четвертого по седьмой и т.д. Для i -го элемента массива можно определить номер P элемента, являющегося родителем, как $P = i \div 2$; номер L элемента, являющегося левым сыном, как $L = 2 \cdot i$; номер R элемента, являющегося правым сыном, как $R = 2 \cdot i + 1$. Если в массиве (в дереве) N элементов, то последний элемент, имеющий хотя бы одного левого сына, имеет номер $(N \div 2)$. Массив M (см. таблицу 10) можно представить деревом на рис.5.

Таблица 10

Массив М

номер элемента массиваМ	1	2	3	4	5	6	7	8	9	10
значение элемента массива М	42	5	87	1	74	12	63	25	58	33

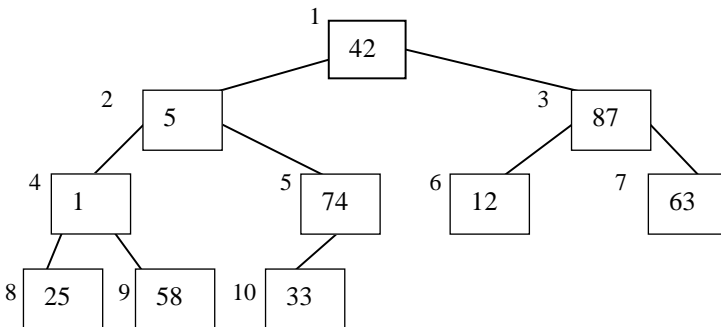


Рис. 5. Представление массива в виде дерева

Поиск максимального элемента массива в этом алгоритме основан на понятии пирамиды. Дерево (поддерево) является пирамидой, если каждый элемент в нем больше или равен элементам, которые являются его сыновьями. Корень пирамиды — максимальный элемент массива. Пирамида для массива М (см. таблицу 10) представлена на рис. 6.

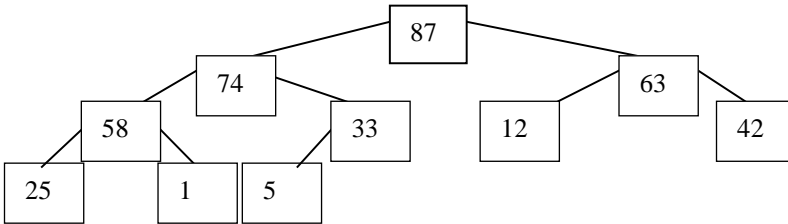


Рис.6. Пирамида для массива М

Построив пирамиду для массива, можно, в соответствии с алгоритмом сортировки вставками, максимальный элемент, который находится в корне пирамиды (корень пирамиды — первый элемент массива), поменять местами с последним элементом массива. В результате получим массив М (см. таблицу 11) и его представление без последнего элемента в виде дерева на рис.7.

Таблица 11

Массив М

номер элемента массива М	1	2	3	4	5	6	7	8	9	10
значение элемента массива М	5	74	63	58	33	12	42	25	1	87

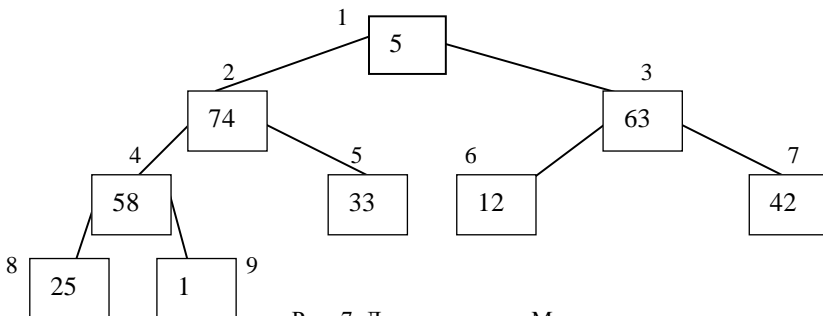


Рис. 7. Дерево массива М

Дерево на рис.7 не является пирамидой, но левое и правое поддеревы — пирамиды. Для того, чтобы это дерево перестроить в пирамиду, достаточно значение из корня дерева «опустить вниз» меняя его местами с сыном, имеющим наибольшее значение. Обмен производить до тех пор, пока есть сын, больший, чем элемент в корне. При перестроении дерева на рис.7 в пирамиду на рис.8 корневой элемент опускается на седьмой, а третий и седьмой поднимаются соответственно на первый и третий. Пирамида построена и теперь корень можно поменять с последним элементом дерева и в дальнейшем его не обрабатывать. Для полной сортировки процесс повторяется, пока в дереве количество вершин больше одного.

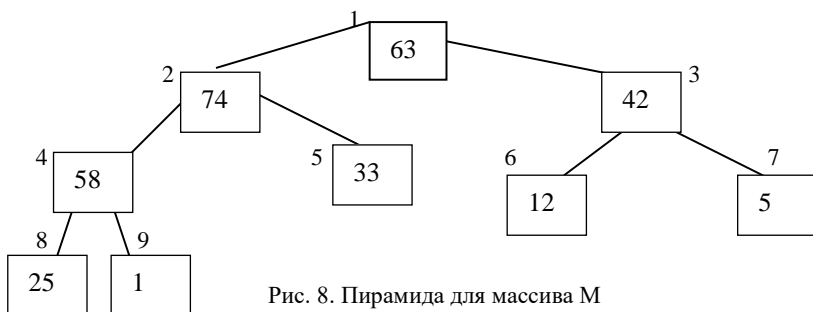


Рис. 8. Пирамида для массива М

Сформулируем алгоритм перестроения дерева, у которого левое и правое поддерево — пирамиды, в пирамиду.

Алгоритм *MakeHeap*.

1. Запоминаем корневой элемент.
2. Перебираем элементы в направлении большего сына и сдвигаем каждый элемент “вверх” на одну позицию, пока не освободится место для корневого элемента.
3. Вставляем корневой элемент на освободившееся место.

Для построения пирамиды из произвольного дерева (массива) необходимо построить пирамиду по алгоритму *MakeHeap* для каждого элемента, имеющего хотя бы одного сына, причем построение должно идти от последнего такого элемента к первому, т.е. снизу вверх.

Теперь можем сформулировать алгоритм пирамидальной сортировки.

Алгоритм *HeapSort*.

1. Построить пирамиду для исходного массива.
2. Пока в массиве более одного элемента, переставить первый и последний элемент, уменьшить размер массива на единицу и перестроить дерево в пирамиду по алгоритму *MakeHeap*.

Анализ пирамидальной сортировки

Пусть дерево массива на нижнем (нулевом) уровне содержит максимальное число вершин. Для построения пирамиды из произвольного дерева (первая часть алгоритма) нужно обработать все вершины, начиная с первого уровня. Для обработки вершины на i -том уровне число сравнений пропорционально i . Число вершин на i -том уровне равно $2^{\lfloor \log_2 N \rfloor - i}$, а всего уровней $m = \lfloor \log_2 N \rfloor$, следовательно, общее число сравнений определяется формулой: $1 \cdot 2^{\lfloor \log_2 N \rfloor - 1} + 2 \cdot 2^{\lfloor \log_2 N \rfloor - 2} + \dots + m \cdot 2^{\lfloor \log_2 N \rfloor - m} = O(N)$.

Во второй части алгоритма последовательно обрабатываются деревья с $N, N-1, \dots, 2$ вершинами. Число сравнений для перестройки дерева, состоящего из i вершин, в пирамиду пропорционально $\lfloor \log_2 i \rfloor$, следовательно, общее число сравнений

$$\lfloor \log_2 N \rfloor + \lfloor \log_2 (N-1) \rfloor + \dots + \lfloor \log_2 2 \rfloor = O(N \cdot \log_2 N).$$

Таким образом порядок функции ВС алгоритма пирамидальной сортировки $O(N \cdot \log_2 N)$.

Примеры программной реализации алгоритмов сортировки на языке Pascal

{=====СОРТИРОВКА ВКЛЮЧЕНИЕМ=====}

```
{ процедура сортировки включением }
procedure InsertSort(var a:t_mas;n:t_index);
var i,j: t_index;
    x : t_el;
begin
  for i := 2 to n do
    begin
      x := a[i];
      j := i-1;
      while (x < a[j]) and (j >= 1) do
        begin
          a[j+1] := a[j];
          j := j-1;
        end;
    end;
```

```

    a[j+1] := x;
end;
end;

```

```

{=====СОРТИРОВКА ВЫБОРОМ=====}

```

```

{ процедура обмена значений }
procedure Swap_el(var a,b:t_el);
var c:t_el;
begin
    c:=a;
    a:=b;
    b:=c;
end;

```

```

{ процедура сортировки выбором }
procedure ChoiceSort(var a:t_mas;n:t_index);
var i,j:t_index;
    x :t_el;
begin
    for i := 1 to n do
        begin
            x := a[i];
            for j := i+1 to n do
                if (a[j] < x) then
                    x := a[j];
            swap_el(a[i],x);
            end;
        end;
    end;
end;

```

```

{=====СОРТИРОВКА ОБМЕНОМ=====}

```

```

{ процедура обмена значений }
procedure Swap_el(var a,b:t_el);
var c:t_el;
begin
    c:=a;
    a:=b;
    b:=c;
end;

```

```

{ процедура сортировки обменом }

```

```
procedure BblSort(var a:t_mas;n:t_index);
```

```
var i,j : t_index;
```

```
begin
```

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

```
    for j := n downto i do
```

```
      if (a[j-1] >= a[j]) then
```

```
        swap_el(a[j-1],a[j]);
```

```
end;
```

```
{ процедура улучшенной сортировки обменом 1 }
```

```
procedure BblSort_improv1(var a:t_mas;n:t_index);
```

```
var i,j : t_index;
```

```
  fl: boolean;
```

```
begin
```

```
  i := 2;
```

```
  repeat
```

```
    fl:= false; //перестановок изначально нет
```

```
    for j := n downto i do
```

```
      if (a[j-1] >= a[j]) then
```

```
        begin
```

```
          swap_el(a[j-1],a[j]);
```

```
          fl:= true; //некоторые элементы переставлены
```

```
        end;
```

```
      i := i+1;
```

```
    until (not fl)or(i > n);
```

```
end;
```

```
{ процедура улучшенной сортировки обменом 2 }
```

```
procedure BblSort_improv2(var a:t_mas;n:t_index);
```

```
var i,j,k : t_index;
```

```
begin
```

```
  i := 2;
```

```
  k := n+1;
```

```
  for j := n downto i do
```

```
    if (a[j-1] >= a[j]) then
```

```
      begin
```

```
        swap_el(a[j-1],a[j]);
```

```
        k := j-1; //запоминаем индекс последнего обмененного элемента
```

```
      end;
```

```
    i := k;
```

```
  until (i > n);
```

```
end;
```

{=====СОРТИРОВКА МЕТОДОМ ШЕЛЛИА=====}

```
{ процедура сортировки методом Шелла }
procedure ShellSort(var a:t_mas;n:t_index);
type t_arr = array [1..65520 div sizeof(word)] of word;
var i,j,hh,t,s : t_index;
    k          : integer;
    h          : t_arr;
begin
    t := round(ln(n)/ln(3))-1;
    if (t < 1) then
        t := 1;
    h[t] := 1;
    for k := t downto 2 do
        h[k-1] := 3*h[k]+1;
    for s := t downto 1 do
        begin
            hh := h[s];
            for j := hh+1 to n do
                begin
                    i := j-hh;
                    k := a[j];
                    while (k <= a[i])and(i > 0) do
                        begin
                            a[i+hh] := a[i];
                            i := i-hh;
                        end;
                        a[i+hh] := k;
                    end;
                end;
            end;
        end;
    end;
```

{=====СОРТИРОВКА МЕТОДОМ ХОАРА=====}

```
{ процедура сортировки методом Хоара }
procedure HoarSort(var a: t_arr; n: integer);
procedure QSort(L,R:integer);
var x,t,i,j:integer;
begin
    x:=a[L]; // в качестве разделителя выбираем первый элемент
```

```

i:=L;
j:=R;
while i≤j do
  begin
    while a[i]<x do
      inc(i);
    while a[j]>x do
      dec(j);
    if i≤j then
      begin
        t:=a[i];
        a[i]:=a[j];
        a[j]:=t;
        inc(i);
        dec(j);
      end;
    end;
  if L<j then
    QSort(L,j);
  if i<R then
    QSort(i,R);
end;

```

```

begin
  QSort(1,n);
end;

```

{=====ПИРАМИДАЛЬНАЯ СОРТИРОВКА=====}

```

procedure sift(var a:t_mas; L,R:t_index);
var i,j : t_index;
    c : t_el;
begin
  i := L;
  j := 2*L;
  c := a[L];
  if (j < R)and(a[j] < a[j+1]) then
    j := j+1;
  while (j ≤ R)and(c < a[j]) do
    begin

```

```

    Swap_el(a[i],a[j]);
    i := j;
    j := 2*j;
    if (j < R) and (a[j] < a[j+1]) then
        j := j+1;
    end;
end;

{ процедура пирамидальной сортировки }
procedure HeapSort(var a:t_mas;n:t_index);
var i,L,R : t_index;
begin
    L := n div 2+1;
    R := n;
    while (L > 1) do
        begin
            L := L-1;
            Sift(a,L,R);
        end;
    while (R > 1) do
        begin
            Swap_el(a[1],a[R]);
            R := R-1;
            Sift(a,L,R);
        end;
    end;
end;

```

Примеры программной реализации алгоритмов сортировки на языке С

```

/*=====СОРТИРОВКА ВКЛЮЧЕНИЕМ=====*/

/* функция сортировки включением */
void Sis(int A[],int nn)
{ int i,j,k;
  for ( j=1; j<nn; j++ )
    { k = A[j];
      i = j -1;
      while ( k < A[i] && i >= 0)
        { A[i+1] = A[i];
          i -= 1; }
    }
}

```

```

        A[i+1] = k;
    }
}

/*=====СОРТИРОВКА ВЫБОРОМ=====*/

/*функция сортировки выбором */
void StrSel(int A[],int nn)
{ int i,j,x,k;
  for ( i=0; i<nn-1; i++ )
  { x = A[i]; k = i;
    for (j=i+1; j<nn; j++)
      if (A[j] < x)
        { k = j; x = A[k]; }
    A[k] = A[i]; A[i] = x;
  }
}

/*=====СОРТИРОВКА ОБМЕНОМ=====*/

/* функция сортировки обменом */
void BblSort(int A[],int nn)
{ int i,j,k,p;
  for ( i=0; i<nn-1; i++ )
  { p = 0;
    for (j=nn-1; j>i; j--)
      if (A[j] < A[j-1])
        { k = A[j]; A[j] = A[j-1]; A[j-1] = k; p = 1;}
    /* Если перестановок не было, то сортировка выполнена */
    if ( p == 0)
      break;
  }
}

/*=====СОРТИРОВКА МЕТОДОМ ШЕЛЛА=====*/

/* функция сортировки методом Шелла */
void ShellSort(int a [], int n){
  int i,j,k,hh,t,s;
  int h [1000];
  t = round(ln(n)/ln(3))-1;
  if (t < 1){

```



```

        t = 1;
    };
    h[t] = 1;
    for (k=t-1; k >= 1; k--) {
        h[k-1] = 3*h[k]+1;
    }
    for (s=t-1; s >= 0; s--) {
        hh = h[s];
        for (j = hh; j <= n; j++) {
            i = j-hh;
            k = a[j];
            while ((k <= a[i]) && (i >= 0)) {
                a[i+hh] = a[i];
                i = i-hh;
            }
            a[i+hh] = k;
        }
    }
}

```

/*=====СОРТИРОВКА МЕТОДОМ ХОАРА=====*/

```

void QSort(int a [], int L, int R){
    int x = a[L], i = L, j = R, t; // в качестве разделителя выбираем первый
    элемент
    while (i <= j) {
        while (a[i] < x)
            i++;
        while (a[j] > x)
            j--;
        if (i <= j) {
            t = a[i];
            a[i] = a[j];
            a[j] = t;
            i++;
            j--;
        }
    }
    if (L < j)
        QSort(a, L, j);
    if (i < R)
        QSort(a, i, R);
}

```

```

/*функция сортировки методом Хоара*/
void HoarSort(int a[], int n){

    QSort(a, 1, n);
}
/*=====ПИРАМИДАЛЬНАЯ СОРТИРОВКА===== */

/* пирамидальная функция сортировки */
void HeapSort(int A[], int nn)
{ int L, R, x, i;
    L = nn/2 ; R = nn-1;
/* Построение пирамиды из исходного массива */
    while ( L>0 )
    { L = L - 1;
        Sift(A, L, R);
    }
/* Сортировка: пирамида => отсортированный массив */
    while ( R>0 )
    { x = A[0]; A[0] = A[R]; A[R] = x;
        R--;
        Sift(A, L, R);
    }
}
/* ===== */
void Sift(int A[], int L, int R)
{ int i, j, x, k;
    i = L;
    j = 2*L+1;
    x = A[L];
    if ((j<R) && (A[j]<A[j+1]))
        j++;
    while ((j<=R) && (x<A[j]))
    { k=A[i]; A[i] = A[j]; A[j]=k;
        i = j;
        j = 2*j+1;
        if ((j<R) && (A[j]<A[j+1]))
            j++;
    }
}
/* ***** */

```

Контрольные вопросы

1. Что такое временная сложность алгоритма?
2. Почему функцию временной сложности нельзя использовать для оценки алгоритма?
3. Что такое порядок функции? Как определяется порядок функции, заданной многочленом?
4. Как можно определить порядок функции временной сложности алгоритма?
5. Что называется сортировкой?
6. В каком случае метод сортировки называется устойчивым?
7. Как выполняется сортировка включением?
8. Зависит ли время сортировки включением от упорядоченности массива?
9. Зависит ли порядок функции временной сложности сортировки включением от упорядоченности массива?
10. Выполните анализ сортировки включением.
11. Реализуйте алгоритм сортировки включением на языке программирования.
12. Как выполняется сортировка выбором?
13. Зависит ли время сортировки выбором от упорядоченности массива?
14. Зависит ли порядок функции временной сложности сортировки выбором от упорядоченности массива?
15. Выполните анализ сортировки выбором.
16. Реализуйте алгоритм сортировки выбором на языке программирования.
17. Как выполняется сортировка обменом?
18. Зависит ли время сортировки обменом от упорядоченности массива?
19. Зависит ли порядок функции временной сложности сортировки обменом от упорядоченности массива?
20. Выполните анализ сортировки обменом.
21. Реализуйте алгоритм сортировки обменом на языке программирования.
22. Как можно улучшить сортировку обменом?
23. Почему сортировка Шелла быстрее сортировки вставками?
24. Выполните итеративную реализацию сортировки Хоара.
25. Чем пирамидальная сортировка отличается от сортировки выбором?

Лабораторная работа № 4

Сравнительный анализ алгоритмов поиска (Pascal/C)

Цель работы: изучение алгоритмов поиска элемента в массиве и закрепление навыков в проведении сравнительного анализа алгоритмов.

З а д а н и е

1. Изучить алгоритмы поиска:
 - 1) в неупорядоченном массиве:
 - линейный;
 - быстрый линейный;
 - 2) в упорядоченном массиве:
 - быстрый линейный;
 - бинарный;
 - блочный.
2. Разработать и программно реализовать средство для проведения экспериментов по определению временных характеристик алгоритмов поиска.
3. Провести эксперименты по определению временных характеристик алгоритмов поиска. Результаты экспериментов представить в виде таблиц 12 и 13. Клетки таблицы 12 содержат максимальное количество операций сравнения при выполнении алгоритма поиска, а клетки таблицы 13 — среднее число операций сравнения.
4. Построить графики зависимости количества операций сравнения от количества элементов в массиве.
5. Определить аналитическое выражение функции зависимости количества операций сравнения от количества элементов в массиве.
6. Определить порядок функций временной сложности алгоритмов поиска.

С о д е р ж а н и е о т ч е т а

1. Тема лабораторной работы.
2. Цель работы.
3. Листинг программы.
4. Результаты работы программы.
5. Графики зависимостей ФВС.
6. Выводы по работе

Теоретические сведения

Задача поиска заключается в том, чтобы определить наличие в массиве элемента, равного заданному.

Во многих программах поиск требует наибольших временных затрат, так что замена плохого поиска на хороший, часто ведет к существенному увеличению скорости работы программы. Выбор алгоритма поиска зависит от характера организованности массива.

Алгоритмы поиска в неупорядоченных массивах

Алгоритм линейного поиска

Работа алгоритма заключается в том, что элементы массива, начиная с первого, последовательно сравниваются с искомым элементом. Сравнение элементов продолжается до тех пор, пока не будут просмотрены все элементы или очередной элемент массива не равен искомому.

Алгоритм быстрого линейного поиска

Любой алгоритм поиска содержит блок проверки на окончание массива. В алгоритме линейного поиска эта проверка осуществляется каждый раз перед обращением к очередному элементу. Однако проверка на окончание массива может осуществляться не при каждом сравнении. Для этого в конец массива включается $(N + 1)$ -й элемент (N — количество элементов в массиве), равный искомому. Тогда проверка на окончание массива осуществляется лишь при совпадении очередного элемента с искомым. Если этот элемент находится внутри массива, то поиск заканчивается удачно и элемент считается найденным. Если же этот элемент оказался $(N + 1)$ -ым, то искомого элемента в массиве нет.

Анализ алгоритмов линейного поиска

Для отыскания искомого элемента с использованием алгоритмов линейного поиска в худшем случае придется просмотреть все N элементов массива, следовательно порядок функции ВС будет $O(N)$. В лучшем случае, когда искомым элементом равен первому элементу массива, число сравнений не зависит от количества элементов в массиве и порядок функции ВС будет $O(1)$.

Оценим среднее время алгоритма, при этом будем исходить из следующего: пусть P_i — вероятность того, что будет осуществляться поиск эле-

мента со значением k_i ; предположим, что $\sum_{i=1}^n P_i = 1$, т.е. элемент со значением k_i не будет отсутствовать (в массиве обязательно есть элемент, поиск которого осуществляется).

Среднее время (\bar{t}), как следует из алгоритма, пропорционально среднему числу операций сравнения (\bar{c}) и равно

$$\bar{t} \sim \bar{c} = 1 \cdot P_1 + 2 \cdot P_2 + \dots + N \cdot P_N.$$

Если ключи имеют одинаковую вероятность $P_1 = P_2 = \dots = P_N = P$, а также учитывая, что $\underbrace{p + p + \dots + p}_n = 1$, $N \cdot P = 1$, $P = 1/N$, тогда

$$\bar{t} = (1 + 2 + \dots + N) \cdot P = \frac{(N+1) \cdot N}{2} \cdot P = \frac{N+1}{2} = O(N),$$

т.е. при линейном поиске в среднем необходимо просмотреть половину массива.

Для экспериментального определения среднего числа сравнений необходимо найти суммарное число сравнений при поиске всех элементов массива и разделить на количество элементов.

Алгоритмы поиска в упорядоченных массивах

Алгоритм быстрого линейного поиска

Применяя алгоритм быстрого линейного поиска для поиска элемента в упорядоченном массиве, поиск можно прекратить, если очередной элемент массива будет больше искомого. Это будет означать, что искомого элемента в массиве нет. В случае поиска элемента, который есть в массиве, этот алгоритм аналогичен алгоритму быстрого линейного поиска в неупорядоченном массиве.

Алгоритм бинарного поиска

Принцип, лежащий в основе алгоритма бинарного поиска (и некоторых других алгоритмов), состоит в том, что иногда удастся последовательно уменьшать объем задачи до такой степени, что ее решение в конце концов становится тривиальным. Главный шаг при бинарном поиске — взять элемент из середины массива и, если он не равен искомому, то в зависимости от его значения ту или другую половину массива убрать из рассмотрения. Повторное выполнение этого шага быстро сокращает размер области поиска.

Алгоритм бинарного поиска

1. Определить середину массива.
2. Если элемент, находящийся в середине массива, совпадает с искомым, поиск завершен.
3. Если элемент из середины массива больше искомого, применить бинарный поиск к первой половине массива.
4. Если элемент из середины массива меньше искомого, бинарный поиск необходимо применить ко второй половине массива.
5. Пункт 1-4 повторять, пока размер области поиска не уменьшается до нуля. Если это произойдет — ключа в массиве нет.

Анализ алгоритма бинарного поиска

Порядок функции ВС алгоритма бинарного поиска равен $O(\log_2 N)$. Это объясняется тем, что на каждом шаге поиска вдвое уменьшается область поиска. До того, как она станет равной одному элементу, произойдет не более $\log_2 N$ таких уменьшений, т.е. выполняется не более $\log_2 N$ шагов алгоритма. В лучшем случае, когда искомый элемент находится в середине массива, число сравнений не зависит от количества элементов в массиве и порядок функции ВС будет $O(1)$.

Определим среднее число шагов при поиске элементов в массиве из N элементов. Для этого подсчитаем суммарное число шагов S при поиске каждого элемента массива. Исходя из алгоритма, не более чем $N/2$ элементов ищется за $\log_2 N$ шагов, $N/4$ элементов — за $(\log_2 N) - 1$ шагов, и т.д. Отсюда $S = ((N/2) \cdot \log_2 N) + ((N/4) \cdot ((\log_2 N) - 1)) + \dots + 1$.

Среднее число шагов равно S/N , что составляет $O(\log_2 N)$.

Алгоритм блочного поиска

Блочный поиск состоит в том, что массив, упорядоченный по возрастанию, разбивается на определенное число блоков. В процессе поиска искомый элемент последовательно сравнивается с последним элементом блоков. Если искомый элемент меньше последнего элемента очередного блока, то искомый элемент может находиться только внутри этого блока. Для поиска элемента в блоке можно применить линейный поиск.

Анализ алгоритма блочного поиска

Пусть массив состоит из N элементов и при поиске разбивается на X равных блоков. Тогда в каждом блоке будет не более N/X элементов. В худшем случае, если искомый элемент окажется в последнем блоке, то для поиска блока потребуется просмотреть X элементов массива. Выполнить линейный поиск в блоке можно, просмотрев не более чем N/X элементов.

Следовательно, общее число обрабатываемых элементов не превышает $X + N / X$. Эта величина зависит от числа блоков и будет минимальна, когда ее производная равна нулю, т.е. при $X = \sqrt{N}$. Число записей в блоке так же равно \sqrt{N} . При таком разбиении массива на блоки в худшем случае понадобится обработать не более чем $2 \cdot \sqrt{N}$ элементов массива, а в среднем — \sqrt{N} элементов.

К о н т р о л ь н ы е в о п р о с ы

1. В чем заключается задача поиска?
2. Всегда ли быстрый линейный поиск быстрее линейного поиска?
3. От чего зависит время поиска в неупорядоченном массиве?
4. Чем алгоритм быстрого линейного поиска в упорядоченном массиве отличается от алгоритма быстрого линейного поиска в неупорядоченном массиве?
5. В чем заключается бинарный поиск?
6. Определите индексы элементов массива, бинарный поиск которых наиболее продолжителен.
7. Разработайте и реализуйте итеративный и рекурсивный алгоритмы бинарного поиска?
8. В чем заключается блочный поиск?
9. От чего зависит время блочного поиска?
10. Как правильно выбрать количество блоков в блочном поиске?
11. Определите максимальное количество элементов массива, которые могут быть обработаны при блочном поиске.
12. Пусть искомый элемент равен i -му элементу массива. Какой алгоритм рациональнее использовать в этом случае?
13. Выполните сравнительный анализ алгоритмов поиска для случая, когда искомого элемента нет в массиве.
14. Выполните сравнительный анализ алгоритмов поиска для случая, когда в массиве только один элемент.
15. Реализуйте алгоритмы поиска на языке программирования высокого уровня. Выполните трассировку при поиске в массиве из одного элемента.
16. От чего зависит порядок функции временной сложности алгоритмов поиска. Каким он может быть для различных алгоритмов?

Лабораторная работа № 5

Структуры данных «линейные списки» (Pascal/C)

Цель работы: изучить СД типа «линейный список», научиться их программно реализовывать и использовать.

З а д а н и е

1. Для СД типа «линейный список» определить:
 - 1.1. Абстрактный уровень представления СД:
 - 1.1.1. Характер организованности и изменчивости.
 - 1.1.2. Набор допустимых операций.
 - 1.2. Физический уровень представления СД:
 - 1.2.1. Схему хранения.
 - 1.2.2. Объем памяти, занимаемый экземпляром СД.
 - 1.2.3. Формат внутреннего представления СД и способ его интерпретации.
 - 1.2.4. Характеристику допустимых значений.
 - 1.2.5. Тип доступа к элементам.
 - 1.3. Логический уровень представления СД.
 Способ описания СД и экземпляра СД на языке программирования.
2. Реализовать СД типа «линейный список» в соответствии с вариантом индивидуального задания (см. табл.14) в виде модуля.
3. Разработать программу для решения задачи в соответствии с вариантом индивидуального задания (см. табл.14) с использованием модуля, полученного в результате выполнения пункта 2 задания.

Таблица 14

Варианты индивидуальных заданий

Номер варианта	Номер модуля	Задача
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	1	9

Окончание табл.14

10	2	10
11	3	11
12	4	1
13	5	2
14	6	3
15	7	4
16	8	5
17	1	6
18	2	7
19	3	8
20	4	9
21	5	10
22	6	11
23	7	1
24	8	2
25	1	3
26	2	4
27	3	5
28	4	6
29	5	7
30	6	8

Задачи

1. Многочлен $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ с целыми коэффициентами можно представить в виде списка, причем если $a_i = 0$, то соответствующее звено не включать в список. Определить логическую функцию РАВНО(p, q), проверяющие на равенство многочлены p и q .

2. Дано натуральное число n целые числа a_1, a_2, \dots, a_n . Требуется получить последовательность $x_1, y_1; x_2, y_2; \dots; x_k, y_k$, где x_1, \dots, x_k — взятые в порядке следования (слева на право) четные члены последовательности a_1, \dots, a_n , а y_1, \dots, y_k — нечетные члены, $k = \min(m, l)$.

3. Дано натуральное число n и целые числа a_1, a_2, \dots, a_n . Вычислить $\min_{1 \leq i \leq n} |a_i - \bar{a}|$, где \bar{a} среднее арифметическое чисел a_1, \dots, a_n .

4. Даны натуральные числа k, m, n , последовательности символов $s_1, \dots, s_k, t_1, \dots, t_m, u_1, \dots, u_n$. Получить по одному разу те символы, которые входят во все три последовательности.

5. Многочлен $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ с целыми коэффициентами можно представить в виде списка, причем если $a_i = 0$, то соответствующее звено не включать в список. Определить функцию, вычисляющую значения многочлена в точке x .

6. Даны натуральное число n , символы s_1, \dots, s_n . Получить символы, принадлежащие последовательности s_1, \dots, s_n , которые входят в нее по одному разу.

7. Многочлен $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ с целыми коэффициентами можно представить в виде списка, причем если $a_i = 0$, то соответствующее звено не включать в список. Определить процедуру, которая строит многочлен p — сумму многочленов q и r ;

8. Многочлен $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ с целыми коэффициентами можно представить в виде списка, причем если $a_i = 0$, то соответствующее звено не включать в список. Определить процедуру **ВЫВОД**(p, y), которая печатает многочлен p как многочлен от переменной, однокбуквенное имя которой является значением литерного параметра y .

9. Многочлен $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ с целыми коэффициентами можно представить в виде списка, причем если $a_i = 0$, то соответствующее звено не включать в список. Определить процедуру, которая строит многочлен p — произведение многочленов q и r .

10. Проверить, удовлетворяют ли элементы списка (базовый тип *integer*) закону $x = f(x_0, h)$, где x — элемент списка, h — шаг, x_0 — начальный элемент списка.

Пример: $x_0 = 5$, $h = 1$. $x_1 = 6$, $x_2 = 7$, $x_3 = 8 \dots$ Элементы списка удовлетворяют закону $x = f(5, 1)$.

11. Многочлен $P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ с целыми коэффициентами можно представить в виде списка, причем если $a_i = 0$, то соответствующее звено не включать в список. Построить многочлен p , являющийся производной многочлена q .

Модули

1. ОЛС в динамической памяти (базовый тип — *pointer*). Выделение памяти под информационную часть элемента ОЛС и запись в нее значения выполняется до обращения к процедуре *PutList*. При выполнении процедуры *GetList* память информационной части элемента не освобождается и ее адрес является выходным параметром.

Реализация на языке *Pascal*:

```

Unit List1;
  Interface
Const ListOk = 0;
      ListNotMem      = 1;
      ListUnder       = 2;
      ListEnd         = 3;
Type  BaseType        = Pointer;
      PtrEl   = ^Element;
      Element= Record
                          Data : BaseType;
                          Next : PtrEl;

      End;
      List= Record
          Start,Ptr : PtrEl;
          N : Word
      End;
Var ListError : 0..3;
Procedure InitList(var L:List);
Procedure PutList(var L:List; E:BaseType);
Procedure GetList(var L:List; var E:BaseType);
Procedure ReadList(var L:List; var E:BaseType);
Function  FullList(var L:List):boolean;
Function  EndList(var L:List):boolean;
Function  Count(var L:List):Word;
Procedure BeginPtr(var L:List);
Procedure EndPtr(var L:List);
Procedure MovePtr(var L:List);
Procedure MoveTo(var L:List; N:word);
Procedure DoneList(var L:List);
Procedure CopyList(var L1,L2:List);

```

Реализация на языке *C*:

```

#ifdef __LIST1_H
const ListOk = 0;
const ListNotMem = 1;
const ListUnder = 2;
const ListEnd = 3;
typedef void *BaseType;
typedef struct element *ptrel;
typedef struct element {basetype data;

```

```

                                ptrrel next;};
typedef struct List {ptrrel Start;
                                ptrrel ptr;
                                unsigned int N}

short ListError;
void InitList(List *L);
void PutList(List *L, BaseType E);
void GetList(List *L, BaseType *E);
void ReadList(List *L,BaseType *E);
int FullList(List *L);
int EndList(List *L);
unsigned int Count(List *L);
void BeginPtr(List *L);
void EndPtr(List *L);
void MovePtr(List *L);
void MoveTo(List *L, unsigned int n);
void DoneList(List *L);
void CopyList(List *L1,List *L2);
#endif

```

2. ОЛС в динамической памяти (базовый тип определяется задачей). Выделение памяти под информационную часть элемента ОЛС и запись в нее значения осуществляется при выполнении процедуры *PutList*. При выполнении процедуры *GetList* память, занимаемая элементом, освобождается.

Реализация на языке *Pascal*:

```

Unit List2;
Interface
Const ListOk = 0;
      ListNotMem = 1;
      ListUnder = 2;
      ListEnd = 3;
Type  BaseType = ...; { определить !!!}
      PtrEl    = ^Element;
      Element = Record
                                Data : BaseType;
                                Next : PtrEl;
      End;
      List = Record
                                Start,Ptr : PtrEl;
                                N : Word;

```

```

    End;
    Var ListError : 0..3;
    Procedure InitList(var L:List);
    Procedure PutList(var L:List; E:Basetype);
    Procedure GetList(var L:List; var E:BaseType);
    Procedure ReadList(var L:List; var E:BaseType);
    Function FullList(var L:List):boolean;
    Function EndList(var L:List):boolean;
    Function Count(var L:List):Word;
    Procedure BeginPtr(var L:List);
    Procedure EndPtr(var L:List);
    Procedure MovePtr(var L:List);
    Procedure MoveTo(var L:List; N:word);
    Procedure DoneList(var L:List);
    Procedure CopyList(var L1,L2:List);

```

Реализация на языке C:

```

#if !defined(__LIST2_H)
    const ListOk = 0;
    const ListNotMem = 1;
    const ListUnder = 2;
    const ListEnd = 3;
    typedef < определить > BaseType;
    typedef struct element *ptrel;
    typedef struct element {basetype data;
                           ptrel next;};
    typedef struct List {ptrel Start;
                        ptrel ptr;
                        unsigned int N}

    short ListError;
    void InitList(List *L);
    void PutList(List *L, BaseType E);
    void GetList(List *L, BaseType *E);
    void ReadList(List *L,BaseType *E);
    int FullList(List *L);
    int EndList(List *L);
    unsigned int Count(List *L);
    void BeginPtr(List *L);
    void EndPtr(List *L);
    void MovePtr(List *L);
    void MoveTo(List *L, unsigned int n);

```

```

void DoneList(List *L);
void CopyList(List *L1, List *L2);
#endif

```

3. ОЛС в динамической памяти (базовый тип — *pointer*). Выделение памяти под информационную часть элемента ОЛС и запись в нее значения происходит при выполнении процедуры *PutList*. При выполнении процедуры *GetList* память, занимаемая элементом, освобождается. Размер информационной части элемента задается при инициализации ОЛС и сохраняется в дескрипторе.

Реализация на языке *Pascal*:

```

Unit List3;
Interface
Const ListOk = 0;
      ListNotMem      = 1;
      ListUnder       = 2;
      ListEnd         = 3;
Type  BaseType       = Pointer;
      PtrEl = ^Element;
      Element = Record
                                Data : BaseType;
                                Next : PtrEl;
      End;
      List = Record
                                Start,Ptr : PtrEl;
                                N : Word; { длина списка          }
                                Size : Word { размер информационной }
                                { части элемента          }
      End;
Var ListError : 0..3;
Procedure InitList(var L:List; Size:Word);
Procedure PutList(var L:List; var E);
Procedure GetList(var L:List; var E);
Procedure ReadList(var L:List; var E);
Function FullList(var L:List):boolean;
Function EndList(var L:List):boolean;
Function Count(var L:List):Word;
Procedure BeginPtr(var L:List);
Procedure EndPtr(var L:List);
Procedure MovePtr(var L:List);
Procedure MoveTo(var L:List; N:word);
Procedure DoneList(var L:List);

```


Procedure CopyList(var L1,L2:List);

Реализация на языке C:

```
#if !defined(__LIST3_H)
const ListOk = 0;
const ListNotMem = 1;
const ListUnder = 2;
const ListEnd = 3;
typedef void* BaseType;
typedef struct element *ptrel;
typedef struct element {basetype data;
                        ptrel next;};
typedef struct List {ptrel Start;
                    ptrel ptr;
                    unsigned int N;//размер списка
                    unsigned int size;//размер информационной части эле-
мента
short ListError;
void InitList(List *L);
void PutList(List *L, BaseType E);
void GetList(List *L, BaseType *E);
void ReadList(List *L,BaseType *E);
int FullList(List *L);
int EndList(List *L);
unsigned int Count(List *L);
void BeginPtr(List *L);
void EndPtr(List *L);
void MovePtr(List *L);
void MoveTo(List *L, unsigned int n);
void DoneList(List *L);
void CopyList(List *L1,List *L2);
#endif
```

4. Элементы ОЛС находятся в массиве *MemList*, расположенном в статической памяти. Базовый тип — pointer. Каждый элемент массива имеет признак того, является ли он элементом ОЛС или «свободен». Выделение памяти под информационную часть элемента ОЛС и запись в нее значения выполняется до обращения к процедуре *PutList*. При выполнении процедуры *GetList* память информационной части элементане освобождается и ее адрес является выходным параметром.

Реализация на языке *Pascal*:

Unit List4;

Interface

$$Const\ ListOk = 0;$$
$$ListNotMem = 1;$$

```
ListUnder = 2;
```

```
ListEnd      = 3;
```

```
SizeList      = 100;
```

```
Type BaseType = Pointer;
```

Index = 0..*SizeList*;

$$PtrEl = Index;$$

Element = Record

Data : *BaseType*;

Next : PtrEl;

Flag : Boolean {TRUE, если элемент
 {принадлежит ОЛС }

End; {FALSE, если “свободен”}

List = Record

$$Start, Ptr : PtrEl;$$

N : Word

End;

Var MemList: array[Index] of Element;

ListError : 0..3;

Procedure InitList(var L:List);

Procedure PutList(var L:List; E:BaseType);

Procedure *GetList*(*var L:List; var E:BaseType*);

Function ReadList(var L:List):Pointer;

Function `FullList(var L:List):boolean;`

Function `EndList(var L:List):boolean;`

Function *Count*(*var L:List*):*Word*;

Procedure BeginPtr(var L:List);

Procedure EndPtr(var L:List);

Procedure MovePtr(var L:List);

Procedure MoveTo(var L:List; N:word);

Procedure DoneList(var L:List);

Procedure CopyList(var L1,L2:List);

Implementation

Procedure InitMem; forward; {устанавливает *Flag* каждого элемента в *FALSE*, вызывается в разделе операторов модуля}

Function EmptyMem: boolean; forward; { возвращает *TRUE*, если в массиве нет свободных элементов }

Function NewMem: word; forward; {возвращает номер свободного элемента}

Procedure DisposeMem(n:word); forward; {делает n-й элемент массива свободным}

Реализация на языке C:

```
#if !defined(__LIST4_H)
#define SizeList 100
const ListOk = 0;
const ListNotMem = 1;
const ListUnder = 2;
const ListEnd = 3;
typedef void *BaseType;
typedef unsigned ptrrel;
typedef struct element {basetype data;
    ptrrel next; /*flag=1 если элемент принадлежит ОЛС*/
    int flag;}; /*flag=0 если свободен
typedef struct List{ptrrel Start;
    ptrrel ptr;
    unsigned int N}
element MemList[SizeList];
short ListError;
void InitList(List *L)
void PutList(List *L, BaseType E)
void GetList(List *L, BaseType *E)
void ReadList(List *L,BaseType *E)
int FullList(List *L)
int EndList(List *L)
unsigned int Count(List *L)
void BeginPtr(List *L)
void EndPtr(List *L)
void MovePtr(List *L)
void MoveTo(List *L, unsigned int n)
void DoneList(List *L)
void CopyList(List *L1,List *L2)
void InitMem()/*присваивает Flag каждого элемента в 0*/
int EmptyMem() /*возвращает 1, если в массиве нет свободных элементов*/
unsigned NewMem()//возвращает номер свободного элемента
void DisposeMem(unsigned n) /*делает n-й элемент массива свободным*/
#endif
```

5. Элементы ОЛС находятся в массиве *MemList*, расположенном в статической памяти. Базовый тип зависит от задачи. «Свободные» элементы массива объединяются в список, на начало которого указывает поле-указатель первого элемента массива. Выделение памяти под информационную часть элемента ОЛС и запись в нее значения происходит при выполнении процедуры *PutList*. При выполнении процедуры *GetList* память, занимаемая элементом, освобождается.

Реализация на языке *Pascal*:

```
Unit List5;
Interface
Const ListOk = 0;
      ListNotMem = 1;
      ListUnder = 2;
      ListEnd = 3;
      SizeList = 100;
Type BaseType = ...; {определить !!!}
      Index = 0..SizeList;
PtrEl = Index;
      Element = Record Data : BaseType;
                      Next : PtrEl;

      End;
      List = Record
                      Start,Ptr : PtrEl;
                      N : Word

      End;
Var MemList: array[Index] of Element;
    ListError : 0..3;
Procedure InitList(var L:List);
Procedure PutList(var L:List; E:BaseType);
Procedure GetList(var L:List; var E:BaseType);
Procedure ReadList(var L:List; var E:BaseType);
Function FullList(var L:List):boolean;
Function EndList(var L:List):boolean;
Function Count(var L:List):Word;
Procedure BeginPtr(var L:List);
Procedure EndPtr(var L:List);
Procedure MovePtr(var L:List);
Procedure MoveTo(var L:List; N:word);
Procedure DoneList(var L:List);
Procedure CopyList(var L1,L2:List);
Implementation
```

Procedure InitMem; forward; {связывает все элементы массива в список свободных элементов}

Function EmptyMem: boolean; forward; {возвращает *TRUE*, если в массиве нет свободных элементов}

Function NewMem: word; forward; {возвращает номер свободного элемента и исключает его из ССЭ}

Procedure DisposeMem(n:word); forward; {делает n-й элемент массива свободным и включает его в ССЭ}

Реализация на языке C:

```
#if !defined(__LIST5_H)
#define SizeList 100
const ListOk = 0;
const ListNotMem = 1;
const ListUnder = 2;
const ListEnd = 3;
typedef <определить> BaseType;
typedef unsigned ptrrel;
typedef struct element {basetype data;
                        ptrrel next; };
typedef struct List {ptrrel Start;
                    ptrrel ptr;
                    unsigned int N}
element MemList[SizeList];
short ListError;
void InitList(List *L)
void PutList(List *L, BaseType E)
void GetList(List *L, BaseType *E)
void ReadList(List *L,BaseType *E)
int FullList(List *L)
int EndList(List *L)
unsigned int Count(List *L)
void BeginPtr(List *L)
void EndPtr(List *L)
void MovePtr(List *L)
void MoveTo(List *L, unsigned int n)
void DoneList(List *L)
void CopyList(List *L1,List *L2)
void InitMem()/*присваивает Flag каждого элемента в 0*/
int EmptyMem() /*возвращает 1, если в массиве нет свободных элемен-
тов*/
```

```

unsigned NewMem())/возвращает номер свободного элемента
void DisposeMem(unsigned n) /*делает n-й элемент массива свободным*/
#endif

```

6. ПЛС. Массив, на основе которого реализуется ПЛС, находится в статической памяти (базовый тип элемента — *pointer*). Выделение памяти под информационную часть элемента ПЛС и запись в нее значения происходит при выполнении процедуры *PutList*. При выполнении процедуры *GetList* память, занимаемая элементом, освобождается. Размер информационной части элемента задается при инициализации ПЛС и сохраняется в дескрипторе.

Реализация на языке *Pascal*:

```

Unit List6;
Interface
Const ListOk = 0;
      ListNotMem = 1;
      ListUnder = 2;
      ListEnd = 3;
Type  BaseType = Pointer;
      Index = 0..100;
      PtrEl = Index;
      List = Record
                                MemList: array[Index] of BaseType;
                                Ptr : PtrEl;
                                N : Word; { длина списка }
                                Size : Word { размер информационной }
                                { части элемента }
      End;
Var   ListError : 0..3;
Procedure InitList(var L:List; Size:Word);
Procedure PutList(var L:List; var E);
Procedure GetList(var L:List; var E);
Procedure ReadList(var L:List; var E);
Function FullList(var L:List):boolean;
Function EndList(var L:List):boolean;
Function Count(var L:List):Word;
Procedure BeginPtr(var L:List);
Procedure EndPtr(var L:List);
Procedure MovePtr(var L:List);
Procedure MoveTo(var L:List; N:word);
Procedure DoneList(var L:List);
Procedure CopyList(var L1,L2:List);

```

Реализация на языке C:

```

#if !defined(__LIST6_H)
#define SizeList 100
const ListOk = 0;
const ListNotMem = 1;
const ListUnder = 2;
const ListEnd = 3;
typedef void *BaseType;
typedef unsigned ptrrel;
typedef struct List {basetype MemList[SizeList];
                    ptrrel ptr;
                    unsigned int N;
                    unsigned int Size;};

short ListError;
void InitList(List *L)
void PutList(List *L, BaseType E)
void GetList(List *L, BaseType *E)
void ReadList(List *L,BaseType *E)
int FullList(List *L)
int EndList(List *L)
unsigned int Count(List *L)
void BeginPtr(List *L)
void EndPtr(List *L)
void MovePtr(List *L)
void MoveTo(List *L, unsigned int n)
void DoneList(List *L)
void CopyList(List *L1,List *L2)
#endif

```

7. ПЛС. Массив, на основе которого реализуется ПЛС, находится в динамической памяти (базовый тип элемента определяется задачей). Память под массив выделяется при инициализации ПЛС и количество элементов сохраняется в дескрипторе.

Реализация на языке *Pascal*:

```

Unit List7;
Interface
Const ListOk = 0;
      ListNotMem    = 1;
      ListUnder  = 2;
      ListEnd    = 3;

```

```

Type BaseType = ...; {определить !!!}
  Index = 0..65520 div sizeof(BaseType);
  TMemList = array[Index] of BaseType;
  PtrEl = Index;
  List = Record
    PMemList: ^TmemList;
    Ptr : PtrEl;
    N : Word; { длина списка }
    SizeМем : Word { размер массива }
  End;
Var ListError : 0..3;
Procedure InitList(var L:List; SizeMem:Word);
Procedure PutList(var L:List; var E:BaseType);
Procedure GetList(var L:List; var E:BaseType);
Procedure ReadList(var L:List; var E:BaseType);
Function FullList(var L:List):boolean;
Function EndList(var L:List):boolean;
Function Count(var L:List):Word;
Procedure BeginPtr(var L:List);
Procedure EndPtr(var L:List);
Procedure MovePtr(var L:List);
Procedure MoveTo(var L:List; N:word);
Procedure DoneList(var L:List);
Procedure CopyList(var L1,L2:List);

```

Реализация на языке C:

```

#if !defined(__LIST7_H)
#define Index 1000
const ListOk = 0;
const ListNotMem = 1;
const ListUnder = 2;
const ListEnd = 3;
typedef <определить> BaseType;
typedef basetype TMemList[Index];
typedef unsigned ptrrel;
typedef struct List {TMemList* PMemList;
  ptrrel ptr;
  unsigned int N; // длина списка
  unsigned int SizeMem;}; // размер массива
short ListError;
void InitList(List *L,unsigned SizeMem)

```



```

void PutList(List *L, BaseType E)
void GetList(List *L, BaseType *E)
void ReadList(List *L, BaseType *E)
int FullList(List *L)
int EndList(List *L)
unsigned int Count(List *L)
void BeginPtr(List *L)
void EndPtr(List *L)
void MovePtr(List *L)
void MoveTo(List *L, unsigned int n)
void DoneList(List *L)
void CopyList(List *L1, List *L2)
#endif

```

8. ПЛС. Массив, на основе которого реализуется ПЛС, находится в динамической памяти (базовый тип элемента — Pointer). Память под массив выделяется при инициализации ПЛС и количество элементов сохраняется в дескрипторе. Выделение памяти под информационную часть элемента ПЛС и запись в нее значения происходит при выполнении процедуры *PutList*. При выполнении процедуры *GetList* память, занимаемая информационной частью элемента, освобождается. Размер информационной части элемента задается при инициализации ПЛС и сохраняется в дескрипторе.

Реализация на языке *Pascal*:

```

Unit List8;
Interface
Const ListOk = 0;
      ListNotMem = 1;
      ListUnder = 2;
      ListEnd = 3;
Type BaseType = Pointer;
Index = 0..65520 div sizeof(BaseType);
TMemList = array[Index] of BaseType;
PtrEl = Index;
List = Record
    PMemList: ^TmemList;
    Ptr : PtrEl;
    N : Word; { длина списка }
    SizeMem : Word { размер массива }
    SizeEl : Word { размер элемента}
End;
Var ListError : 0..3;

```

```

Procedure InitList(var L:List; SizeMem, SizeEl:Word);
Procedure PutList(var L:List; var E);
Procedure GetList(var L:List; var E);
Procedure ReadList(var L:List; var E);
Function FullList(var L:List):boolean;
Function EndList(var L:List):boolean;
Function Count(var L:List):Word;
Procedure BeginPtr(var L:List);
Procedure EndPtr(var L:List);
Procedure MovePtr(var L:List);
Procedure MoveTo(var L:List; N:word);
Procedure DoneList(var L:List);
Procedure CopyList(var L1,L2:List);

```

Реализация на языке C:

```

#if !defined(__LIST8_H)
#define SizeList 100
#define Index 1000
const ListOk = 0;
const ListNotMem = 1;
const ListUnder = 2;
const ListEnd = 3;
typedef void *BaseType;
typedef basetype TMemList[Index];
typedef unsigned ptrrel;
typedef struct List {TMemList* PMemList;
                    ptrrel ptr;
                    unsigned int N; // длина списка
                    unsigned int SizeMem;}; // размер массива

short ListError;
void InitList(List *L,unsigned SizeMem)
void PutList(List *L, BaseType E)
void GetList(List *L, BaseType *E)
void ReadList(List *L,BaseType *E)
int FullList(List *L)
int EndList(List *L)
unsigned int Count(List *L)
void BeginPtr(List *L)
void EndPtr(List *L)
void MovePtr(List *L)
void MoveTo(List *L, unsigned int n)

```

```
void DoneList(List *L)
void CopyList(List *L1, List *L2)
#endif
```

Назначение процедур и функций

InitList — инициализация списка.
PutList — включение элемента в список.
GetList — исключение элемента из списка.
ReadList — чтение элемента списка.
EmptyList — проверка: свободен ли список.
EndList — проверка: является ли элемент последним.
Count — возвращает количество элементов в списке.
BeginPtr — установка в начало списка.
EndPtr — установка в конец списка.
MovePtr — переход к следующему элементу.
MoveTo — переход к n -му элементу.
DoneList — удаление списка.
CopyList — копирование списка $L1$ в список $L2$.

Содержание отчета

1. Тема лабораторной работы.
2. Цель работы.
3. Характеристика СД типа «линейный список» (п.1 задания).
4. Индивидуальное задание.
5. Текст модуля для реализации СД типа «линейный список», текст программы для отладки модуля, тестовые данные результат работы программы.
6. Текст программы для решения задачи с использованием модуля, тестовые данные, результат работы программы.

Теоретические сведения

Линейный список (ЛС) — это конечная последовательность однотипных элементов (узлов).

Количество элементов в последовательности называется длиной списка, причем длина в процессе работы программы может изменяться, поэтому ЛС — динамическая структура.

Над СД ЛС определены следующие основные операции:

1. Инициализация.

2. Включение элемента.
3. Исключение элемента.
4. Чтение текущего элемента.
5. Переход в начало списка.
6. Переход в конец списка.
7. Переход к следующему элементу.
8. Переход к i -му элементу.
9. Определение длины списка.
10. Уничтожение списка.

Кардинальное число СД ЛС определяется по формуле:

$$CAR(LC) = CAR(BaseType)^0 + CAR(BaseType)^1 + \dots + CAR(BaseType)^{max},$$

где $CAR(BaseType)$ — кардинальное число элемента ЛС типа $BaseType$, max — максимальное количество элементов в ЛС (не всегда определено, т.к. может зависеть от объема свободной динамической памяти).

На *абстрактном* уровне ЛС представляет собой линейную структуру — последовательность.

На *физическом* уровне ЛС может быть реализован последовательной или связной схемой хранения.

Располагаться ЛС может в статической или динамической памяти. ЛС, реализованный последовательной схемой хранения, называется последовательным линейным списком (ПЛС). ЛС, реализованный связной схемой хранения, называется связным линейным списком (СЛС). Для реализации ПЛС используется вспомогательная встроенная СД — массив, элементы которого могут быть элементами ПЛС. Доступ к элементам ПЛС — прямой. Для реализации СЛС используется множество связанных элементов типа «запись», связь между которыми устанавливается с помощью поля-указателя. Элементы СЛС могут располагаться в массиве или в динамической памяти. СЛС подразделяются на односвязные линейные списки (ОЛС) и двусвязные линейные списки (ДЛС). В ОЛС каждый элемент состоит из информационного поля (поле данных) и одного поля-указателя, содержащего адрес следующего элемента. В ДЛС каждый элемент состоит из информационного поля и двух полей указателей, одно из которых содержит адрес следующего элемента в списке, а другое — предыдущего. Реализуется ОЛС, как правило, с первым фиктивным элементом, поле данных которого не используется, а ДЛС — с первым и последним фиктивным элементом. Введение фиктивных элементов позволяет упростить реализацию некоторых операций над ЛС.

Рассмотрим некоторые принципы реализации ЛС.

1. Реализация ПЛС

1.1. Последовательному линейному списку можно поставить в соответствие дескриптор, который состоит из 3-х полей:

- 1 — массив, на основе которого реализуется ПЛС;
- 2 — индекс текущего элемента;
- 3 — длина ПЛС.

Дескриптор располагается в статической памяти в виде переменной соответствующего типа. Массив, на основе которого реализуется ПЛС, также располагается в статической памяти.

1.2. Дескриптор ПЛС состоит из 4-х полей:

- 1 — указатель на массив, на основе которого реализуется ПЛС;
- 2 — количество элементов массива;
- 3 — индекс текущего элемента;
- 4 — длина ПЛС.

Дескриптор располагается в статической памяти в виде переменной соответствующего типа. Массив, на основе которого реализуется ПЛС, располагается в динамической памяти. Память под массив выделяется при инициализации ПЛС и количество элементов этого массива заносится во второе поле.

Элементы ПЛС могут находиться непосредственно в элементах массива, или в динамической памяти, а их адреса — в массиве.

2. Реализация ОЛС

2.1. Элементы ОЛС располагаются в динамической памяти. В статической памяти находится дескриптор ОЛС, состоящий из 3-х полей:

- 1 — указатель на фиктивный элемент ОЛС;
- 2 — указатель на текущий элемент;
- 3 — длина ОЛС.

Адрес фиктивного элемента определяется при инициализации. Элемент ОЛС может содержать либо поле данных, либо адрес данных.

2.2. Элементы ОЛС располагаются в массиве. Элемент массива может содержать элемент ОЛС или быть «свободным». При включении элемента в ОЛС необходимо найти «свободный» элемент, сделать его «занятым» и включить в ОЛС. При исключении элемента нужно исключить элемент из ОЛС путем переопределения указателя предшествующего элемента на последующий и «освободить» исключенный элемент. «Свободные» элементы массива могут иметь специальный признак, отличающий их от элементов ОЛС, или могут быть объединены в *список свободных элементов* (ССЭ), на первый элемент которого указывает поле-указатель первого эле-

мента массива. При включении элемента в ОЛС берется первый элемент ССЭ, а исключаемый элемент заносится в начало ССЭ.

В статической памяти находится дескриптор ОЛС, состоящий из трех полей:

- 1 — указатель на фиктивный элемент ОЛС;
- 2 — указатель на текущий элемент;
- 3 — длина ОЛС.

Адрес фиктивного элемента определяется при инициализации.

Элемент ОЛС может содержать либо поле данных, либо адрес данных.

3. Реализация ДЛС

3.1. В статической памяти находится дескриптор ДЛС, состоящий из 3-х полей:

- 1 — указатель на первый фиктивный элемент ДЛС;
- 2 — указатель на последний фиктивный элемент ДЛС;
- 3 — указатель на текущий элемент;
- 4 — длина ДЛС.

Адреса фиктивных элементов определяется при инициализации.

Элемент ДЛС может содержать либо поле данных, либо адрес данных и располагаться либо в динамической памяти, либо в массиве аналогично элементам ОЛС.

В заключении выполним сравнительный анализ последовательных и связанных списков по эффективности хранения и выполнения операций.

1. СЛС требуют дополнительной памяти для связей.

2. Легко исключить элемент СЛС. Для исключения элемента из ОЛС достаточно лишь установить поле-указатель предшествующего элемента на последующий.

3. Время выполнения операции не зависит от положения исключаемого элемента. При исключении элемента из ПЛС обычно требуется перемещение элементов, расположенных за исключаемым, к началу списка. Чем ближе к началу списка расположен исключаемый элемент, тем больше времени требуется на выполнение операции.

4. Легко включить элемент в СЛС. Для включения элемента в ОЛС необходимо установить поле-указатель включаемого элемента на элемент, следующий за текущим, и переопределить поле-указатель текущего элемента (установить его на включаемый). Время выполнения операции не зависит от положения текущего элемента. При включении элемента в ПЛС требуется перемещения элементов, расположенных за текущим, к концу списка. Чем ближе к началу списка расположен текущий элемент, тем больше времени требуется на выполнение операции.

5. В ПЛС быстрее, чем в СЛС, выполняется обращения к i -му элементу списка, т.к. доступ к элементам ПЛС прямой, а к элементам СЛС — последовательный.

6. При использовании СЛС упрощается задача объединение двух списков или разбиение списков на части.

Контрольные вопросы

1. Что такое линейный список?
2. Определите характер изменчивости линейного списка.
3. Назовите основные операции над линейным списком.
4. Что собой представляет линейный список на абстрактном уровне?
5. Чем отличается последовательный линейный список от массива?
6. Что такое односвязный линейный список?
7. Какую структуру имеет элемент односвязного линейного списка?
8. Что такое двусвязный линейный список?
9. Какую структуру имеет элемент двусвязного линейного списка?
10. Как можно реализовать связный линейный список на массиве?
11. Какую структуру может иметь дескриптор линейного списка?
12. Зачем нужны фиктивные элементы в связных линейных списках?
13. Определите порядок функции временной сложности операции включения элемента в последовательный и связный линейный список.
14. Определите порядок функции временной сложности операции исключения элемента в последовательный и связный линейный список.
15. Определите порядок функции временной сложности операции перехода в начало последовательного и связного линейного списка.
16. Определите порядок функции временной сложности операции перехода в конец последовательного и связного линейного списка.
17. Определите порядок функции временной сложности операции перехода к следующему элементу последовательного и связного линейного списка.
18. Определите порядок функции временной сложности операции перехода к i -му элементу последовательного и связного линейного списка.
19. Определите порядок функции временной сложности линейного поиска в последовательном и связном линейном списке.
20. Какой алгоритм поиска целесообразно использовать в упорядоченном последовательном и связном линейном списке?
21. Предложите различные варианты реализации операции обмена соседних элементов в связном линейном списке.
22. Выполните сравнительный анализ алгоритмов сортировки связных линейных списков.

Лабораторная работа № 6

Структуры данных «стек» и «очередь» (Pascal/C)

Цель работы: изучить СД типа «стек» и «очередь», научиться их программно реализовывать и использовать.

З а д а н и е

1. Для СД типа «стек» и «очередь» определить:
 - 1.1. Абстрактный уровень представления СД:
 - 1.1.1. Характер организованности и изменчивости.
 - 1.1.2. Набор допустимых операций.
 - 1.2. Физический уровень представления СД:
 - 1.2.1. Схему хранения.
 - 1.2.2. Объем памяти, занимаемый экземпляром СД.
 - 1.2.3. Формат внутреннего представления СД и способ его интерпретации.
 - 1.2.4. Характеристику допустимых значений.
 - 1.2.5. Тип доступа к элементам.
 - 1.3. Логический уровень представления СД.
 Способ описания СД и экземпляра СД на языке программирования.
2. Реализовать СД типа «стек» и «очередь» в соответствии с вариантом индивидуального задания в виде модуля.
3. Разработать программу, моделирующую вычислительную систему с постоянным шагом по времени (дискретное время) в соответствии с вариантом индивидуального задания (табл.16) с использованием модуля, полученного в результате выполнения пункта 2. Результат работы программы представить в виде таблицы 15. В первом столбце указывается время моделирования 0, 1, 2, ..., N . Во втором — для каждого момента времени указываются имена объектов (очереди — F_1, F_2, \dots, F_N ; стеки — S_1, S_2, \dots, S_M ; процессоры — P_1, P_2, \dots, P_K), а в третьем — задачи (имя, время), находящиеся в объектах.

Таблица 15

Результаты работы программы

Время	Объекты	Задачи
0	F_1	(имя, время), (имя, время), .., (имя, время)
	:	: : :
	F_N	(имя, время), (имя, время), .., (имя, время)
	S_1	(имя, время), (имя, время), .., (имя, время)
	:	: : :
	S_M	(имя, время), (имя, время), .., (имя, время)
	P_1	(имя, время)
	:	:
1	P_K	(имя, время)
	F_1	(имя, время), (имя, время), .., (имя, время)
	:	: : :
	F_N	(имя, время), (имя, время), .., (имя, время)
	S_1	(имя, время), (имя, время), .., (имя, время)
	:	: : :
	S_M	(имя, время), (имя, время), .., (имя, время)
	P_1	(имя, время)
:	:	:
	:	:

Таблица 16

Варианты индивидуальных заданий

Номер варианта	Номер модуля для стека	Номер модуля для очереди	Номер задачи
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
6	6	6	6
7	7	7	7
8	8	8	8
9	9	9	9
10	10	10	10
11	11	10	11
12	1	2	4
13	2	3	5

14	3	4	6
15	4	5	7
16	5	6	8
17	6	7	9
18	7	8	10
19	8	9	11
20	9	10	1
21	10	11	2
22	11	10	3
23	1	3	4
24	2	4	5
25	3	5	6
26	4	6	7
27	5	7	8
28	6	8	9
29	7	9	10
30	8	10	11

Варианты задач

1. Система состоит из процессора P , трех очередей F_0 , F_1 , F_2 и стека S (рис.9). В систему поступают запросы. Запрос можно представить записью.

Реализация на языке *Pascal*:

Type

TInquiry = record

Name: String[10]; {имя запроса}

Time: Word; {время обслуживания}

P: Byte; {приоритет задачи 0 — высший,
1 — средний, 2 — низший}

end;

Реализация на языке *C*:

typedef struct TInquiry

{

char Name[10]; // Имя запроса

unsigned Time; // Время обслуживания

char P; /* Приоритет задачи: 0 — высший,
1 — средний, 2 — низший */

};

Поступающие запросы ставятся в соответствующие приоритетам очереди. Сначала обрабатываются задачи из очереди F_0 . Если она пуста, можно обрабатывать задачи из очереди F_1 . Если и она пуста, то можно обрабатывать задачи из очереди F_2 . Если все очереди пусты, то система находится в ожидании поступающих задач (процессор свободен), либо в режиме обработки предыдущей задачи (процессор занят). Если поступает задача с более высоким приоритетом, чем обрабатываемая в данный момент, то обрабатываемая помещается в стек и может обрабатываться тогда и только тогда, когда все задачи с более высоким приоритетом уже обработаны.

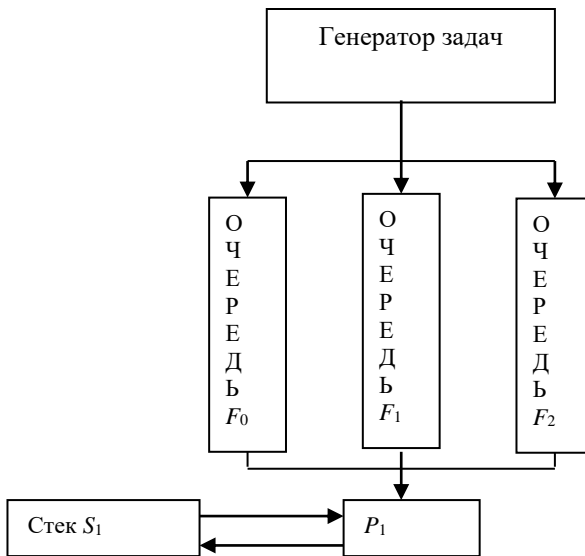


Рис.9. Система задач 1 — 3

2. Система состоит из процессора P , трех очередей F_0 , F_1 , F_2 и стека S (см. рис.9). В систему поступают запросы. Запрос можно представить записью.

Реализация на языке *Pascal*:

Type

TInquiry = record

Name: String[10]; {имя запроса}

Time: Word; {время обслуживания}

P: Byte; {приоритет задачи 0 — высший, 1 — средний, 2 — низший}

end;

Реализация на языке C:

```
typedef struct TInquiry
{
    char Name[10]; // имя запроса
    unsigned Time; // время обслуживания
    char P; /* приоритет задачи: 0 — высший, 1 — средний, 2 — низший */
};
```

Поступающие запросы ставятся в соответствующие приоритетам очереди. Сначала обрабатываются задачи из очереди F_0 . Если она пуста, можно обрабатывать задачи из очереди F_1 . Если она пуста, то можно обрабатывать задачи из очереди F_2 . Если все очереди пусты, то система находится в ожидании поступающих задач (процессор свободен), либо в режиме обработки предыдущей задачи (процессор занят). Если обрабатывается задача с низшим приоритетом и поступает задача с более высоким приоритетом, то обрабатываемая помещается в стек и может обрабатываться тогда и только тогда, когда все задачи с более высоким приоритетом уже обработаны.

3. Система состоит из процессора P , трех очередей F_0 , F_1 , F_2 и стека S (см. рис.9). В систему поступают запросы. Запрос можно представить записью.

Реализация на языке Pascal:

```
Type
    TInquiry= record
        Name: String[10]; {имя запроса}
        Time: Word; {время обслуживания}
        P: Byte; {приоритет задачи 0 — высший,
                1 — средний, 2 — низший}
    end;
```

Реализация на языке C:

```
typedef struct TInquiry
{
    char Name[10]; // Имя запроса
    unsigned Time; // Время обслуживания
    char P; /* Приоритет задачи: 0 — высший,
            1 — средний, 2 — низший */
};
```

Поступающие запросы ставятся в соответствующие приоритетам очереди. Сначала обрабатываются задачи из очереди F_0 . Если она пуста, можно обрабатывать задачи из очереди F_1 . Если и она пуста, то можно обрабатывать задачи из очереди F_2 . Если все очереди пусты, то система находится в ожидании поступающих задач (процессор свободен), либо в режиме обработки предыдущей задачи (процессор занят). Если поступает задача с более высоким приоритетом, чем обрабатываемая в данный момент, то обрабатываемая помещается в стек, если она выполнена менее чем на половину по времени, и может обрабатываться тогда и только тогда, когда все задачи с более высоким приоритетом уже обработаны.

4. Система состоит из двух процессоров P_1 и P_2 , трех очередей F_0 , F_1 , F_2 и двух стеков S_1 и S_2 (рис.10).

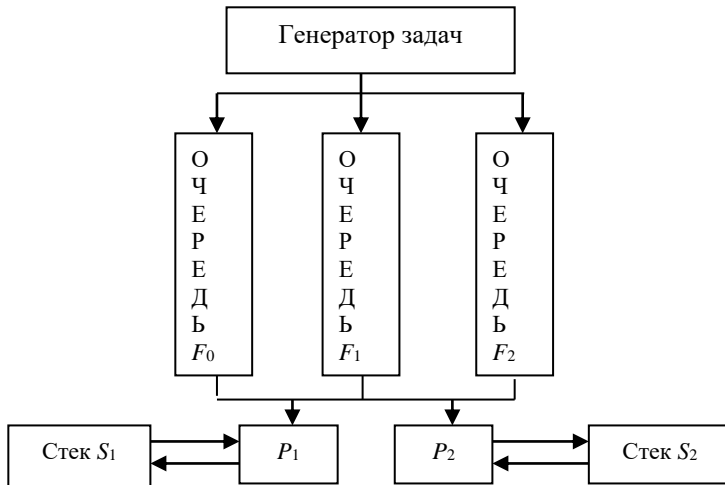


Рис.10. Система задачи 4

В систему поступают запросы. Запрос можно представить записью.

Реализация на языке *Pascal*:

Type

TInquiry = record

Name: String[10]; {имя запроса}

Time: Word; {время обслуживания}

P: Byte; {приоритет задачи 0 — высший,
1 — средний, 2 — низший}

end;

Реализация на языке C:

```
typedef struct TInquiry
```

```
{
    char Name[10]; // имя запроса
    unsigned Time; // время обслуживания
    char P; /* приоритет задачи: 0 — высший,
            1 — средний, 2 — низший */
};
```

Поступающие запросы ставятся в соответствующие приоритетам очереди. Сначала обрабатываются задачи из очереди F_0 . Задача из очереди F_0 поступает в свободный процессор P_1 или P_2 , если оба свободны, то в P_1 . Если очередь F_0 пуста, то обрабатываются задачи из очереди F_1 . Задача из очереди F_1 поступает в свободный процессор P_1 или P_2 , если оба свободны, то в P_1 . Если очереди F_0 и F_1 пусты, то обрабатываются задачи из очереди F_2 . Задача из очереди F_2 поступает в свободный процессор P_1 или P_2 , если оба свободны, то в P_1 . Если процессоры заняты и поступает задача с более высоким приоритетом, чем обрабатываемая в одном из процессоров, то задача из процессора помещается в соответствующий стек, а поступающая — в процессор. Задача из стека помещается в соответствующий процессор, если он свободен, и очереди с задачами более высокого приоритета пусты.

5. Система состоит из трех процессоров P_1, P_2, P_3 , очереди F , стека S и распределителя R (рис.11).

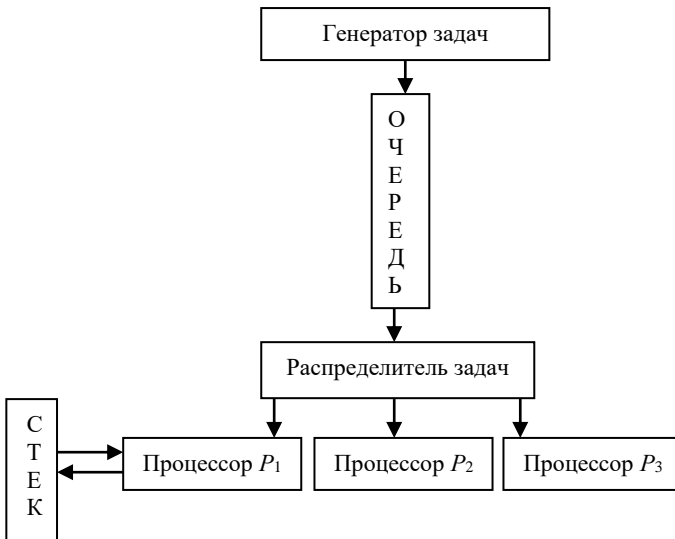


Рис.11. Система задач 5 и 6

В систему поступают запросы на выполнение задач трех типов — T_1 , T_2 и T_3 , каждая для своего процессора. Запрос можно представить записью.

Реализация на языке *Pascal*:

Type

TInquiry = *record*

Name: *String*[10]; {имя запроса}

Time: *Word*; {время обслуживания}

T: *Byte*; {тип задачи 1 — T_1 , 2 — T_2 , 3 — T_3 }

end;

Реализация на языке *C*:

typedef struct TInquiry

{

char Name[10]; // Имя запроса

unsigned Time; // Время обслуживания

char T; // Тип задачи: 1 — T_1 , 2 — T_2 , 3 — T_3

};

Поступающие запросы ставятся в очередь. Если в «голове» очереди находится задача T_i и процессор P_i свободен, то распределитель ставит задачу на выполнение в процессор P_i , а если процессор P_i занят, то распределитель отправляет задачу в стек и из очереди извлекается следующая задача. Если в вершине стека находится задача, процессор которой в данный момент свободен, то эта задача извлекается и отправляется на выполнение.

6. Система состоит из трех процессоров P_1 , P_2 , P_3 , очереди F , стека S и распределителя R (рис.11). В систему поступают запросы на выполнение задач трех типов — T_1 , T_2 и T_3 , каждая для своего процессора. Запрос можно представить записью.

Реализация на языке *Pascal*:

Type

TInquiry = *record*

Name: *String*[10]; {имя запроса}

Time: *Word*; {время обслуживания}

T: *Byte*; {тип задачи 1 — T_1 , 2 — T_2 , 3 — T_3 }

end;

Реализация на языке *C*:

typedef struct TInquiry

{

char Name[10]; // Имя запроса

unsigned Time; // Время обслуживания

char T; // Тип задачи: 1 — T_1 , 2 — T_2 , 3 — T_3

};

Поступающие запросы ставятся в очередь. Если в «голове» очереди находится задача T_i и процессор P_i свободен, то распределитель ставит задачу на выполнение в процессор P_i , а если процессор P_i занят, то распределитель отправляет задачу в стек и из очереди извлекается следующая задача. Задача из стека поступает в соответствующий ей свободный процессор только тогда, когда очередь пуста.

7. Система состоит из двух процессоров P_1 и P_2 , трех очередей F_1, F_2, F_3 и стека (рис.12).

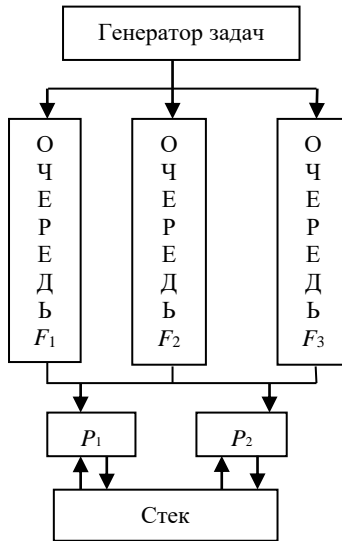


Рис.12. Система задач 7 и 10

В систему могут поступать запросы на выполнение задач трех типов — T_1, T_2, T_3 . Задача типа T_1 может выполняться только процессором P_1 . Задача типа T_2 может выполняться только процессором P_2 . Задача типа T_3 может выполняться любым процессором. Запрос можно представить записью.

Реализация на языке *Pascal*:

Типе

TInquiry = record

Name: String[10]; {имя запроса}

Time: Word; {время обслуживания}

T: Byte; {тип задачи 1 — T_1 , 2 — T_2 , 3 — T_3 }

end;

Реализация на языке C:

```
typedef struct TInquiry
```

```
{
    char Name[10]; // имя запроса
    unsigned Time; // время обслуживания
    char T; // тип задачи: 1 —  $T_1$ , 2 —  $T_2$ , 3 —  $T_3$ 
};
```

Поступающие запросы ставятся в соответствующие типам задач очереди. Если очередь F_1 не пуста и процессор P_1 свободен, то задача из очереди F_1 поступает на обработку в процессор P_1 . Если процессор P_1 обрабатывает задачу типа T_3 , а процессор P_2 свободен и очередь F_2 пуста, то задача из процессора P_1 поступает в процессор P_2 , а задача из очереди F_1 в процессор P_1 , если же процессор P_2 занят или очередь F_2 не пуста, то задача из процессора P_1 помещается в стек.

Если очередь F_2 не пуста и процессор P_2 свободен, то задача из очереди F_2 поступает на обработку в процессор P_2 . Если процессор P_2 обрабатывает задачу типа T_3 , а процессор P_1 свободен и очередь F_1 пуста, то задача из процессора P_2 поступает в процессор P_1 , а задача из очереди F_2 — в процессор P_2 , если же процессор P_1 занят или очередь F_1 не пуста, то задача из процессора P_1 помещается в стек.

Если очередь F_3 не пуста и процессор P_1 свободен, и очередь F_1 пуста или свободен процессор P_2 и очередь F_2 пуста, то задача из очереди F_3 поступает на обработку в свободный процессор. Задача из стека поступает на обработку в свободный процессор P_1 , если очередь F_1 пуста, или в свободный процессор P_2 , если очередь F_2 пуста.

8. Система состоит из двух процессоров P_1 и P_2 и двух очередей F_1 , F_2 и стека S (рис.13). В систему могут поступать запросы на выполнение задач двух типов — T_1 и T_2 . Задача типа T_1 может выполняться только процессором P_1 . Задача типа T_2 может выполняться любым процессором. Запрос можно представить записью.

Реализация на языке Pascal:

```
Type
```

```
TInquiry= record
    Name: String[10]; {имя запроса}
    Time: Word; {время обслуживания}
    T: Byte; {тип задачи 1 —  $T_1$ , 2 —  $T_2$ }
end;
```

Реализация на языке C:

```
typedef struct TInquiry
{
    char Name[10]; // имя запроса
    unsigned Time; // время обслуживания
    char T; // тип задачи 1 —  $T_1$ , 2 —  $T_2$ 
};
```

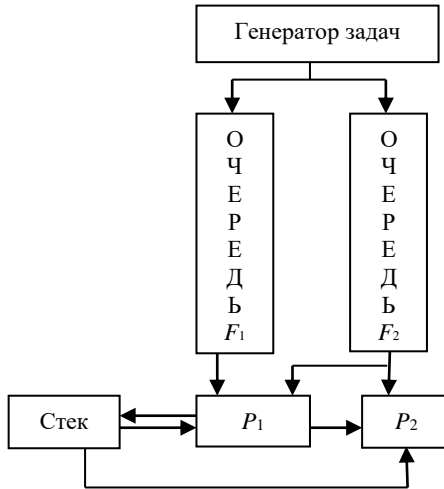


Рис.13. Система задачи 8

Поступающие запросы ставятся в соответствующие типам задач очереди. Если очередь F_1 не пуста и процессор P_1 свободен, то задача из очереди F_1 поступает на обработку в процессор P_1 . Если процессор P_1 обрабатывает задачу типа T_2 , а процессор P_2 свободен, то задача из процессора P_1 поступает в процессор P_2 , а задача из очереди F_1 в процессор P_1 , если же процессор P_2 занят, то задача из процессора P_1 помещается в стек.

Если очередь F_2 не пуста и процессор P_2 свободен, то задача из очереди F_2 поступает на обработку в процессор P_2 . Если процессор P_2 занят, а процессор P_1 свободен и очередь F_1 пуста, то задача из очереди F_2 поступает в процессор P_1 , а задача из стека поступает на обработку в свободный процессор P_2 , если F_2 пуста, или в свободный процессор P_1 , если очередь F_1 пуста и задачу нельзя поместить в процессор P_2 .

9. Система состоит из двух процессоров P_1 и P_2 и трех очередей F_1 , F_2 , F_3 и стека (рис.14).

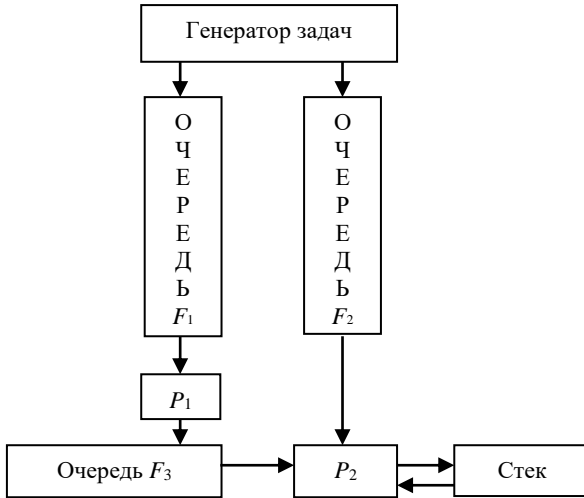


Рис.14. Система задачи 9

В систему могут поступать запросы на выполнение задач двух типов — T_1 и T_2 . Задача типа T_2 обрабатывается процессором P_2 . Задача типа T_1 сначала обрабатывается процессором P_1 , затем результат обработки (T_1') обрабатывается процессором P_2 .

Запрос можно представить записью.

Реализация на языке *Pascal*:

Type

TInquiry = record

Name: String[10]; {имя запроса}

T: Byte; {тип запроса}

Time1: Word; {время обработки
процессором P_1 . Для задач
типа T_2 *Time1*=0}

Time2: Word; {время обработки
процессором P_2 }

end;

Реализация на языке C:

```
typedef struct TInquiry
{
    char Name[10]; // имя запроса
    char T; // тип запроса
    unsigned Time1; /* время обработки
                     процессором P1. Для задач
                     типа T2 Time1=0 */
    unsigned Time2; /* время обработки
                     процессором P2 */
};
```

Поступающие запросы на выполнение задач типа T_1 и T_2 ставятся в соответствующие типам задач очереди. Результат обработки T_1 задачи T_1 процессором P_1 ставится в очередь F_3 . Если очередь F_1 не пуста и процессор P_1 свободен, то задача из очереди F_1 поступает на обработку в процессор P_1 .

Если очередь F_3 не пуста и процессор P_2 свободен, то задача из очереди F_3 поступает на обработку в процессор P_2 . Если процессор P_2 занят выполнением задачи типа T_2 , то она помещается в стек, а задача из очереди F_3 — в процессор P_2 . Задача из стека возвращается в процессор P_2 , если очередь F_3 пуста. Задача из очереди F_2 поступает на обработку в процессор P_2 , если он свободен и очередь F_3 и стек пусты.

10. Система состоит из двух процессоров P_1 и P_2 , трех очередей F_1 , F_2 , F_3 и стека (см. рис.12). В систему поступают запросы. Запрос можно представить записью.

Реализация на языке Pascal:

Type

```
TInquiry= record
    Name: String[10]; {имя запроса}
    Time: Word; {время обслуживания}
    P: Byte; {приоритет задачи 0 — высший,
              1 — средний, 2 — низший}
end;
```

Реализация на языке C:

```
typedef struct TInquiry
{
    char Name[10]; // имя запроса
    unsigned Time; // время обслуживания
    char P; /* приоритет задачи: 0 — высший,
             1 — средний, 2 — низший */
};
```

Поступающие запросы ставятся в соответствующие приоритетам очереди. Сначала обрабатываются задачи из очереди F_1 . Задача из очереди F_1 поступает в свободный процессор P_1 или P_2 , если оба свободны, то в P_1 . Если очередь F_1 пуста, то обрабатываются задачи из очереди F_2 . Задача из очереди F_2 поступает в свободный процессор P_1 или P_2 , если оба свободны, то в P_1 . Если очереди F_1 и F_2 пусты, то обрабатываются задачи из очереди F_3 . Задача из очереди F_3 поступает в свободный процессор P_1 или P_2 , если оба свободны, то в P_2 . Если процессоры заняты и поступает задача с более высоким приоритетом, чем обрабатываемая в одном из процессоров, то задача из процессора помещается в стек, а поступающая — в процессор. Задача из стека поступает в один из освободившихся процессоров, если все задачи с более высоким приоритетом уже обработаны.

11. Система состоит из двух процессоров P_1 и P_2 , двух стеков S_1 и S_2 и четырех очередей F_1, F_2, F_3, F_4 (рис.15).

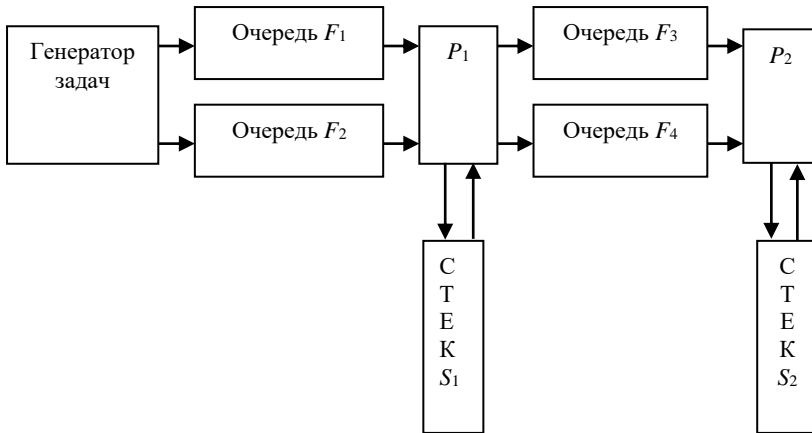


Рис.15. Система задачи 11

В систему могут поступать запросы на выполнение задач двух приоритетов — высший (1) и низший (2). Задачи сначала обрабатываются процессором P_1 , затем P_2 . Запрос можно представить записью.

Реализация на языке *Pascal*:

Type

TInquiry = record

Name: String[10]; {имя запроса}

P: Byte; {приоритет}

Time1: Word; {время выполнения задачи процессором P_1 }

Time2: Word; {время выполнения задачи процессором P_2 }

end;

Реализация на языке C:

```
typedef struct TInquiry
{
    char Name[10]; // Имя запроса
    char P; // Приоритет
    unsigned Time1; /* Время выполнения
                     задачи процессором P1 */
    unsigned Time2; /* Время выполнения
                     задачи процессором P2 */
};
```

Запросы на выполнение задач высшего приоритета ставятся в очередь F_1 , а поступающие с процессора P_1 — в очередь F_3 . Запросы на выполнение задач низшего приоритета, поступающие с генератора задач, ставятся в очередь F_2 , а поступающие с процессора P_1 — в очередь F_4 .

Процессор P_1 обрабатывает запросы из очередей F_1 и F_2 , а процессор P_2 — из очередей F_3 и F_4 . Процессор сначала обрабатывает задачи из очереди задач с высшим приоритетом, затем из очереди задач с низшим приоритетом. Если процессор выполняет задачу с низшим приоритетом и приходит запрос на выполнение задачи с высшим приоритетом, то выполняемая задача помещается в соответствующий процессору стек, а пришедшая задача — в процессор. Задача из стека возвращается в процессор, если все задачи большего приоритета обработаны.

Модули для реализации стека

1. Стек на массиве в статической памяти.

Реализация на языке Pascal:

```
Unit Stack1;
Interface
Const
    StackSize = 1000;
    StackOk   = 0;
    StackOver = 1;
    StackUnder = 2;
var
    StackError: 0..2;
Type
    Index   = 0..StackSize;
    BaseType = ...; {определить тип элемента стека}
    Stack = record
```

```

    Buf: array[Index]of BaseType;
    Uk: Index; {указывает на элемент, являющийся
                вершиной стека}
end;
procedure InitStack(var s:Stack); {инициализация стека}
function EmptyStack(var s:Stack):boolean;{стек пуст}
procedure PutStack(var s:Stack;El:BaseType); {поместить
                                              элемент в стек}
procedure GetStack(var s:Stack;var El:BaseType);
    {извлечь элемент из стека}
procedure ReadStack(const s:Stack;var El: BaseType);
    {прочитать элемент из вершины стека}

```

Реализация на языке C:

```

#ifdef __STACK1_H
#define __STACK1_H
const StackSize = 1000;
const StackOk = 0;
const StackOver = 1;
const StackUnder = 2;
int StackError; // Переменная ошибок
typedef ... BaseType; // Определить тип элемента стека
typedef struct Stack
{
    BaseType Buff[StackSize];
    unsigned Uk; /* Указывает на элемент, являющийся
                  вершиной стека */
};
void InitStack(Stack *s); // Инициализация стека
int EmptyStack(Stack *s); // Проверка: стек пуст?
void PutStack(Stack *s, BaseType E); /* Поместить элемент в
                                     стек */
void GetStack(Stack *s, BaseType *E); /* Извлечь элемент из
                                     стека */
void ReadStack(Stack *s, BaseType *E); /* Прочитать элемент
                                     из вершины стека */
#endif

```

2. Стек на массиве в динамической памяти.

Реализация на языке *Pascal*:

```
Unit Stack2;
```

*Interface**Const**StackOk* = 0;*StackOver* = 1;*StackUnder* = 2;*var StackError*:0..2;*Type**Index* = 0..*StackSize*;*BaseType* = . . .; {определить тип элемента стека}*Const**StackSize* = 65520 div sizeof(*BaseType*);*Type**Index* = 0..*StackSize*;*TBuf* = array[*Index*]of *BaseType*;*PBuf* = ^*TBuf*;*Stack* = record*Buf*: *PBuf*;*Kbuf*: word; {количество элементов в массиве, заполняется при инициализации}*Uk* : *Index*; {указывает на элемент, следующий за вершиной стека}*end*;*procedure InitStack*(*var s*:*Stack*; *var SizeBuf*:word);

{инициализация стека}

function EmptyStack(*var s*:*Stack*):boolean; {стек пуст}*procedure PutStack*(*var s*:*Stack*; *El*:*BaseType*); {поместить элемент в стек}*procedure GetStack*(*var s*:*Stack*; *var El*:*BaseType*);

{извлечь элемент из стека}

procedure ReadStack(*const s*:*Stack*; *var El*: *BaseType*);

{прочитать элемент из вершины стека}

procedure DoneStack(*Var S*:*Stack*); {уничтожить стек}**Реализация на языке C:**

#if !defined(__STACK2_H)

#define __STACK2_H

const *StackOk* = 0;const *StackOver* = 1;const *StackUnder* = 2;int *StackError*; // Переменная ошибокtypedef ... *BaseType*; // Определить тип элемента стека


```

typedef struct Stack
{
    BaseType *Buf; // Массив элементов базового типа
    unsigned Kbuf; /* Количество элементов в массиве,
                    заполняется при инициализации */
    unsigned Uk; /* Указывает на элемент, следующий за
                  вершиной стека */
};

void InitStack(Stack *s, unsigned SizeBuf); /* Инициализация
                                             стека */

int EmptyStack(Stack *s); // Проверка: стек пуст?

void PutStack(Stack *s, BaseType E); /* Поместить элемент в
                                       стек */

void GetStack(Stack *s, BaseType *E); /* Извлечь элемент из
                                       стека */

void ReadStack(Stack *s, BaseType *E); /* Прочитать элемент
                                       из вершины стека */

void DoneStack(Stack *s); // Уничтожить стек
#endif

```

3. Стек на массиве в статической памяти, элементы стека — в динамической.

Реализация на языке *Pascal*:

```

Unit Stack3;
  Interface
Const
  StackSize = 1000;
  StackOk   = 0;
  StackOver = 1;
  StackUnder = 2;
var
  StackError: 0..2;
Type
  Index   = 0..StackSize;
  BaseType = Pointer;
  Stack = record
    Buf: array[Index] of BaseType;
    SizeEl: word; {размер элемента стека, определяется
                  при инициализации}
    Uk : Index; {указывает на элемент, являющийся

```

```

        вершиной стека}
end;

procedure InitStack(var s:Stack; size: word);
    {инициализация стека}
function EmptyStack(var s:Stack):boolean; {стек пуст}
procedure PutStack(var s:Stack; var El); {поместить
        элемент в стек}
procedure GetStack(var s:Stack;var El); {извлечь элемент
        из стека}
procedure ReadStack(const s:Stack;var El); {прочитать
        элемент из вершины стека}

```

Реализация на языке C:

```

#ifdef __STACK3_H
#define __STACK3_H
const StackSize = 1000;
const StackOk = 0;
const StackOver = 1;
const StackUnder = 2;
int StackError; // Переменная ошибок
typedef void *BaseType;
typedef struct Stack
{
    BaseType Buff[StackSize];
    unsigned SizeEl; /* Размер элемента стека,
        определяющийся при инициализации */
    unsigned Uk; /* Указывает на элемент, являющийся
        вершиной стека */
};
void InitStack(Stack *s, unsigned Size); /* Инициализация
        стека */
int EmptyStack(Stack *s); // Проверка: стек пуст?
void PutStack(Stack *s, void *E); /* Поместить элемент в
        стек */
void GetStack(Stack *s, void *E); /* Извлечь элемент из
        стека */
void ReadStack(Stack *s, void *E); /* Прочитать элемент из
        вершины стека */
#endif

```

4. Стек в динамической памяти (связанная схема).

Реализация на языке *Pascal*:

```

Unit Stack4;
  Interface
Const
  StackOk   = 0;
  StackOver = 1;
  StackUnder= 2;
var
  StackError:0..2;
Type
  BaseType = ...; {определить тип элемента стека}
  PtrEl= ^Element;
  Element      = Record
    Data : BaseType;
    Next : PtrEl;
  End;
  Stack = PtrEl; {указывает на первый элемент в ОЛС,
                  являющийся вершиной стека}
  procedure InitStack(var s:Stack); {инициализация стека}
  function  EmptyStack(var s:Stack):boolean; {стек пуст}
  procedure PutStack(var s:Stack;El:BaseType); {поместить
                  элемент в стек}
  procedure GetStack(var s:Stack;var El:BaseType);
                  {извлечь элемент из стека}
  procedure ReadStack(const s:Stack;var El: BaseType);
                  {прочитать элемент из вершины стека}
  procedure DoneStack(Var S:Stack); {уничтожить стек}

```

Реализация на языке *C*:

```

#if !defined(__STACK4_H)
#define __STACK4_H
const StackOk   = 0;
const StackOver = 1;
const StackUnder = 2;
int  StackError; // Переменная ошибок
typedef ... BaseType; // Определить тип элемента стека
typedef struct Element
{
  BaseType Data;

```

```

unsigned SizeEl; /* Размер элемента стека, определяющийся
                  при инициализации */
Element *Next;
};
typedef Element *Stack; /* Указывает на первый элемент в ОЛС,
                          являющийся вершиной стека */
void InitStack(Stack *s); /* Инициализация стека
int  EmptyStack(Stack s); /* Проверка: стек пуст?
void PutStack(Stack *s, BaseType E); /* Поместить элемент в
                                     стек */
void GetStack(Stack *s, BaseType *E); /* Извлечь элемент из
                                     стека */
void ReadStack(Stack s, BaseType *E); /* Прочитать элемент
                                     из вершины стека */
void DoneStack(Stack *s); /* Уничтожить стек
#endif

```

5. Стек на ОЛС. Вершина стека – первый элемент ОЛС.

Реализация на языке *Pascal*:

```

unit stack5;
interface
uses list1; {см лаб.раб. №5}
const StackOk=ListOk;
      StackUnder=ListUnder;
      StackOver=ListNotMem;
type stack=list;
procedure InitStack(var s : stack); {инициализация стека}
procedure PutStack(var s : stack; b : basetype);
      {поместить элемент в стек}
procedure GetStack(var s : stack; var b : basetype);
      {извлечь элемент из стека}
function EmptyStack(s : stack):boolean; {стек пуст}
procedure ReadStack(s:Stack var b : basetype); {прочитать
      элемент из вершины стека}
procedure DoneStack(var s:Stack);{разрушить стек}
var stackerror:byte;

```

Реализация на языке *C*:

```

#if !defined(__STACK5_H)
#define __STACK5_H

```

```

#include "list1.h" // Смотреть лаб. раб. №5
const StackOk = ListOk;
const StackUnder = ListUnder;
const StackOver = ListNotMem;
int StackError; // Переменная ошибок
typedef List Stack;
void InitStack(Stack *s); // Инициализация стека
void PutStack(Stack *s, BaseType E); /* Поместить элемент в
                                     стек */
void GetStack(Stack *s; BaseType *E); /* Извлечь элемент из
                                     стека */
int EmptyStack(Stack s); // Проверка: стек пуст?
void ReadStack(Stack s, BaseType *E); /* Прочитать элемент из
                                     вершины стека */
void DoneStack(Stack *s); // Уничтожить стек
#endif

```

6. Стек на ОЛС. Вершина стека – последний элемент ОЛС.

Реализация на языке *Pascal*:

```

unit stack6;
interface
uses list3; { см лаб. раб. №5 }
const StackOk=ListOk;
      StackUnder=ListUnder;
      StackOver=ListNotMem;
type stack=list;
procedure InitStack(var s : stack); {инициализация стека}
procedure PutStack(var s : stack; var b);
      { поместить элемент в стек }
procedure GetStack(var s : stack; var b);
      { извлечь элемент из стека }
function EmptyStack(s : stack):boolean; {стек пуст}
procedure ReadStack(s:Stack; var b); {прочитать
      элемент из вершины стека}
procedure DoneStack(var s:Stack); {разрушить стек}
var stackerror:byte;

```

Реализация на языке *C*:

```

#if !defined(__STACK6_H)
#define __STACK6_H

```

```

#include "list3.h" // Смотреть лаб. раб. №5
const StackOk = ListOk;
const StackUnder = ListUnder;
const StackOver = ListNotMem;
int StackError; // Переменная ошибок
typedef List Stack;
void InitStack(Stack *s); // Инициализация стека
void PutStack(Stack *s, void *E); /* Поместить элемент в
                                стек */
void GetStack(Stack *s, void *E); /* Извлечь элемент из
                                стека */
int EmptyStack(Stack s); // Проверка: стек пуст?
void ReadStack(Stack s, void *E); /* Прочитать элемент из
                                вершины стека */
void DoneStack(Stack *s); // Разрушить стек
#endif

```

7. Стек на ОЛС. Вершина стека – первый элемент ОЛС.

Реализация на языке *Pascal*:

```

unit stack7;
interface
uses list3; { см лаб. раб. №5 }
const StackOk=ListOk;
      StackUnder=ListUnder;
      StackOver=ListNotMem;
type stack=list;
procedure InitStack(var s : stack; size : word);
      { инициализация стека }
procedure PutStack(var s : stack; var b);
      { поместить элемент в стек }
procedure GetStack(var s : stack; var b);
      { извлечь элемент из стека }
function EmptyStack(s : stack):boolean; { стек пуст }
procedure ReadStack(s:Stack; var b); { прочитать
      элемент из вершины стека }
procedure DoneStack(var s:Stack); { разрушить стек }
var stackerror:byte;

```

Реализация на языке *C*:

```

#ifndef __STACK7_H

```

```

#define __STACK7_H
#include "list3.h" // Смотреть лаб. раб. №5
const StackOk = ListOk;
const StackUnder = ListUnder;
const StackOver = ListNotMem;
int StackError; // Переменная ошибок
typedef List Stack;
void InitStack(Stack *s, unsigned Size); /* Инициализация
                                         стека */
void PutStack(Stack *s, void *E); // Поместить элемент в стек
void GetStack(Stack *s; void *E); // Извлечь элемент из стека
int EmptyStack(Stack s); // Проверка: стек пуст?
void ReadStack(Stack s, void *E); /* Прочитать элемент из
                                   вершины стека */
void DoneStack(Stack *s); // Уничтожить стек
#endif

```

8. Стек на ОЛС. Вершина стека – первый элемент ОЛС.

Реализация на языке *Pascal*:

```

unit stack8;
interface
uses list4; { см лаб. раб. №5}
const StackOk=ListOk;
      StackUnder=ListUnder;
      StackOver=ListNotMem;
type stack=list;
procedure InitStack(var s : stack); {инициализация стека}
procedure PutStack(var s : stack; b : basetype);
      {поместить элемент в стек}
procedure GetStack(var s : stack; var b : basetype);
      {извлечь элемент из стека}
function EmptyStack(s : stack):boolean; {стек пуст}
procedure ReadStack(s:Stack;var b : basetype); {прочитать
      элемент из вершины стека}
procedure DoneStack(var s:Stack); {разрушить стек}
var stackerror:byte;

```

Реализация на языке *C*:

```

#if !defined(__STACK8_H)
#define __STACK8_H

```

```

#include "list4.h" // Смотреть лаб. раб. №5
const StackOk = ListOk;
const StackUnder = ListUnder;
const StackOver = ListNotMem;
int StackError; // Переменная ошибок
typedef List Stack;
void InitStack(Stack *s); /* Инициализация стека */
void PutStack(Stack *s, BaseType E); /* Поместить элемент в
                                     стек */
void GetStack(Stack *s, BaseType *E); /* Извлечь элемент из
                                     стека */
int EmptyStack(Stack s); // Проверка: стек пуст?
void ReadStack(Stack s, BaseType *E); /* Прочитать элемент из
                                     вершины стека */
void DoneStack(Stack *s); // Уничтожить стек
#endif

```

9. Стек на ОЛС. Вершина стека – последний элемент ОЛС.

Реализация на языке *Pascal*:

```

unit stack9;
interface
uses list5; { см лаб. раб. №5 }
const StackOk=ListOk;
      StackUnder=ListUnder;
      StackOver=ListNotMem;
type stack=list;
procedure InitStack(var s : stack); {инициализация стека}
procedure PutStack(var s : stack; b : basetype);
      {поместить элемент в стек}
procedure GetStack(var s : stack; var b : basetype);
      {извлечь элемент из стека}
function EmptyStack(s : stack):boolean; {стек пуст}
procedure ReadStack(s:Stack;var b : basetype); {прочитать
      элемент из вершины стека}
procedure DoneStack(var s:Stack); {разрушить стек}
var stackerror:byte;

```

Реализация на языке *C*:

```

#ifndef __STACK9_H
#define __STACK9_H

```



```

#include "list5.h" // Смотреть лаб. раб. №5
const StackOk = ListOk;
const StackUnder = ListUnder;
const StackOver = ListNotMem;
int StackError; // Переменная ошибок
typedef List Stack;
void InitStack(Stack *s); /* Инициализация стека */
void PutStack(Stack *s, BaseType E); /* Поместить элемент в
                                     стек */
void GetStack(Stack *s, BaseType *E); /* Извлечь элемент из
                                     стека */
int EmptyStack(Stack s); // Проверка: стек пуст?
void ReadStack(Stack s, BaseType *E); /* Прочитать элемент из
                                     вершины стека */
void DoneStack(Stack *s); // Уничтожить стек
#endif

```

10. Стек на ПЛС. Вершина стека – первый элемент ПЛС.

Реализация на языке *Pascal*:

```

unit stack10;
interface
uses list6; { см лаб. раб. №5 }
const StackOk=ListOk;
      StackUnder=ListUnder;
      StackOver=ListNotMem;
type stack=list;
procedure InitStack(var s : stack; size : word);
      { инициализация стека }
procedure PutStack(var s : stack; var b);
      { поместить элемент в стек }
procedure GetStack(var s : stack; var b);
      { извлечь элемент из стека }
function EmptyStack(s : stack):boolean; { стек пуст }
procedure ReadStack(s:Stack; var b); { прочитать
      элемент из вершины стека }
procedure DoneStack(var s:Stack); { разрушить стек }
var stackerror:byte;

```

Реализация на языке *C*:

```

#if !defined(__STACK10_H)

```

```

#define __STACK10_H
#include "list6.h" // Смотреть лаб. раб. №5
const StackOk = ListOk;
const StackUnder = ListUnder;
const StackOver = ListNotMem;
int StackError; // Переменная ошибок
typedef List Stack;
void InitStack(Stack *s, unsigned Size); /* Инициализация стека */
void PutStack(Stack *s, void *E); // Поместить элемент в стек
void GetStack(Stack *s, void *E); // Извлечь элемент из стека
int EmptyStack(Stack s); // Проверка: стек пуст?
void ReadStack(Stack s, void *E); /* Прочитать элемент из
                                вершины стека */
void DoneStack(Stack *s); // Уничтожить стек
#endif

```

11. Стек на ПЛС. Вершина стека – последний элемент ПЛС.

Реализация на языке *Pascal*:

```

unit stack11;
interface
uses list8; {см лаб. раб. №5}
const StackOk=ListOk;
      StackUnder=ListUnder;
      StackOver=ListNotMem;
type stack=list;
procedure InitStack(var s : stack; SizeMem, SizeEl:Word);
      {инициализация стека}
procedure PutStack(var s : stack; var b);
      {поместить элемент в стек}
procedure GetStack(var s : stack; var b);
      {извлечь элемент из стека}
function EmptyStack(s : stack):boolean; {стек пуст}
procedure ReadStack(s:Stack; var b); {прочитать
      элемент из вершины стека}
procedure DoneStack(var s:Stack);{разрушить стек}
var stackerror:byte;

```

Реализация на языке *C*:

```

#ifndef __STACK11_H
#define __STACK11_H

```

```

#include "list8.h" // Смотреть лаб. раб. №5
const StackOk = ListOk;
const StackUnder = ListUnder;
const StackOver = ListNotMem;
int StackError; // Переменная ошибок
typedef List Stack;
void InitStack(Stack *s, unsigned SizeMem, unsigned SizeEl);
/* Инициализация стека */
void PutStack(Stack *s, void *E); // Поместить элемент в стек
void GetStack(Stack *s, void *E); // Извлечь элемент из стека
int EmptyStack(Stack s); // Проверка: стек пуст?
void ReadStack(Stack s, void *E); /* Прочитать элемент из
                                   вершины стека */
void DoneStack(Stack *s); // Уничтожить стек
#endif

```

Модули для реализации очереди

1. Очередь на ПЛС. «Хвост» очереди — последний, а «голова» — первый элемент ПЛС.

Реализация на языке *Pascal*:

```

unit Fifo1;
interface
uses list8; { см лаб. раб. №5 }
const FifoOk=ListOk;
      FifoUnder=ListUnder;
      FifoOver=ListNotMem;
type Fifo=list;
procedure InitFifo(var f: fifo; SizeMem, SizeEl: Word);
      {инициализация очереди}
procedure PutFifo(var f: fifo; b: basetype);
      {поместить элемент в очередь}
procedure GetFifo(var f: fifo; var b: basetype);
      {извлечь элемент из очереди}
function EmptyFifo(f: Fifo):boolean; {очередь пуста}
procedure DoneFifo(var f: fifo); {разрушить очередь}
var fifoerror:byte;

```

Реализация на языке *C*:

```

#ifndef __FIFO1_H
#define __FIFO1_H

```

```

#include "list8.h" // Смотреть лаб. раб. №5
const FifoOk = ListOk;
const FifoUnder = ListUnder;
const FifoOver = ListNotMem;
int FifoError; // Переменная ошибок
typedef List Fifo;
void InitFifo(Fifo *f, unsigned SizeMem, unsigned SizeEl);
// Инициализация очереди
void PutFifo(Fifo *f, BaseType E); /* Поместить элемент в очередь */
void GetFifo(Fifo *f, BaseType *E); /* Извлечь элемент из очереди */
void ReadFifo(Fifo *f, BaseType *E); // Прочитать элемент
int EmptyFifo(Fifo *f); // Проверка, пуста ли очередь?
void DoneFifo(Fifo *f); // Разрушить очередь
#endif

```

2. Очередь на ПЛС. «Хвост» очереди — первый, а «голова» — последний элемент ПЛС.

Реализация на языке *Pascal*:

```

unit Fifo2;
interface
uses list7; { см лаб. раб. №5 }
const FifoOk=ListOk;
      FifoUnder=ListUnder;
      FifoOver=ListNotMem;
type Fifo=list;
procedure InitFifo(var f : fifo; Size:Word);
      {инициализация очереди}
procedure PutFifo(var f : fifo; b : basetype);
      {поместить элемент в очередь}
procedure GetFifo(var f : fifo; var b : basetype);
      {извлечь элемент из очереди}
function EmptyFifo(f : Fifo):boolean; {очередь пуста}
procedure DoneFifo(var f: fifo); {разрушить очередь}
var fifoerror:byte;

```

Реализация на языке *C*:

```

#ifndef __FIFO2_H
#define __FIFO2_H
#include "list7.h" // Смотреть лаб. раб. №5
const FifoOk = ListOk;
const FifoUnder = ListUnder;

```

```

const FifoOver = ListNotMem;
int FifoError; // Переменная ошибок
typedef List Fifo;
void InitFifo(Fifo *f, unsigned Size); /* Инициализация очереди */
void PutFifo(Fifo *f, BaseType E); /* Поместить элемент в очередь */
void GetFifo(Fifo *f, BaseType *E); /* Извлечь элемент из очереди */
void ReadFifo(Fifo *f, BaseType *E); // Прочитать элемент
int EmptyFifo(Fifo *f); // Проверка, пуста ли очередь?
void DoneFifo(Fifo *f); // Разрушить очередь
#endif

```

3. Очередь на ОЛС. «Хвост» очереди — последний, а «голова» — первый элемент ОЛС.

Реализация на языке Pascal:

```

unit Fifo3;
interface
uses list5; { см лаб. раб. №5 }
const FifoOk=ListOk;
      FifoUnder=ListUnder;
      FifoOver=ListNotMem;
type Fifo=list;
procedure InitFifo(var f : fifo); { инициализация очереди }
procedure PutFifo(var f : fifo; b : basetype);
      { поместить элемент в очередь }
procedure GetFifo(var f : fifo; var b : basetype);
      { извлечь элемент из очереди }
function EmptyFifo(f : Fifo):boolean; { очередь пуста }
procedure DoneFifo(var s: fifo); { разрушить очередь }
var fifoerror:byte;

```

Реализация на языке C:

```

#ifndef __FIFO3_H
#define __FIFO3_H
#include "list5.h" // Смотреть лаб. раб. №5
const FifoOk = ListOk;
const FifoUnder = ListUnder;
const FifoOver = ListNotMem;
int FifoError; // Переменная ошибок
typedef List Fifo;
void InitFifo(Fifo *f); // Инициализация очереди
void PutFifo(Fifo *f, BaseType E); /* Поместить элемент в очередь */
void GetFifo(Fifo *f, BaseType *E); /* Извлечь элемент из очереди */

```

```

void ReadFifo(Fifo *f, BaseType *E); // Прочитать элемент
int EmptyFifo(Fifo *f); // Проверка, пуста ли очередь?
void DoneFifo(Fifo *f); // Разрушить очередь
#endif

```

4. Очередь на ОЛС. «Хвост» очереди — первый, а «голова» — последний элемент ОЛС.

Реализация на языке *Pascal*:

```

unit Fifo4;
interface
uses list5; { см лаб. раб. №5 }
const FifoOk=ListOk;
      FifoUnder=ListUnder;
      FifoOver=ListNotMem;
type Fifo=list;
procedure InitFifo(var f : fifo); { инициализация очереди }
procedure PutFifo(var f : fifo; b : basetype);
      { поместить элемент в очередь }
procedure GetFifo(var f : fifo; var b : basetype);
      { извлечь элемент из очереди }
function EmptyFifo(f : Fifo):boolean; { очередь пуста }
procedure DoneFifo(var f : Fifo); { разрушить очередь }
var fifoerror:byte;

```

Реализация на языке *C*:

```

#ifndef __FIFO4_H
#define __FIFO4_H
#include "list5.h" // Смотреть лаб. раб. №5
const FifoOk = ListOk;
const FifoUnder = ListUnder;
const FifoOver = ListNotMem;
int FifoError; // Переменная ошибок
typedef List Fifo;
void InitFifo(Fifo *f); // Инициализация очереди
void PutFifo(Fifo *f, BaseType E); /* Поместить элемент в очередь */
void GetFifo(Fifo *f, BaseType *E); /* Извлечь элемент из очереди */
void ReadFifo(Fifo *f, BaseType *E); // Прочитать элемент
int EmptyFifo(Fifo *f); // Проверка, пуста ли очередь?
void DoneFifo(Fifo *f); // Разрушить очередь
#endif

```

5. Очередь на ОЛС. «Хвост» очереди — последний, а «голова» — первый элемент ОЛС.

Реализация на языке *Pascal*:

```
unit Fifo5;
interface
uses list3; { см лаб. раб. №5 }
const FifoOk=ListOk;
      FifoUnder=ListUnder;
      FifoOver=ListNotMem;
type Fifo=list;
procedure InitFifo(var f : fifo; Size:Word);
      {инициализация очереди}
procedure PutFifo(var f : fifo; var b);
      {поместить элемент в очередь}
procedure GetFifo(var f : fifo; var b);
      {извлечь элемент из очереди}
function EmptyFifo(f : fifo):boolean; {очередь пуста}
procedure DoneFifo(var f: fifo); {разрушить очередь}
var fifoerror:byte;
```

Реализация на языке *C*:

```
#if !defined(__FIFO5_H)
#define __FIFO5_H
#include "list3.h" // Смотреть лаб. раб. №5
const FifoOk = ListOk;
const FifoUnder = ListUnder;
const FifoOver = ListNotMem;
int FifoError; // Переменная ошибок
typedef List Fifo;
void InitFifo(Fifo *f, unsigned Size); /* Инициализация очереди */
void PutFifo(Fifo *f, void *E); // Поместить элемент в очередь
void GetFifo(Fifo *f, void *E); // Извлечь элемент из очереди
void ReadFifo(Fifo *f, void *E); // Прочитать элемент
int EmptyFifo(Fifo *f); // Проверка, пуста ли очередь?
void DoneFifo(Fifo *f); // Разрушить очередь
#endif
```

6. Очередь на ОЛС. «Хвост» очереди — первый, а «голова» — последний элемент ОЛС.

Реализация на языке *Pascal*:

```
unit Fifo6;
```

```

interface
uses list3; { см лаб.раб. №5}
const FifoOk=ListOk;
      FifoUnder=ListUnder;
      FifoOver=ListNotMem;
type Fifo=list;
procedure InitFifo(var f : fifo; Size:Word);
      {инициализация очереди}
procedure PutFifo(var f : fifo; var b);
      {поместить элемент в очередь}
procedure GetFifo(var f : fifo; var b);
      {извлечь элемент из очереди}
function EmptyFifo(f : fifo):boolean; {очередь пуста}
procedure DoneFifo(var f: fifo); {разрушить очередь}
var fifoerror:byte;

```

Реализация на языке C:

```

#ifndef __FIFO6_H
#define __FIFO6_H
#include "list3.h" // Смотреть лаб. раб. №5
const FifoOk = ListOk;
const FifoUnder = ListUnder;
const FifoOver = ListNotMem;
int FifoError; // Переменная ошибок
typedef List Fifo;
void InitFifo(Fifo *f, unsigned Size); //Инициализация очереди
void PutFifo(Fifo *f, void *E); // Поместить элемент в очередь
void GetFifo(Fifo *f, void *E); // Извлечь элемент из очереди
void ReadFifo(Fifo *f, void *E); // Прочитать элемент
int EmptyFifo(Fifo *f); // Проверка, пуста ли очередь?
void DoneFifo(Fifo *f); // Разрушить очередь
#endif

```

7. Очередь на ОЛС. «Хвост» очереди — последний, а «голова» — первый элемент ОЛС.

Реализация на языке Pascal:

```

unit Fifo7;
interface
uses list2; { см лаб.раб. №5}
const FifoOk=ListOk;
      FifoUnder=ListUnder;

```



```

    FifokOver=ListNotMem;
type Fifo=list;
procedure InitFifo(var f: fifo); {инициализация очереди}
procedure PutFifo(var f: fifo; b: basetype);
    {поместить элемент в очередь}
procedure GetFifo(var f: fifo; var b: basetype);
    {извлечь элемент из очереди}
function EmptyFifo(f: fifo):boolean; {очередь пуста}
procedure DoneFifo(var f: fifo); {разрушить очередь}
var fifoerror:byte;

```

Реализация на языке C:

```

#if !defined(__FIFO7_H)
#define __FIFO7_H
#include "list2.h" // Смотреть лаб. раб. №5
const FifoOk = ListOk;
const FifoUnder = ListUnder;
const FifoOver = ListNotMem;
int FifoError; // Переменная ошибок
typedef List Fifo;
void InitFifo(Fifo *f); // Инициализация очереди
void PutFifo(Fifo *f, BaseType E); /* Поместить элемент в очередь */
void GetFifo(Fifo *f, BaseType *E); /* Извлечь элемент из очереди */
void ReadFifo(Fifo *f, BaseType *E); // Прочитать элемент
int EmptyFifo(Fifo *f); // Проверка, пуста ли очередь?
void DoneFifo(Fifo *f); // Разрушить очередь
#endif

```

8. Очередь (кольцевая) на массиве в статической памяти.

Реализация на языке Pascal:

```

Unit Fifo8;
Interface
Const
    FifoSize = 1000;
    FifoOk = 0;
    FifoOver = 1;
    FifoUnder = 2;
var
    FifoError:0..2;
Type

```

```

Index = 0..FifoSize;
BaseType = ...; {определить тип элемента очереди}
Fifo = record
  Buf: array[Index]of BaseType;
  Uk1 : Index; {указывает на "голову" очереди}
  Uk2 : Index; {указывает на "хвост" очереди}
end;
procedure InitFifo(var f : fifo); {инициализация очереди}
procedure PutFifo(var f : fifo; var b); {поместить элемент в очередь}
procedure GetFifo(var f : fifo; var b); {извлечь элемент из очереди}
function EmptyFifo(f : fifo):boolean; {очередь пуста}

```

Реализация на языке C:

```

#ifndef __FIFO8_H
#define __FIFO8_H
const FifoOk = 0;
const FifoUnder = 1;
const FifoOver = 2;
const FifoSize = 1000;
int FifoError; // Переменная ошибок
typedef... BaseType; // Определить тип элемента очереди
typedef struct Fifo
{
  BaseType Buf[FifoSize];
  unsigned Uk1; // Указатель на "голову" очереди
  unsigned Uk2; // Указатель на "хвост" очереди
  unsigned N; // Количество элементов очереди
};
void InitFifo(Fifo*f); // Инициализация очереди
void PutFifo(Fifo *f, BaseType E); /* Поместить элемент в очередь */
void GetFifo(Fifo *f, BaseType *E); /* Извлечь элемент из очереди */
void ReadFifo(Fifo *f, BaseType *E); // Прочитать элемент
int EmptyFifo(Fifo *f); // Проверка, пуста ли очередь?
#endif

```

9. Очередь (кольцевая) на массиве в динамической памяти.

Реализация на языке Pascal:

```

Unit Fifo9;
Interface
Const

```

```

FifoOk = 0;
FifoOver = 1;
FifoUnder = 2;
var FifoError: 0..2;
Type
  BaseType = ...; {определить тип элемента очереди}
Const
  FifoSize = 65520 div sizeof(BaseType);
Type
  Index = 0..FifoSize;
  TBuf = array[Index] of BaseType
  Fifo = record
    PBuf: ^TBuf;
    SizeBuf: word; {количество элементов в массиве}
    Uk1: Index; {указывает на "голову" очереди}
    Uk2: Index; {указывает на "хвост" очереди}
  end;
procedure InitFifo(var f: fifo; size: word);
  {инициализация очереди}
procedure PutFifo(var f: fifo; b: basetype);
  {поместить элемент в очередь}
procedure GetFifo(var f: fifo; var b: basetype);
  {извлечь элемент из очереди}
function EmptyFifo(f: fifo):boolean; {очередь пуста}
procedure DoneFifo(var f: fifo); {разрушить очередь}

```

Реализация на языке C:

```

#ifdef __FIFO9_H
#define __FIFO9_H
const FifoOk = 0;
const FifoUnder = 1;
const FifoOver = 2;
const FifoSize = 1000;
int FifoError; // Переменная ошибок
typedef ... BaseType; // Определить тип элемента очереди
typedef struct Fifo
{
  BaseType *Buf;
  unsigned SizeBuf; /* Максимальная длина массива,
                     определяющаяся при инициализации */
  unsigned Uk1; // Указатель на "голову" очереди

```

```

    unsigned Uk2; // Указатель на "хвост" очереди
    unsigned N; // Количество элементов очереди
};
void InitFifo(Fifo* f, unsigned Size); /* Инициализация очереди */
void PutFifo(Fifo *f, BaseType E); /* Поместить элемент в очередь */
void GetFifo(Fifo *f, BaseType *E); /* Извлечь элемент из очереди */
void ReadFifo(Fifo *f, BaseType *E); // Прочитать элемент
int EmptyFifo(Fifo *f); // Проверка, пуста ли очередь?
void DoneFifo(Fifo *f); // Разрушить очередь
#endif

```

10. Очередь (кольцевая) на массиве в статической памяти с элементами в динамической памяти.

Реализация на языке *Pascal*:

```

Unit Fifo10;
  Interface
  Const
    FifoSize = 1000;
    FifoOk  = 0;
    FifoOver = 1;
    FifoUnder = 2;
  var
    FifoError: 0..2;
  Type
    Index  = 0..FifoSize;
    BaseType = pointer;
    Fifo = record
      Buf: array[Index] of BaseType;
      SizeEl: word; {размер элемента очереди}
      Uk1 : Index; {указывает на "голову" очереди}
      Uk2 : Index; {указывает на "хвост" очереди}
    end;
  procedure InitFifo(var f : fifo; size: word);
    {инициализация очереди}
  procedure PutFifo(var f : fifo; b : basetype);
    {поместить элемент в очередь}
  procedure GetFifo(var f : fifo; var b : basetype);
    {извлечь элемент из очереди}
  function EmptyFifo(f : fifo):boolean; {очередь пуста}

```

Реализация на языке C:

```

#ifndef __FIFO10_H
#define __FIFO10_H
const FifoOk = 0;
const FifoUnder = 1;
const FifoOver = 2;
const FifoSize = 1000;
int FifoError; // Переменная ошибок
typedef void *BaseType;
typedef struct Fifo
{
    BaseType Buf[FifoSize];
    unsigned SizeEl; // Размер элемента очереди
    unsigned Uk1; // Указатель на "голову" очереди
    unsigned Uk2; // Указатель на "хвост" очереди
    unsigned N; // Количество элементов очереди
};
void InitFifo(Fifo* f, unsigned Size); /* Инициализация очереди */
void PutFifo(Fifo *f, void *E); /* Поместить элемент в очередь */
void GetFifo(Fifo *f, void *E); // Извлечь элемент из очереди
void ReadFifo(Fifo *f, void *E); // Прочитать элемент
int EmptyFifo(Fifo *f); // Проверка, пуста ли очередь?
void DoneFifo(Fifo *f); // Разрушить очередь
#endif

```

11. Очередь (кольцевая) на массиве в динамической памяти.

Реализация на языке Pascal:

```

Unit Fifo11;
Interface
Const
    FifoSize = 65520 div sizeof(pointer);
    FifoOk = 0;
    FifoOver = 1;
    FifoUnder = 2;
var
    FifoError: 0..2;
Type
    Index = 0..FifoSize;
    BaseType = pointer;
    TBuf = array[Index] of BaseType

```

```

Stack = record
  PBuf: ^TBuf;
  SizeBuf: word; {количество элементов в массиве}
  SizeEl: word; {размер элемента очереди}
  Uk1 : Index; {указывает на “голову” очереди}
  Uk2 : Index; {указывает на “хвост” очереди}
end;
procedure InitFifo(var f : fifo; SizeBuf, SizeEL: word);
    {инициализация очереди}
procedure PutFifo(var f : fifo; b : basetype);
    {поместить элемент в очередь}
procedure GetFifo(var f : fifo; var b : basetype);
    {извлечь элемент из очереди}
function EmptyFifo(f : fifo):boolean; {очередь пуста}
procedure DoneFifo(var f: fifo); {разрушить очередь}

```

Реализация на языке C:

```

#ifndef __FIFO11_H
#define __FIFO11_H
const FifoOk = 0;
const FifoUnder = 1;
const FifoOver = 2;
int FifoError; // Переменная ошибок
typedef void *BaseType;
typedef struct Fifo
{
    BaseType *Buf;
    unsigned SizeBuf; // Максимальная длина очереди
    unsigned SizeEl; // Размер элемента очереди
    unsigned Uk1; // Указатель на «голову» очереди
    unsigned Uk2; // Указатель на «хвост» очереди
    unsigned N; // Количество элементов очереди
};
void InitFifo(Fifo* f, unsigned SizeEl, unsigned SizeBuf);
// Инициализация очереди
void PutFifo(Fifo *f, void *E); /* Поместить элемент в очередь */
void GetFifo(Fifo *f, void *E); // Извлечь элемент из очереди
void ReadFifo(Fifo *f, void *E); // Прочитать элемент
int EmptyFifo(Fifo *f); // Проверка, пуста ли очередь?
void DoneFifo(Fifo *f); // Разрушить очередь
#endif

```

Содержание отчета

1. Тема лабораторной работы.
2. Цель работы.
3. Характеристика СД типа «стек» и «очередь» (п.1 постановки задачи).
4. Индивидуальное задание.
5. Текст модулей для реализации СД типа «стек» и «очередь», текст программы для отладки модулей, тестовые данные и результат работы программы.
6. Текст программы моделирования системы.

Теоретические сведения

Стек

Стек — это последовательность элементов, в которой доступ (операции включения, исключения, чтения элемента) осуществляется только с одного конца структуры — с вершины стека. Стек называют структурой типа *LIFO* (от англ. *Last In, First Out*). Иногда стек называют магазином (по аналогии с магазином автомата). Стек — это динамическая структура.

Над стеком определены следующие основные операции:

1. Инициализация.
2. Включение элемента.
3. Исключение элемента.
4. Чтение элемента.
5. Проверка пустоты стека.
6. Уничтожение стека.

Кардинальное число стека определяется по формуле

$$CAR(\text{стек}) = CAR(BaseType)^0 + CAR(BaseType)^1 + \dots + CAR(BaseType)^{max},$$

где $CAR(BaseType)$ — кардинальное число элемента стека типа *BaseType*, *max* — максимальное количество элементов в стеке (не всегда определено, т.к. может зависеть от объема свободной динамической памяти).

На *абстрактном* уровне стек представляет собой линейную структуру — последовательность.

На *физическом* уровне стек может быть реализован последовательной или связной схемой хранения. Располагаться стек может в статической или динамической памяти. Стек, по сути, является линейным списком с ограниченным доступом к элементам, по этому он может быть реализован на основе СД ЛС. Достаточно ввести ограничения на операции и обеспечить доступ к элементу, являющемуся вершиной стека. В качестве такого эле-

мента может быть либо первый, либо последний элемент в ЛС. Рассмотрим особенности выполнения операций включения и исключения в зависимости от типа ЛС, на основе которого реализуется стек, и выбора вершины стека.

Пусть стек реализуется на основе ПЛС. Если в качестве вершины стека выбрать первый элемент ПЛС, то при выполнении операций включения или исключения все элементы стека перемещаются на одну позицию к концу или к началу соответственно, что требует значительных затрат времени при большом количестве элементов в стеке. Если в качестве вершины стека выбрать последний элемент ПЛС, то при выполнении операции включения необходимо включить новый элемент и сделать его вершиной стека (текущим), т.е. увеличить индекс текущего элемента. При выполнении операции исключения нужно текущим сделать предпоследний элемент, т.е. уменьшить индекс текущего элемента, и исключить следующий за ним. Эти операции довольно просты по времени выполнения. Операции над стеком можно сделать более эффективными, если реализовывать стек непосредственно на массиве. Тогда стеку ставится в соответствие дескриптор, состоящий из 3-х полей:

- 1 — массив (или указатель на массив), на основе которого реализуется стек;
- 2 — количество элементов в массиве;
- 3 — указатель вершины стека.

Указатель вершины стека может содержать индекс элемента, являющегося вершиной стека (рис.16) или индекс элемента, следующего за вершиной стека (рис.17).

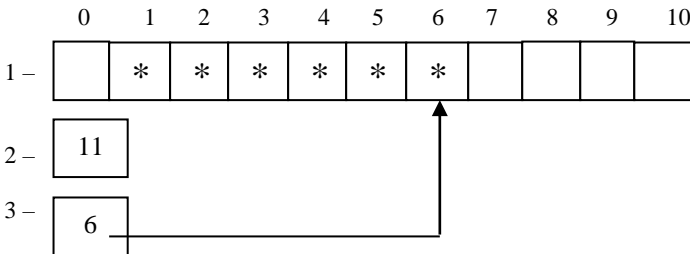


Рис.16. СД типа «стек»

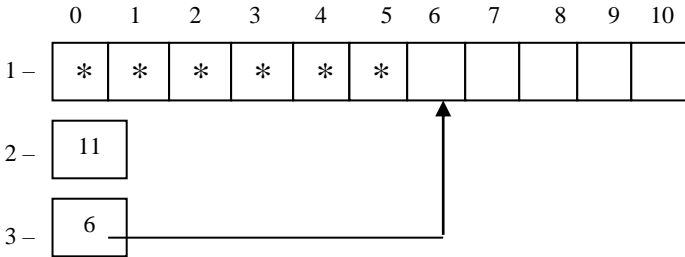


Рис.17. СД типа «стек»

На рис.16 элементы массива с индексами 1..6 являются элементами стека, а на рис.17 элементы стека — это элементы с индексами 0..5.

В указатель вершины стека при инициализации стека (см. рис.16) заносится 0, а при инициализации стека (см. рис.17) заносится 1. При включении элемента в стек (см. рис.16) сначала увеличивается указатель вершины стека, а затем в элемент массива с индексом, соответствующем указателю вершины стека, заносится значение включаемого в стек элемента. При включении элемента в стек (см. рис.17) порядок действий изменяется на противоположный.

При чтении элемента стека (см. рис.16) читается элемент массива, индекс которого соответствует указателю вершины стека, а при чтении элемента стека (рис.17) читается элемент массива, индекс которого на единицу меньше указателя вершины стека.

Для исключения элемента стека (см. рис.16, рис.17) достаточно уменьшить указатель вершины стека на единицу.

Теперь рассмотрим реализацию стека на СЛС.

Пусть стек реализуется на основе ОЛС. Если в качестве вершины стека выбрать первый элемент списка, то указатель текущего элемента ОЛС должен указывать на фиктивный элемент. При этом операции чтения, включения и исключения элемента выполняются достаточно быстро, не требуют изменения указателя на текущий элемент и перемещения элементов ОЛС. Если в качестве вершины стека принять последний элемент ОЛС, то при выполнении операций чтения и включения элемента в стек текущим элементом ОЛС должен быть последний элемент ОЛС. При выполнении операции исключения текущего элементом должен быть предпоследний в ОЛС, для поиска которого необходимо стать в начало ОЛС и последовательно пройти по всем элементам до предпоследнего, постоянно переопределяя указатель на текущий элемент, что требует неоправданных вычислений и затрат времени.

Учитывая ограниченный характер доступа к элементам стека можно достаточно эффективно реализовать по принципу ОЛС без фиктивного элемента. Тогда дескриптор стека будет содержать только одно поле (при реализации на связанных элементах, расположенных в динамической памяти), содержащее адрес первого элемента (вершины стека). Если элементами стека будут связанные элементы, расположенные в массиве (в статической или динамической памяти), то дескриптор будет содержать три поля:

- 1 — массив (или указатель на массив);
- 2 — количество элементов в массиве;
- 3 — указатель на вершину стека.

При реализации стека на основе ДЛС выбор элемента, являющегося вершиной стека не имеет значения, т.к. и для первого и для последнего элемента ДЛС операции чтения, включения и исключения эффективны в одинаковой степени.

Очередь

Очередь — это последовательность элементов, в которой включают элементы с одной стороны («хвост» очереди), а исключают с другой («голова» очереди). Очередь называют структурой типа *FIFO* (от англ. *First In, First Out*).

Очередь — это динамическая структура.

Над очередью определены следующие основные операции:

1. Инициализация.
2. Включение элемента.
3. Исключение элемента.
6. Проверка пустоты очереди.
7. Уничтожение очереди.

Кардинальное число очереди определяется по формуле

$$CAR(FIFO) = CAR(BaseType)^0 + CAR(BaseType)^1 + \dots + CAR(BaseType)^{max},$$

где $CAR(BaseType)$ — кардинальное число элемента очереди типа *BaseType*, *max* — максимальное количество элементов в очереди (не всегда определено, т.к. может зависеть от объема свободной динамической памяти).

На *абстрактном* уровне очередь представляет собой линейную структуру — последовательность.

На *физическом* уровне очередь может быть реализована последовательной или связанной схемой хранения. Располагаться очередь может в статической или динамической памяти. Очередь, по сути, является линейным списком с ограниченным доступом к элементам, по этому он может быть реализован на основе СД ЛС. Достаточно ввести ограничения на операции и обеспечить доступ к элементам, расположенным в начале и в конце очереди.

Рассмотрим особенности выполнения операций включения и исключения в зависимости от типа ЛС, на основе которого реализуется очередь, и выбора «головы» и «хвоста».

Пусть очередь реализуется на основе ПЛС. Если в качестве «головы» очереди выбрать первый элемент ПЛС, а в качестве «хвоста» — последний элемент ПЛС, то при выполнении операции включения необходимо стать в начало ПЛС и включить элемент, при этом все элементы ПЛС перемещаются к концу ПЛС. При выполнении операции исключения нужно стать на предпоследний элемент ПЛС и исключить последний, при этом элементы ПЛС не изменяют своего положения, поэтому операция исключения будет выполняться значительно эффективнее по сравнению с операцией включения.

Если в качестве «головы» очереди выбрать последний элемент ПЛС, а в качестве «хвоста» — первый элемент ПЛС, то при выполнении операции включения необходимо стать в конец ПЛС и включить элемент, при этом элементы ПЛС не изменяют своего положения. При выполнении операции исключения нужно стать в начало ПЛС и исключить элемент, при этом все элементы ПЛС перемещаются к началу ПЛС, поэтому операция исключения будет выполняться менее эффективно по сравнению с операцией включения.

Операции включения и исключения будут выполняться в одинаковой степени эффективно, если реализовать очередь непосредственно на массиве. В этом случае очереди ставится в соответствие дескриптор, состоящий из 5-ти полей:

- 1 — массив (или указатель на массив);
- 2 — количество элементов в массиве;
- 3 — указатель на «голову» очереди;
- 4 — указатель на «хвост» очереди;
- 5 — количество элементов в очереди.

Пусть указатель на «голову» указывает на тот элемент массива, который содержит элемент очереди, являющийся «головой», а указатель на «хвост» указывает на элемент массива, следующий за тем, который содержит «хвост» очереди.

При включении элемента в очередь его значение записывается в элемент массива, на который указывает указатель на «хвост», а сам указатель увеличивается на единицу.

При исключении элемента читается элемент массива, на который указывает указатель на «голову», а сам указатель увеличивается на единицу. Таким образом при последовательном выполнении операций включения и исключения очередь как бы перемещается по массиву от начала к концу, «захватывая» элементы массива с большими индексами и «освобождая»

элементы с меньшими индексами. При достижении «хвоста» очереди верхней границы массива выполнение операции включения становится невозможным, несмотря на то что первые элементы массива не используются. Можно устранить этот недостаток, организовав кольцевую очередь.

В кольцевой очереди, когда указатель «хвоста» достигает последнего элемента массива, элемент включается в очередь, а указатель устанавливается на первый элемент массива. Аналогично, когда указатель «головы» достигает последнего элемента массива, элемент исключается из очереди, а указатель устанавливается на первый элемент массива. На рис.18 и рис.19 показаны возможные состояния кольцевой очереди.

Пусть очередь реализуется на основе ОЛС. Тогда в качестве «головы» очереди нужно выбрать тот элемент ОЛС, над которым эффективно выполняется операция исключения, а в качестве «хвоста» — тот, над которым эффективно выполняется операция включения. Поэтому в качестве «головы» очереди выбираем первый элемент ОЛС, а в качестве «хвоста» — последний. Однако при выполнении операции включения необходимо стать в конец ОЛС, а это трудоемкая операция для ОЛС. Операции над очередью можно выполнить более эффективно, если хранить адрес последнего элемента (для быстрого включения). Тогда дескриптор очереди будет содержать три поля:

- 1 — указатель на «голову»;
- 2 — указатель на «хвост»;
- 3 — количество элементов в очереди.

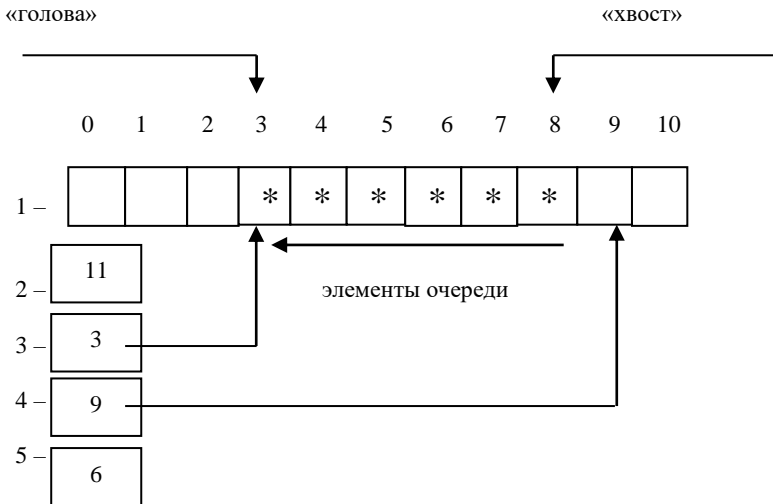


Рис.18. СД типа «кольцевая очередь»

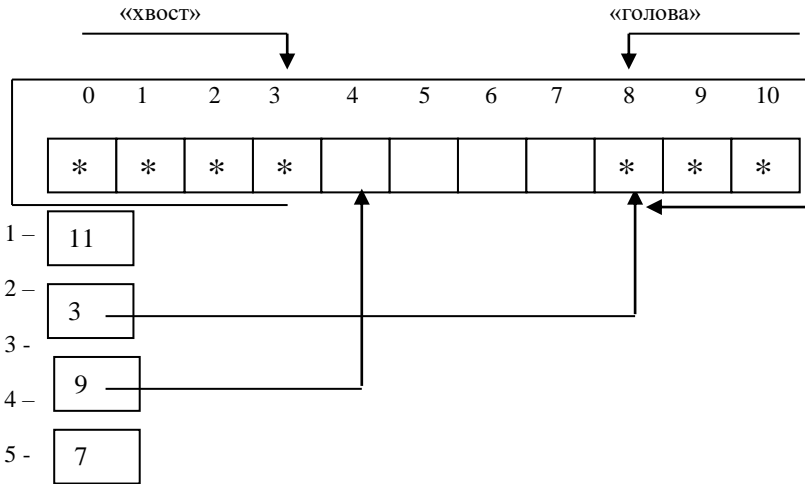


Рис.19. СД типа «кольцевая очередь»

При реализации очереди на основе ДЛС выбор элементов, являющихся «головой» и «хвостом» очереди не имеет значения, т.к. и для первого и для последнего элемента ДЛС операции чтения, включения и исключения эффективны в одинаковой степени.

Контрольные вопросы

1. Определите порядок функции временной сложности операции включения элемента в стек, если вершиной стека является первый (последний) элемент последовательного списка.
2. Определите порядок функции временной сложности операции включения элемента в стек, если вершиной стека является первый (последний) элемент односвязного списка.
3. Определите порядок функции временной сложности операций включения и исключения элемента очереди при ее реализации на последовательном списке.
4. Определите порядок функции временной сложности операций включения и исключения элемента кольцевой очереди.
5. Определите порядок функции временной сложности операций включения и исключения элемента очереди при ее реализации на односвязном списке.

Лабораторная работа № 7

Структуры данных типа «дерево» (Pascal/C)

Цель работы: изучить СД типа «дерево», научиться их программно реализовывать и использовать.

З а д а н и е

1. Для СД типа «дерево» определить:
 - 1.1. Абстрактный уровень представления СД:
 - 1.1.1. Характер организованности и изменчивости.
 - 1.1.2. Набор допустимых операций.
 - 1.2. Физический уровень представления СД:
 - 1.2.1. Схему хранения.
 - 1.2.2. Объем памяти, занимаемый экземпляром СД.
 - 1.2.3. Формат внутреннего представления СД и способ его интерпретации.
 - 1.2.4. Характеристику допустимых значений.
 - 1.2.5. Тип доступа к элементам.
 - 1.3. Логический уровень представления СД.
 Способ описания СД и экземпляра СД на языке программирования.
2. Реализовать СД типа «дерево» в соответствии с вариантом индивидуального (табл.17) задания в виде модуля.
3. Разработать программу для решения задачи в соответствии с вариантом индивидуального задания (см. табл.17) с использованием модуля, полученного в результате выполнения пункта 2 задания.

Таблица 17

Варианты индивидуальных заданий

Номер варианта	Номер модуля	Варианты задач
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	7	9

10	6	10
11	1	9
12	2	8
13	3	7
14	4	6
15	5	5
16	6	4
17	7	3
18	8	2
19	5	1
20	4	1
21	1	2
22	2	3
23	3	4
24	4	5
25	5	6
26	6	7
27	7	8
28	8	9
29	3	10
30	1	9

Варианты задач

Вариант 1

а) *Procedure BildTree*(var *T:Tree*);

Строит дерево арифметического выражения, заданного в ОПЗ.

Операнды — целочисленные константы.

Операции — «+», «-», «*» и «div».

б) *Procedure WritePrefix*(*T:Tree*);

Выводит арифметическое выражение в ППЗ.

в) *Function Calc*(*T:Tree*):integer;

Вычисляет значение по дереву арифметического выражения.

Вариант 2

а) *Procedure BildTree*(var *T:Tree*);

Строит дерево арифметического выражения, заданного в ППЗ. Операнды — целочисленные константы.

Операции — «+», «-», «*» и «div».

б) *Procedure WritePostfix*(*T:Tree*);

Выводит арифметическое выражение в ОПЗ.

в) *Function WriteCalc(T:Tree):integer;*

Вычисляет значение по дереву арифметического выражения и выводит результат выполнения каждой операции в виде:

<операнд><операция><операнд>=<значение>

Вариант 3

а) *Procedure BildBalansTree(const M: T_mas; var T:Tree);*

Строит упорядоченное сбалансированное дерево *T*.

M — упорядоченный массив.

б) *Function HTree(T:Tree):byte;*

Вычисляет высоту дерева.

в) *Procedure WriteTree(T:Tree);*

Выводит дерево по уровням (в *i*-ю строку вывода — вершины *i*-го уровня).

Вариант 4

а) *Procedure BildTree(var T:Tree);*

Строит дерево по его скобочному представлению.

б) *Procedure WriteWays(T:Tree);*

Выводит все пути от корня к листьям дерева (в *i*-ю строку вывода — *i*-й путь).

в) *Procedure WriteLeafs(T:Tree);*

Выводит листья дерева.

Вариант 5

а) *Procedure BildRegTree(var T:Tree);*

Строит упорядоченное дерево.

б) *Procedure WriteSequence1(T:Tree);*

Выводит упорядоченную по возрастанию последовательность, составленную из элементов дерева.

в) *Procedure WriteSequence2(T:Tree);*

Выводит упорядоченную по убыванию последовательность, составленную из элементов дерева.

Вариант 6

а) *Procedure BildTree(var T:Tree);*

Строит дерево в ширину.

б) *Function CalcNode(T:Tree; n:byte):byte;*

Вычисляет количество вершин дерева *T*, имеющих *n* сыновей.

в) *Procedure WriteMaxWay(T:Tree);*

Выводит самый длинный путь от корня до листа.

Вариант 7

a) *Procedure BildTree*(var *T:Tree*);

Строит дерево в глубину.

b) *Function CalcLevel*(*T:Tree*; *n:byte*):*byte*;

Определяет количество вершин в дереве *T* на *n*-ом уровне.

в) *Procedure WriteWays*(*T:Tree*);

Выводит все пути от листьев до корня (в *i*-ю строку вывода — *i*-ый путь).

Вариант 8

a) *Procedure BildRegTree*(var *T:Tree*);

Строит упорядоченное дерево.

b) *Procedure WriteSequence*(*T:Tree*; *k:integer*);

Выводит упорядоченную по возрастанию последовательность, составленную из элементов дерева, меньших *k*.

в) *Procedure WriteNodes*(*T:Tree*);

Выводит вершины, для которых левое и правое поддереве имеют равное количество вершин.

Вариант 9

a) *Procedure BildRegTree*(var *T:Tree*);

Строит упорядоченное дерево.

b) *Procedure WriteSequence*(*T:Tree*; *n:byte*);

Выводит упорядоченную по возрастанию последовательность, составленную из *n* элементов дерева (начиная с минимального).

в) *Procedure WriteNodes*(*T:Tree*);

Выводит количество вершин в левом и в правом поддереве для каждой вершины дерева.

Вариант 10

a) *Procedure BildTree*(var *T:Tree*);

Строит дерево по его скобочному представлению.

b) *Procedure CopyTree*(*T:Tree*; var *TNew:Tree*);

Копирует дерево *T* в *TNew*.

в) *Function CompTree*(*T1,T2:Tree*):*boolean*;

Возвращает *TRUE*, если деревья *T1* и *T2* одинаковые.

Модули

1. Дерево в динамической памяти (базовый тип определяется задачей).

Реализация на языке *Pascal*:

Unit Tree1;

Interface

```

Const TreeOk          = 0;
      TreeNotMem       = 1;
      TreeUnder        = 2;
Type BaseType         = ...; {определить !!!}
PtrEl   = ^Element;
Element = Record
          Data : BaseType;
          LSon,RSon : PtrEl;

      End;
Tree= ^PtrEl;
Var TreeError : 0..2;
Procedure InitTree(var T:Tree); {инициализация — создается элемент, ко-
торый будет содержать корень дерева}
Procedure CreateRoot(var T:Tree); {создание корня}
Procedure WriteDataTree(var T:Tree; E:BaseType); {запись данных}
Procedure ReadDataTree(var T:Tree; var E:BaseType); {чтение}
Function IsLSon(var T:Tree):boolean; {есть левый сын ?}
Function IsRSon(var T:Tree):boolean; {есть правый сын ?}
Procedure MoveToLSon(var T,TS:Tree); {перейти к левому сыну, где T —
адрес ячейки, содержащей адрес текущей вершины, TS — адрес ячейки,
содержащей адрес корня левого поддерева(левого сына)}
Procedure MoveToRSon(var T,TS:Tree); {перейти к правому сыну}
Function IsEmptyTree(var T:Tree):boolean; {пустое дерево ?}
Procedure DelTree(var T:Tree); {удаление листа}

```

Реализация на языке C:

```

#ifdef __TREE1_H
const TreeOk = 0;
const TreeNotMem = 1;
const TreeUnder = 2;
typedef /*определить !!!*/ BaseType;
typedef struct element *ptrel;
typedef struct element{basetype data;
                      ptrel LSon;
                      ptrel RSon;}

typedef PtrEl *Tree;
short TreeError;
void InitTree(Tree *T)// инициализация — создается элемент, который бу-
дет содержать корень дерева
void CreateRoot(Tree *T) //создание корня
void WriteDataTree(Tree *T, BaseType E) //запись данных

```

```

void ReadDataTree(Tree *T, BaseType *E)//чтение
int IsLSon(Tree *T)//1 — есть левый сын, 0 — нет
int IsRSon(Tree *T)//1 — есть правый сын, 0 — нет
void MoveToLSon(Tree *T, Tree *TS)// перейти к левому сыну, где T — ад-
рес ячейки, содержащей адрес текущей вершины, TS — адрес ячейки, со-
держашей адрес корня левого поддерева(левого сына)
void MoveToRSon(Tree *T, Tree *TS)//перейти к правому сыну
int IsEmptyTree(Tree *T)//1 — пустое дерево, 0 — не пустое
void DelTree(Tree *T)//удаление листа
#endif

```

2. Элементы дерева находятся в массиве MemTree, расположенном в статической памяти. Базовый тип зависит от задачи. «Свободные» элементы массива объединяются в список (ССЭ), на начало которого указывает левый сын первого элемента массива. В массиве MemTree могут располагаться несколько деревьев.

Реализация на языке *Pascal*:

```

Unit Tree2;
Interface
Const TreeOk          = 0;
      TreeNotMem       = 1;
      TreeUnder        = 2;
      SizeMem          = 100;
Type  BaseType         = ...; {определить !!!}
      Index = 0..SizeMem;
      PtrEl  = Index;
      Element = Record
                                Data : BaseType;
                                LSon,RSon : PtrEl;
      End;
      Tree      = ^PtrEl;
Var  TreeError : 0..2;
      MemTree: array[Index] of Element;
Procedure InitTree(var T:Tree); {инициализация}
Procedure CreateRoot(var T:Tree); {создание корня}
Procedure WriteDataTree(var T:Tree; E:BaseType); {запись данных}
Procedure ReadDataTree(var T:Tree; var E:BaseType); {чтение}
Function IsLSon(var T:Tree):boolean; {есть левый сын ?}
Function IsRSon(var T:Tree):boolean; {есть правый сын ?}

```

Procedure MoveToLSon(var T,TS:Tree); {перейти к левому сыну, где T — адрес ячейки, содержащей адрес текущей вершины, TS — адрес ячейки, содержащей адрес корня левого поддерева(левого сына)}

Procedure MoveToRSon(var T,TS:Tree); {перейти к правому сыну}

Function IsEmptyTree(var T:Tree):boolean; {пустое дерево ?}

Procedure DelTree(var T:Tree); {удаление листа}

Implementation

Procedure InitMem; forward; {связывает все элементы массива в список свободных элементов}

Function EmptyMem: boolean; forward; {возвращает TRUE, если в массиве нет свободных элементов}

Function NewMem: word; forward; {возвращает номер свободного элемента и исключает его из ССЭ}

Procedure DisposeMem(n:word); forward; {делает n-й элемент массива свободным и включает его в ССЭ}

Реализация на языке C:

```
#if !defined(__TREE2_H)
#define      SizeMem      100
const TreeOk =      0;
const  TreeNotMem = 1;
const  TreeUnder = 2;
typedef      /*определить !!!*/ BaseType;
typedef unsigned char PtrEl;
typedef struct element{basetype data;
                    ptrel LSon;
                    ptrel RSon;}
typedef PtrEl *Tree;
short TreeError;
Element MemTree[SizeMem];
void InitTree(Tree *T)// инициализация — создается элемент, который будет
содержать корень дерева
void CreateRoot(Tree *T) //создание корня
void WriteDataTree(Tree *T, BaseType E) //запись данных
void ReadDataTree(Tree *T,BaseType *E)//чтение
int IsLSon(Tree *T)//1 — есть левый сын, 0 — нет
int IsRSon(Tree *T)//1 — есть правый сын, 0 — нет
void MoveToLSon(Tree *T, Tree *TS)// перейти к левому сыну, где T — адрес
ячейки, содержащей адрес текущей вершины, TS — адрес ячейки, со-
держащей адрес корня левого поддерева(левого сына)
void MoveToRSon(Tree *T, Tree *TS)//перейти к правому сыну
int IsEmptyTree(Tree *T)//1 — пустое дерево,0 — не пустое
```

```

void DelTree(Tree *T)//удаление листа
void InitMem() /*связывает все элементы массива в список свободных
элементов*/
int EmptyMem() /*возвращает 1, если в массиве нет свободных элемен-
тов,0 — в противном случае*/
unsigned int NewMem() /*возвращает номер свободного элемента и ис-
ключает его из ССЭ*/
void DisposeMem(unsigned n)/*делает n-й элемент массива свободным и
включает его в ССЭ*/
#endif

```

3. Элементы дерева находятся в массиве, расположенном в динамической памяти. Базовый тип зависит от задачи. «Свободные» элементы массива объединяются в список (ССЭ), на начало которого указывает левый сын первого элемента массива. В массиве может находиться несколько деревьев.

Реализация на языке *Pascal*:

```

Unit Tree3;
Interface
Const TreeOk          = 0;
      TreeNotMem      = 1;
      TreeUnder       = 2;
Type  BaseType        = ...; {определить !!!}
      Index = 0..65520 div sizeof(BaseType);
      PtrEl  = Index;
      Element = Record
                      Data : BaseType;
                      LSon,RSon : PtrEl;
                      End;
      Mem = array[Index] of Element;
      Pmem  = ^Mem;
      Tree  = ^Ptr: PtrEl;

Var  TreeError : 0..2;
      Pbuf: Pmem; {указатель на массив}
      Size: word; {размер массива}
Procedure InitTree(var T:Tree); {инициализация памяти}
Procedure InitTree(var T:Tree); {инициализация дерева}
Procedure CreateRoot(var T:Tree); {создание корня}
Procedure WriteDataTree(var T:Tree; E:BaseType); {запись данных}
Procedure ReadDataTree(var T:Tree; var E:BaseType); {чтение}

```

```

Function IsLSon(var T:Tree):boolean; {есть левый сын ?}
Function IsRSon(var T:Tree):boolean; {есть правый сын ?}
Procedure MoveToLSon(var T,TS:Tree); {перейти к левому сыну}
Procedure MoveToRSon(var T,TS:Tree); {перейти к правому сыну}
Function IsEmptyTree(var T:Tree):boolean; {пустое дерево ?}
Procedure DelTree(var T:Tree); {удаление листа}

```

Реализация на языке C:

```

#ifndef __TREE3_H
#define Index 1000
const TreeOk = 0;
const TreeNotMem = 1;
const TreeUnder = 2;
typedef /*определить !!!*/ BaseType;
typedef unsigned char PtrEl;
typedef struct element{basetype data;
                        ptrEl LSon;
                        ptrEl RSon;}

typedef Element Mem[Index];
typedef Mem *Pmem;
typedef ptrEl* Tree;
short TreeError;
Pmem Pbuf; //указатель на массив//
unsigned Size; //размер массива
void InitTree(Tree *T)// инициализация дерева
void CreateRoot(Tree *T) //создание корня
void WriteDataTree(Tree *T, BaseType E) //запись данных
void ReadDataTree(Tree *T,BaseType *E)//чтение
int IsLSon(Tree *T)//1 — есть левый сын, 0 — нет
int IsRSon(Tree *T)//1 — есть правый сын, 0 — нет
void MoveToLSon(Tree *T, Tree *TS)// перейти к левому сыну, где Т — ад-
рес ячейки, содержащей адрес текущей вершины, TS — адрес ячейки, со-
держащей адрес корня левого поддерева(левого сына)
void MoveToRSon(Tree *T, Tree *TS)//перейти к правому сыну
int IsEmptyTree(Tree *T)//1 — пустое дерево,0 — не пустое
void DelTree(Tree *T)//удаление листа
#endif

```

4. Элементы дерева находятся в массиве, расположенном в динамической памяти. Базовый тип зависит от задачи. Каждый элемент массива имеет признак того, является ли он элементом дерева или «свободен». Корень дерева находится в первом элементе массива. Для вершины дерева, распо-

ложенной в n -м элементе массива, $L\text{Son}$ будет находиться в $(2*n)$ -м элементе, а $R\text{Son}$ — в $(2*n + 1)$ -м.

Реализация на языке *Pascal*:

```
Unit Tree4;
Interface
Const TreeOk          = 0;
      TreeNotMem       = 1;
      TreeUnder        = 2;
Type BaseType         = ...; {определить !!!}
      Index = 0..65520 div sizeof(BaseType);
      PtrEl  = Index;
      Element = Record
                      Data : BaseType;
                      Flag : Boolean; {FALSE — элемент свободен}
                      {TRUE - элемент дерева }
                      End;
      Mem = array[Index] of Element;
      Pmem= ^Mem;
      Tree= record
              Pbuf: Pmem; {указатель на массив}
              Size: word; {размер массива}
              Ptr: PtrEl
            end;
Var   TreeError : 0..2;
      Procedure InitTree(var T:Tree; Size: word); {инициализация}
      Procedure CreateRoot(var T:Tree); {создание корня}
      Procedure WriteDataTree(var T:Tree; E:BaseType); {запись данных}
      Procedure ReadDataTree(var T:Tree;var E:BaseType); {чтение}
      Function IsLSon(var T:Tree):boolean; {есть левый сын ?}
      Function IsRSon(var T:Tree):boolean; {есть правый сын ?}
      Procedure MoveToLSon(var T:Tree); {перейти к левому сыну}
      Procedure MoveToRSon(var T:Tree); {перейти к правому сыну}
      Function IsEmptyTree(var T:Tree):boolean; {пустое дерево ?}
      Procedure DelTree(var T:Tree); {удаление листа}
```

Реализация на языке *C*:

```
#if !defined(__TREE4_H)
#define Index 1000
const TreeOk = 0;
const TreeNotMem = 1;
const TreeUnder = 2;
```

```

typedef      /*определить !!!*/ BaseType;
typedef unsigned char PtrEl;
typedef struct element{basetype data;
                                short flag;}; //0 — элемент занят
                                           //1 — элемент свободен

typedef Element Mem[Index];
typedef Mem *Pmem;
typedef      struct Tree{Pmem Pbuf; //указатель на массив
                        unsigned Size; //размер массива
                        PtrEl Ptr;};

short TreeError;
void InitTree(Tree *T,unsigned size)// инициализация дерева
void CreateRoot(Tree *T) //создание корня
void WriteDataTree(Tree *T, BaseType E) //запись данных
void ReadDataTree(Tree *T,BaseType *E)//чтение
int IsLSon(Tree *T)//1 — есть левый сын, 0 — нет
int IsRSon(Tree *T)//1 — есть правый сын, 0 — нет
void MoveToLSon(Tree *T, Tree *TS)// перейти к левому сыну, где Т — ад-
рес ячейки, содержащей адрес текущей вершины, TS — адрес ячейки, со-
держащей адрес корня левого поддерева(левого сына)
void MoveToRSon(Tree *T, Tree *TS)//перейти к правому сыну
int IsEmptyTree(Tree *T)//1 — пустое дерево,0 — не пустое
void DelTree(Tree *T)//удаление листа
#endif

```

5. Дерево в динамической памяти (базовый тип — произвольный).

Реализация на языке *Pascal*:

```

Unit Tree5;
Interface
Const TreeOk          = 0;
      TreeNotMem      = 1;
      TreeUnder       = 2;
Type BaseType         = pointer;
      PtrEl   = ^Element;
      Element = Record
                                Data : BaseType;
                                LSon,RSon : PtrEl;
      End;
      Tree= ^PtrEl;
Var   TreeError : 0..2;
      SizeEl: word;

```


Procedure InitTree(var *T:Tree*; *Size: word*); {инициализация}
Procedure CreateRoot(var *T:Tree*); {создание корня}

Procedure WriteDataTree(var *T:Tree*; *E:BaseType*); {запись данных}
Procedure ReadDataTree(var *T:Tree*;var *E:BaseType*); {чтение}
Function IsLSon(var *T:Tree*):*boolean*; {есть левый сын ?}
Function IsRSon(var *T:Tree*):*boolean*; {есть правый сын ?}
Procedure MoveToLSon(var *T:Tree*); {перейти к левому сыну}
Procedure MoveToRSon(var *T:Tree*); {перейти к правому сыну}
Function IsEmptyTree(var *T:Tree*):*boolean*; {пустое дерево ?}
Procedure DelTree(var *T:Tree*); {удаление листа}

Реализация на языке C:

```
#if !defined(__TREE5_H)
const TreeOk = 0;
const TreeNotMem = 1;
const TreeUnder = 2;
typedef void* BaseType;
typedef struct element *ptrel;
typedef struct element{basetype data;
                        ptrel LSon;
                        ptrel RSon;}

typedef PtrEl *Tree;
unsigned Size;
short TreeError;
void InitTree(Tree *T,unsigned size)// инициализация дерева
void CreateRoot(Tree *T) //создание корня
void WriteDataTree(Tree *T, BaseType E) //запись данных
void ReadDataTree(Tree *T,BaseType *E)//чтение
int IsLSon(Tree *T)//1 — есть левый сын, 0 — нет
int IsRSon(Tree *T)//1 — есть правый сын, 0 — нет
void MoveToLSon(Tree *T, Tree *TS)// перейти к левому сыну, где T — ад-
рес ячейки, содержащей адрес текущей вершины, TS — адрес ячейки, со-
держащей адрес корня левого поддерева(левого сына)
void MoveToRSon(Tree *T, Tree *TS)//перейти к правому сыну
int IsEmptyTree(Tree *T)//1 — пустое дерево,0 — не пустое
void DelTree(Tree *T)//удаление листа
#endif
```

6. Элементы дерева находятся в массиве MemTree, расположенном в статической памяти. Базовый тип — произвольный. «Свободные» элементы массива объединяются в список (ССЭ), на начало которого указывает

левый сын первого элемента массива. В массиве *MemTree* могут располагаться несколько деревьев.

Реализация на языке *Pascal*:

```
Unit Tree6;
Interface
Const TreeOk          = 0;
      TreeNotMem       = 1;
      TreeUnder        = 2;
      SizeMem          = 100;
Type BaseType         = pointer;
      Index = 0..SizeMem;
      PtrEl  = Index;
      Element = Record
                                Data : BaseType;
                                LSon,RSon : PtrEl;
      End;
      Tree= ^PtrEl;
Var   TreeError : 0..2;
      MemTree: array[Index] of Element;
      SizeEl: word;
Procedure InitTree(var T:Tree; Size: word); {инициализация}
Procedure CreateRoot(var T:Tree); {создание корня}
Procedure WriteDataTree(var T:Tree; E:BaseType); {запись данных}
Procedure ReadDataTree(var T:Tree;var E:BaseType); {чтение}
Function IsLSon(var T:Tree):boolean; {есть левый сын ?}
Function IsRSon(var T:Tree):boolean; {есть правый сын ?}
Procedure MoveToLSon(var T:Tree); {перейти к левому сыну}
Procedure MoveToRSon(var T:Tree); {перейти к правому сыну}
Function IsEmptyTree(var T:Tree):boolean; {пустое дерево ?}
Procedure DelTree(var T:Tree); {удаление листа}
Implementation
Procedure InitMem; forward; {связывает все элементы массива в
                             список свободных элементов}
Function EmptyMem: boolean; forward; {возвращает TRUE, если в
                                     массиве нет свободных элементов}
Function NewMem: word; forward; {возвращает номер свободного
                                 элемента и исключает его из ССЭ}
Procedure DisposeMem(n:word); forward; {делает n-й элемент мас-
                                       сива свободным и включает его в ССЭ}
```

Реализация на языке *C*:

```
#if !defined(__TREE6_H)
```

```

#define Index 1000
const TreeOk = 0;
const TreeNotMem = 1;
const TreeUnder = 2;
typedef void *BaseType;
typedef unsigned char PtrEl;
typedef struct element{basetype data;
                        ptrrel LSon;
                        ptrrel RSon;}

typedef PtrEl *Tree;
unsigned Size;
short TreeError;
Element MemTree[Index];
void InitTree(Tree *T,unsigned size)// инициализация дерева
void CreateRoot(Tree *T) //создание корня
void WriteDataTree(Tree *T, BaseType E) //запись данных
void ReadDataTree(Tree *T,BaseType *E)//чтение
int IsLSon(Tree *T)//1 — есть левый сын, 0 — нет
int IsRSon(Tree *T)//1 — есть правый сын, 0 — нет
void MoveToLSon(Tree *T, Tree *TS)// перейти к левому сыну, где Т — ад-
рес ячейки, содержащей адрес текущей вершины, TS — адрес ячейки, со-
держащей адрес корня левого поддерева(левого сына)
void MoveToRSon(Tree *T, Tree *TS)//перейти к правому сыну
int IsEmptyTree(Tree *T)//1 — пустое дерево,0 — не пустое
void DelTree(Tree *T)//удаление листа
void InitMem() /*связывает все элементы массива в список свободных
элементов*/
int EmptyMem() /*возвращает 1, если в массиве нет свободных элемен-
тов,0 — в противном случае*/
unsigned int NewMem() /*возвращает номер свободного элемента и ис-
ключает его из ССЭ*/
void DisposeMem(unsigned n)/*делает n-й элемент массива свободным и
включает его в ССЭ*/
#endif

```

7. Элементы дерева находятся в массиве, расположенном в динамической памяти. Базовый тип - произвольный. «Свободные» элементы массива объединяются в список (ССЭ), на начало которого указывает левый сын первого элемента массива. В массиве может находиться несколько деревь-ев.

Реализация на языке *Pascal*:

```

Unit Tree7;
  Interface
Const TreeOk           = 0;
      TreeNotMem       = 1;
      TreeUnder        = 2;
Type  BaseType         = pointer;
      Index = 0..65520 div sizeof(BaseType);
      PtrEl  = Index;
      Element = Record
                          Data : BaseType;
                          LSon,RSon : PtrEl;
                        End;
      Mem = array[Index] of Element;
      Pmem= ^Mem;
      Tree=^Ptr: PtrEl;
Var   TreeError : 0..2;
      Pbuf: Pmem; {указатель на массив}
      SizeMem: word; {размер массива}
      SizeEl: word; {размер элемента}
Procedure InitTree(var T:Tree; Size: word); {инициализация}
Procedure CreateRoot(var T:Tree); {создание корня}
Procedure WriteDataTree(var T:Tree; E:BaseType); {запись данных}
Procedure ReadDataTree(var T:Tree;var E:BaseType); {чтение}
Function IsLSon(var T:Tree):boolean; {есть левый сын ?}
Function IsRSon(var T:Tree):boolean; {есть правый сын ?}
Procedure MoveToLSon(var T:Tree); {перейти к левому сыну}
Procedure MoveToRSon(var T:Tree); {перейти к правому сыну}
Function IsEmptyTree(var T:Tree):boolean; {пустое дерево ?}
Procedure DelTree(var T:Tree); {удаление листа}

```

Реализация на языке *C*:

```

#ifndef __TREE7_H
#define Index 1000
const TreeOk = 0;
const TreeNotMem = 1;
const TreeUnder = 2;
typedef void *BaseType;
typedef unsigned PtrEl;
typedef struct element{basetype data;
                      ptrel LSon;

```

8. Элементы дерева находятся в массиве, расположенном в динамической памяти. Базовый тип - произвольный. Каждый элемент массива имеет признак того, является ли он элементом дерева или «свободен». Корень дерева находится в первом элементе массива. Для вершины дерева, расположенной в n -м элементе массива, $LSon$ будет находиться в $(2*n)$ -м элементе, а $RSon$ — в $(2*n + 1)$ -м.

{ *TRUE* - элемент дерева }

```

        End;
    Mem = array[Index] of Element;
    Pmem= ^Mem;
    Tree= record
        Pbuf: Pmem; {указатель на массив}
        Size: word; {размер массива}
        Ptr: PtrEl
    end;
Var
    TreeError : 0..2;
    SizeEl: word; {размер элемента}
Procedure InitTree(var T:Tree; Size: word); {инициализация}
Procedure CreateRoot(var T:Tree); {создание корня}
Procedure WriteDataTree(var T:Tree; E:BaseType); {запись данных}
Procedure ReadDataTree(var T:Tree;var E:BaseType); {чтение}
Function IsLSon(var T:Tree):boolean; {есть левый сын ?}
Function IsRSon(var T:Tree):boolean; {есть правый сын ?}
Procedure MoveToLSon(var T:Tree); {перейти к левому сыну}
Procedure MoveToRSon(var T:Tree); {перейти к правому сыну}
Function IsEmptyTree(var T:Tree):boolean; {пустое дерево ?}
Procedure DelTree(var T:Tree); {удаление листа}

```

Реализация на языке C:

```

#ifndef __TREE8_H
#define Index 1000
const TreeOk = 0;
const TreeNotMem = 1;
const TreeUnder = 2;
typedef void *BaseType;
typedef unsigned PtrEl;
typedef struct element{basetype data;
                        short flag;}; //0 — элемент занят
                                           //1 — элемент свободен
typedef Element Mem[Index];
typedef Mem *Pmem;
typedef struct Tree{Pmem Pbuf; //указатель на массив
                   unsigned Size; //размер массива
                   PtrEl Ptr;};

short TreeError;
unsigned SizeEl; //размер элемента
void InitTree(Tree *T, unsigned size) // инициализация дерева
void CreateRoot(Tree *T) //создание корня
void WriteDataTree(Tree *T, BaseType E) //запись данных

```

```

void ReadDataTree(Tree *T, BaseType *E)//чтение
int IsLSon(Tree *T)//1 — есть левый сын, 0 — нет
int IsRSon(Tree *T)//1 — есть правый сын, 0 — нет
void MoveToLSon(Tree *T, Tree *TS)// перейти к левому сыну, где T — ад-
рес ячейки, содержащей адрес текущей вершины, TS — адрес ячейки, со-
держащей адрес корня левого поддерева(левого сына)
void MoveToRSon(Tree *T, Tree *TS)//перейти к правому сыну
int IsEmptyTree(Tree *T)//1 — пустое дерево, 0 — не пустое
void DelTree(Tree *T)//удаление листа
#endif

```

Назначение процедур и функций:

InitTree — инициализация дерева;
CreateRoot — создание корня;
WriteDataTree — запись данных в дерево;
ReadDataTree — чтение элемента дерева;
IsLSon — проверка на наличие левого сына;
IsRSon — проверка на наличие правого сына;
MoveToLSon — переход к левому сыну;
MoveToRSon — переход к правому сыну;
IsEmptyTree — проверка: дерево пусто или нет;
DelTree — удаление листа.

С о д е р ж а н и е о т ч е т а

1. Тема лабораторной работы.
2. Цель работы.
3. Характеристика СД типа «дерево» (п.1 задания).
4. Индивидуальное задание.
5. Текст модуля для реализации СД типа «дерево», текст программы для отладки модуля, тестовые данные результат работы программы.
6. Текст программы для решения задачи с использованием модуля, тестовые данные, результат работы программы.

Т е о р е т и ч е с к и е с в е д е н и я

Дерево — конечное непустое множество T , состоящее из одной или более вершин таких, что выполняются следующие условия:

- 1) Имеется одна специально обозначенная вершина, называемая корнем данного дерева.

2) Остальные вершины (исключая корень) содержатся в $m \geq 0$ попарно непересекающихся множествах T_1, T_2, \dots, T_m , каждое из которых в свою очередь является деревом. Деревья T_1, T_2, \dots, T_m называются поддеревьями данного корня.

Наиболее наглядным способом представления дерева является графический (рис.20), в котором вершины изображаются окружностями с вписанной в них информацией и корень дерева соединяется с корнями поддеревьев T_1, T_2, \dots, T_m дугой (линией).

Если подмножества T_1, T_2, \dots, T_m упорядочены, то дерево называют упорядоченным. Если два дерева считаются равными и тогда, когда они отличаются порядком подмножеств T_1, T_2, \dots, T_m , то такие деревья называются ориентированными деревьями. Конечное множество непересекающихся деревьев называется лесом.

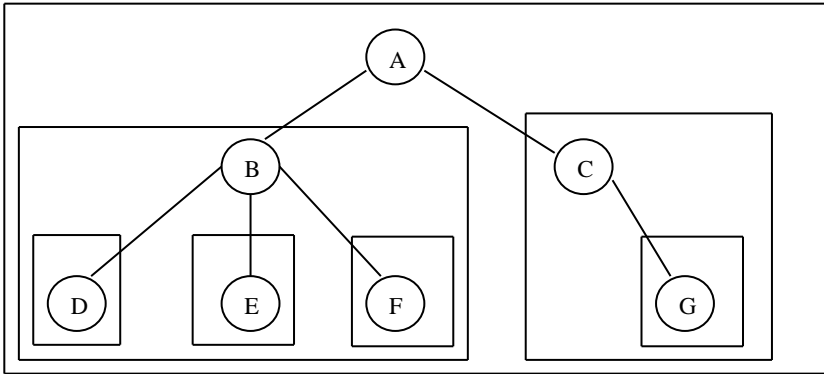


Рис.20. Графический способ представления дерева

Количество подмножеств для данной вершины называется *степенью* вершины. Если такое количество равно нулю, то вершина является *листом*. Максимальная степень вершины в дереве — *степень дерева*. *Уровень вершины* — длина пути от корня до вершины. Максимальная длина пути от корня до вершины определяет *высоту дерева* (количество уровней в дереве).

Бинарное дерево — конечное множество элементов, которое может быть пустым, состоящее из корня и двух непересекающихся бинарных деревьев, причем поддеревья упорядочены: левое поддерево и правое поддерево. В дальнейшем будем рассматривать только СД типа «бинарное дерево» (БД). Корень дерева будем называть «отцом», корень левого поддерева — «ле-

вым сыном», а корень правого поддерева — «правым сыном». На рис.21 приведен пример графического представления БД.

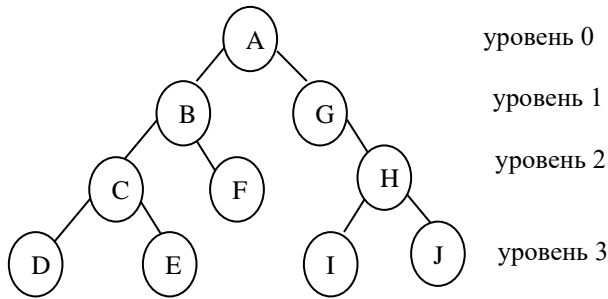


Рис.21. Графическое представление БД

БД — динамическая структура. Над СД БД определены следующие основные операции: инициализация, создание корня, запись данных, чтение данных, проверка — есть ли левый сын, проверка — есть ли правый сын, переход к левому сыну, переход к правому сыну, проверка — пустое ли дерево, удаление листа.

Кардинальное число СД БД определяется по формуле

$$CAR(БД) = (\sum (2i)! / ((i+1)(i!)^2)) \cdot CAR(BaseType) + 1, i=1...max,$$

где $CAR(BaseType)$ — кардинальное число элемента БД типа BaseType, max — максимальное количество элементов в БД (не всегда определено, т.к. может зависеть от объема свободной динамической памяти).

На *абстрактном* уровне БД представляет собой иерархическую структуру — дерево.

На *физическом* уровне БД реализуется связанной схемой хранения. Располагаться БД может в статической или динамической памяти.

Принципы размещения бинарного дерева в памяти ЭВМ

1. БД располагается в динамической памяти. Каждая вершина БД представляет собой СД типа «запись», содержащую информационную часть и адреса сыновей. Память под вершину БД выделяется по мере надобности при построении БД и освобождается при исключении вершины. Адрес корня дерева находится в специальной ячейке динамической памяти, адрес которой хранится в статической памяти. БД (см. рис.21) может быть представлено в памяти так, как показано на рис.22.

2. Для хранения вершин БД используется СД типа «массив». Массив может располагаться в статической или динамической памяти. Элементы массива представляют собой СД типа «запись», содержащую информаци-

онную часть и адреса сыновей вершины БД. Индекс элемента массива, содержащего корень дерева, находится в специальной ячейке динамической памяти, адрес которой хранится в статической памяти. Элементы массива, не являющиеся вершинами БД, объединяются в список свободных элементов (ССЭ), на первый элемент которого указывает правый сын первого элемента массива. При включении элемента в БД берется первый элемент ССЭ, а исключаемый элемент заносится в начало ССЭ. БД (рис.21) может быть представлено в памяти так, как показано на рис.23. При таком способе в одном массиве может быть размещено несколько БД.

3. Для хранения вершин БД используется СД типа «массив». Массив может располагаться в статической или динамической памяти. Элементы массива представляют собой СД типа «запись», содержащую информационную часть вершины БД и признак использования элемента массива — 1 — если элемент содержит вершину БД, 0 — если элемент свободен. Корнем дерева является элемент массива с индексом 1, индекс левого сына вычисляется по формуле $i*2$, где i — индекс отца, а правый сын располагается в следующем $(i*2 + 1)$ элементе массива. БД (см. рис.21) может быть представлено в памяти так, как показано на рис.24.

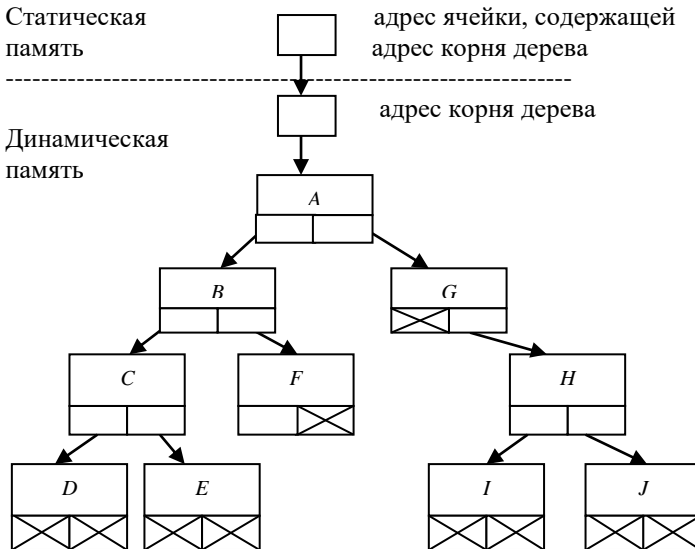


Рис.22. Представление БД в памяти



Рис.23. Представление БД в памяти

A	B	G	C	F		H	D	E					I	J
1	1	1	1	1	0	1	1	1	0	0	0	0	1	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Рис.24. Представление БД в памяти

Алгоритмы обхода бинарного дерева

Пусть в памяти ЭВМ находится БД. Задача обхода БД состоит в том, чтобы вывести информационную часть каждой вершины БД. Порядок прохождения вершин определяет алгоритм обхода БД. Рассмотрим некоторые алгоритмы обхода БД.

Обход бинарного дерева «в глубину» (в прямом порядке)

Чтобы пройти БД в прямом порядке нужно выполнить следующие три операции:

1. Попасть в корень.
2. Пройти в прямом порядке левое поддерево.
3. Пройти в прямом порядке правое поддерево.

Это рекурсивный алгоритм, т.к. поддеревья обрабатываются точно так же, как и БД. Нерекурсивный выход произойдет при достижении пустого дерева. Применяв алгоритм к БД на рис.21, получим последовательность: ABCDEFGHIJ.

Для обхода БД в прямом порядке можно использовать итеративный алгоритм. В этом алгоритме, для того, чтобы вернуться к правому сыну после прохождения левого поддерева используется стек для запоминания корня пройденного левого поддерева.

Итеративный алгоритм прохождения БД «в глубину»:

1. Инициализировать стек.
2. Пройти корень Т и включить его адрес в стек.
3. Если стек пуст, то перейти к п.5. Если есть левый сын вершины Т, то пройти его и занести его адрес в стек, иначе извлечь из стека адрес вершины Т и если у нее есть правый сын, то пройти его и занести в стек.
4. Перейти к п.3.
5. Конец алгоритма.

Обход бинарного дерева «в ширину» (по уровням)

В этом обходе сначала выписывается корень, затем вершины первого уровня, второго и т.д. до последнего. Выполнив обход БД на рис.21, получим последовательность: ABGCFHDEIJ.

В алгоритме обхода БД «в ширину» будем использовать очередь, в которую будут помещаться адреса вершин в порядке обхода.

Алгоритм прохождения БД «в ширину»:

1. Инициализировать очередь.
2. Занести адрес корня Т в очередь.
3. Если очередь пуста, то перейти к п.5. Извлечь из очереди адрес вершины Т и пройти ее. Если у нее есть левый сын, то занести его адрес в очередь. Если у нее есть правый сын, то занести его адрес в очередь.
4. Перейти к п.3.
5. Конец алгоритма.

Обход бинарного дерева в симметричном порядке

Для прохождения БД в симметричном порядке нужно выполнить следующие действия:

1. Пройти в симметричном порядке левое поддерево.
2. Попасть в корень.
3. Пройти в симметричном порядке правое поддерево.

Выполнив обход в симметричном порядке БД на рис.21, получим последовательность: DCEBFAGIHJ.

Для обхода БД в симметричном порядке, также как и для обхода в прямом порядке, можно применить итеративный алгоритм с использованием стека.

Итеративный алгоритм прохождения БД в симметричном порядке:

1. Инициализировать стек.

2. Адрес корня Т включить в стек.
3. Если стек пуст, то перейти к п.5. Если есть левый сын вершины Т, то занести его адрес в стек, иначе извлечь из стека адрес вершины Т, пройти ее и если у нее есть правый сын, то занести его в стек.
4. Перейти к п.3.
5. Конец алгоритма.

Обход бинарного дерева в обратном порядке

Для прохождения БД в обратном порядке нужно выполнить следующие действия:

1. Пройти в обратном порядке левое поддерево.
2. Пройти в обратном порядке правое поддерево.
3. Попасть в корень.

Это рекурсивный алгоритм, т.к. поддеревья обрабатываются точно так же, как и БД. Нерекурсивный выход произойдет при достижении пустого дерева. Применяв алгоритм к БД на рис.21, получим последовательность: DEC F B I J H G A.

Алгоритмы формирования бинарного дерева

Для того, чтобы сформировать в памяти ЭВМ БД, необходимо представить его в виде последовательности информационных частей вершин БД. Такую последовательность можно получить в результате обхода БД. Последовательность, полученная в результате обхода, неоднозначно определяет БД, т.к. не содержит информации о наличии сыновей у вершин БД. Чтобы получить такую информацию, введем в БД фиктивные вершины (с символом «.» в информационной части). Такие вершины соответствуют несуществующим сыновьям вершин БД. БД (см. рис.21) с фиктивными вершинами представлено на рис.25.

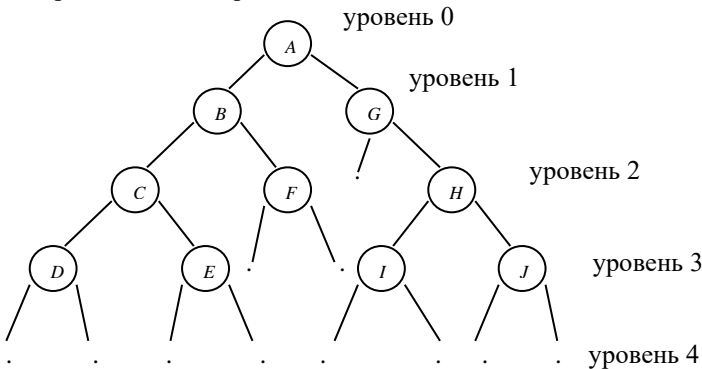


Рис.25. БД с фиктивными вершинами

При обходе БД (см. рис.25) в прямом порядке получим последовательность: ABCD..E..F..G.HI..J.. Из полученной последовательности следует, что вершины A,B,C,D и H имеют по два сына, причем левый сын записан непосредственно за отцом, вершины D,E,F,I и J — листья (за ними следуют две точки), вершина G не имеет левого сына (одна точка за этой вершиной), а правый сын — H.

Сформировать БД исходя из данной последовательности можно рекурсивным и итеративным способом, причем БД будет строиться «в глубину».

Рекурсивный алгоритм формирования бинарного дерева «в глубину»

1. Прочитать очередной символ последовательности.
2. Если прочитанный символ точка, то БД пустое, иначе создать корень, записать в него символ, построить левое поддерево, построить правое поддерево.

Итеративный алгоритм формирования бинарного дерева «в глубину»

1. Инициализировать стек.
2. Прочитать очередной символ последовательности.
3. Если прочитанный символ точка, то БД пустое, иначе создать корень, записать в него символ, корень поместить в стек.
4. Пока стек не пуст, выполнять п.5.
5. Прочитать символ, извлечь вершину из стека. Если для вершины левое поддерево не построено, то построить его в соответствии с п.6, если не построено правое поддерево, то построить его в соответствии с п.7.
6. Если символ точка, то левое поддерево пустое, вернуть вершину в стек, иначе создать корень левого поддерева, записать в него символ, вернуть в стек вершину, созданный корень поместить в стек.
7. Если символ точка, то правое поддерево пустое, иначе создать корень правого поддерева, записать в него символ, созданный корень поместить в стек.

Если при прохождении БД (см. рис.21) в прямом порядке сначала дерево, а потом все поддеревья заключать в скобки, то получим *скобочное представление* БД:

$(A(B(C(D())(E()))(F()))(G)(H(I())(J()))))$

Скобочное представление БД можно использовать в качестве исходных данных для формирования БД «в глубину», если пару символов ‘()’ считать точкой, а остальные скобки игнорировать.

При обходе БД (см. рис.25) «в ширину» получим последовательность: *ABGCF.HDE..IJ.*

Используя эту последовательность в качестве исходных данных можно сформировать БД, применив алгоритм формирования БД «в ширину».

Алгоритм формирования бинарного дерева «в ширину»

1. Инициализировать очередь.
2. Прочитать очередной символ последовательности.
3. Если прочитанный символ точка, то БД пустое, иначе создать корень, записать в него символ, корень поместить в очередь.
4. Пока очередь не пуста, выполнять п.5.
5. Взять вершину из очереди. Прочитать символ. Если символ точка, то левое поддерево пустое, иначе создать корень левого поддерева, записать в него символ и включить в очередь. Прочитать следующий символ. Если символ точка, то правое поддерево пустое, иначе создать корень правого поддерева, записать в него символ и включить в очередь.

При обходе БД (см. рис.25) в обратном порядке получим последовательность: *..D..EC..FB...I..JHGA*

Если ограничить эту последовательность концевым маркером, то ее можно будет использовать в качестве исходных данных для формирования БД «снизу вверх».

Алгоритм формирования бинарного дерева «снизу вверх»

1. Инициализировать очередь.
2. Последовательно обрабатывать символы последовательности, пока не достигнут концевой маркер.
3. Если прочитанный символ точка, то поместить в очередь пустое БД, иначе создать корень, записать в него символ, взять из очереди его левое, затем правое поддерево, полученное БД поместить в очередь.
4. Если по окончании работы алгоритма очередь окажется пустой, то сформировано пустое БД, иначе очередь будет содержать единственный элемент — корень сформированного дерева.

При обходе БД (см. рис.25) в симметричном порядке получим последовательность: *.D.C.E.B.F.A.G.I.H.J.*

Такую же последовательность получим при обходе в симметричном порядке БД на рис.26, поэтому она неоднозначно определяет БД и не может быть использована в качестве исходных данных для формирования БД.

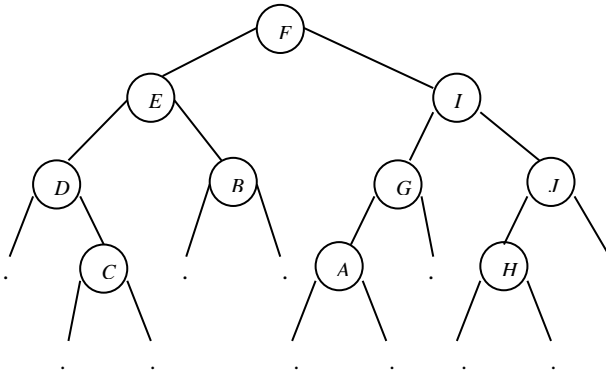


Рис.26. Бинарное дерево

Рассмотрим алгоритм формирования БД, в котором для любой вершины T_i значения всех вершин в левом поддереве должны быть меньше значения в вершине T_i , а в правом — больше. Такое дерево можно построить, обработав последовательность неповторяющихся значений.

Рекурсивный алгоритм формирования бинарного дерева

1. Прочитать элемент последовательности.
2. Если БД пустое, то создать корень и записать в него значение элемента, иначе включить элемент в левое поддерево, если его значение меньше значения корня, или включить в правое поддерево, если значение элемента больше значения корня.

Итеративный алгоритм формирования бинарного дерева

1. Пока в последовательности неповторяющихся значений есть элементы, читать и включать их в дерево в соответствии с п.2 или п.3.
2. Если БД пустое, то создать корень, записать в него значение.
3. Если значение элемента меньше значения корня и у корня есть левый сын, то перейти к левому сыну, считать его корнем и выполнять п.3, если же левого сына нет, то создать и записать в него значение элемента. Если значение элемента больше значения корня и у корня есть правый сын, то перейти к правому сыну, считать его корнем и выполнять п.3, если же правого сына нет, то создать и записать в него значение элемента.

Применив рассмотренные алгоритмы к последовательности целых чисел 20, 15, 23, 10, 5, 12, 30, 18, 44, 25 получим БД на рис.27.

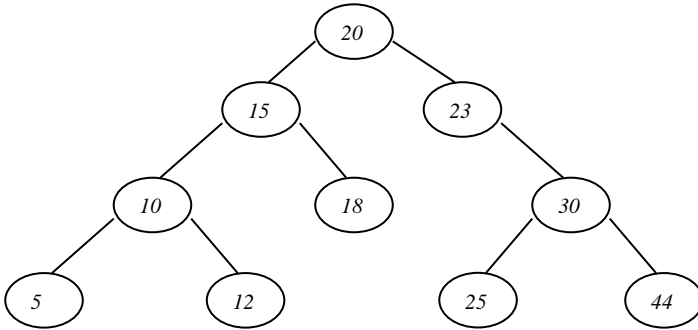


Рис.27. БД, построенное по итеративному алгоритму

Если последовательность будет упорядоченной, то БД вырождается в последовательность.

Пусть необходимо построить БД для последовательности из n элементов, имеющее минимальную высоту. Тогда все уровни дерева, кроме, может быть, последнего, будут заполнены. Для того чтобы построить такое БД, нужно элементы последовательности распределять поровну между левым и правым поддеревом. Такое распределение достаточно просто можно выполнить для упорядоченной по возрастанию последовательности.

Алгоритм формирования бинарного дерева минимальной высоты

Если в упорядоченной последовательности есть элементы, то создать корень и записать в него средний элемент последовательности, для первой части элементов последовательности (от первого до среднего элемента) построить левое поддерево, а для второй (от среднего до последнего) — правое поддерево.

БД, в котором относительно каждой вершины дерева количество вершин в левом и правом поддеревьях может отличаться максимум на единицу, называется идеально сбалансированным.

Сформировать идеально сбалансированное БД для заданной упорядоченной по возрастанию последовательности неповторяющихся элементов можно с помощью итеративного алгоритма. Средний элемент последовательности будет корнем БД и он разбивает последовательность на две части — левую и правую. Левое поддерево корня строится из левой части последовательности, а правое — из правой части точно также, как и БД для всей последовательности. Поэтому границы вновь образующихся последовательностей и адрес корня будем хранить в стеке.

Итеративный алгоритм формирования сбалансированного бинарного дерева

1. Создать корень, занести в него значение среднего элемента последовательности. Адрес корня, границы правой и левой части последовательности занести в стек.

2. Пока стек не пуст, извлечь из стека границы левой, правой части последовательности и адрес корня.

Если левой части нет, то нет левого поддерева для корня, иначе создать корень левого поддерева (левого сына), записать в него значение среднего элемента левой части, адрес левого сына и границы левой и правой части занести в стек.

Если правой части нет, то нет правого поддерева для корня, иначе создать корень правого поддерева (правого сына), записать в него значение среднего элемента правой части, адрес правого сына и границы левой и правой части занести в стек.

3. Конец алгоритма.

Представление алгебраических выражений бинарными деревьями

Любое алгебраическое выражение, которое содержит переменные, константы, знаки операций и скобки можно представить в виде бинарного дерева, в котором знак операции помещается в корне, первый операнд — в левом поддереве, а второй операнд — в правом. Скобки при этом опускаются. В результате все константы и переменные окажутся в листьях, а знаки операций — во внутренних вершинах. БД алгебраического выражения $a + 5 * b * (3 + c * d)$ представлено на рис.28.

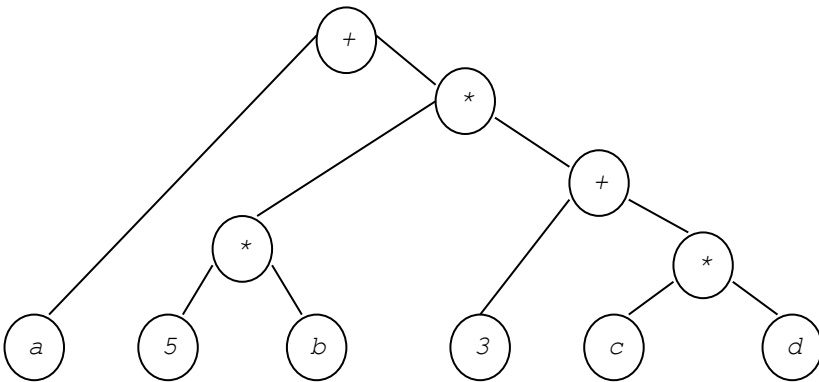


Рис.28. БД алгебраического выражения $a + 5 * b * (3 + c * d)$

Если выполнить обход БД (рис.28) в прямом порядке, то получим *прямую польскую запись* (ППЗ) алгебраического выражения:

$$+ a * * 5 b + 3 * c d$$

Используя ППЗ можно сформировать БД алгебраического выражения.

Алгоритм формирования бинарного дерева по прямой польской записи

1. Прочитать элемент ППЗ, создать корень, занести в него значение элемента.
2. Если прочитанный элемент — знак операции, то для корня построить левое, затем правое поддерево, иначе левое и правое поддерева пустые. Поддерева строятся точно также, как и БД в целом.

Если выполнить обход БД (см. рис.28) в обратном порядке, то получим *обратную польскую запись* (ОПЗ) алгебраического выражения:

$$a 5 b * 3 c d * + * +$$

Используя ОПЗ можно сформировать БД алгебраического выражения.

Алгоритм формирования бинарного дерева по обратной польской записи

1. Пока в ОПЗ есть элементы, прочитать очередной элемент, создать корень, занести в него значение элемента.
2. Если прочитанный элемент — знак операции, то извлечь из стека адреса правого и левого поддеревьев корня.
3. Адрес корня занести в стек.

По окончании работы алгоритма стек будет содержать единственный элемент — адрес корня БД.

К о н т р о л ь н ы е в о п р о с ы

1. Что такое бинарное дерево? Какие операции определены над бинарным деревом?
2. Как можно разместить бинарное дерево в памяти ЭВМ?
3. В чем заключается задача обхода бинарного дерева?
4. Опишите алгоритмы обхода бинарных деревьев.
5. Опишите алгоритмы формирования бинарных деревьев.
6. Разработайте алгоритм сортировки массива с использованием бинарного дерева. Определите порядок функции временной сложности алгоритма сортировки.
7. Опишите алгоритм поиска элемента в бинарном дереве. Определите порядок функции временной сложности алгоритма поиска.

Лабораторная работа № 8

Структуры данных типа «таблица» (Pascal/C)

Цель работы: изучить СД типа «таблица», научиться их программно реализовывать и использовать.

З а д а н и е

1. Для СД типа «таблица» определить:

1.1. Абстрактный уровень представления СД:

1.1.1. Характер организованности и изменчивости.

1.1.2. Набор допустимых операций.

1.2. Физический уровень представления СД:

1.2.1. Схему хранения.

1.2.2. Объем памяти, занимаемый экземпляром СД.

1.2.3. Формат внутреннего представления СД и способ его интерпретации.

1.2.4. Характеристику допустимых значений.

1.2.5. Тип доступа к элементам.

1. 3. Логический уровень представления СД:

Способ описания СД и экземпляра СД на языке программирования.

2. Реализовать СД типа «таблица» в соответствии с вариантом индивидуального задания (табл.18) в виде модуля.

3. Разработать программу для решения задачи в соответствии с вариантом индивидуального задания (см. табл.18) с использованием модуля, полученного в результате выполнения пункта 2 задания.

Таблица 18

Варианты индивидуальных заданий

Номер варианта	Номер модуля	Задача
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7

Окончание табл.18

8	8	8
9	9	9
10	10	10
11	11	9
12	11	8
13	10	7
14	9	6
15	8	5
16	7	4
17	6	3
18	5	2
19	4	1
20	3	2
21	2	3
22	1	4
23	11	5
24	10	6
25	9	7
26	8	8
27	7	9
28	6	10
29	5	1
30	4	2

Задачи

1. Текст программы на некотором алгоритмическом языке может содержать символы-разделители, служебные слова, числовые константы и идентификаторы (слова, начинающиеся не с цифры и не являющиеся служебными). Вывести все идентификаторы, которые встречаются в программе.

Исходные данные: текстовые файлы, содержащие

- а) текст программы;
- б) символы-разделители;
- в) служебные слова.

Для хранения символов-разделителей и служебных слов использовать таблицы.

2. Текст программы на некотором алгоритмическом языке может содержать символы-разделители, служебные слова, числовые константы и идентификаторы (слова, начинающиеся не с цифры и не являющиеся служебными). Все идентификаторы, используемые в программе, должны быть описаны до первого появления в тексте программы служебного слова «*BEGIN*». Идентификатор считать описанным, если он встречается в тексте программы до первого слова «*BEGIN*», причем только один раз. Вывести все идентификаторы, которые описаны более одного раза и неописанные идентификаторы.

Исходные данные: текстовые файлы, содержащие

- а) текст программы;
- б) символы-разделители;
- в) служебные слова.

Для хранения символов-разделителей, служебных слов и идентификаторов использовать таблицы.

3. Текст программы на некотором алгоритмическом языке может содержать символы-разделители, служебные слова, числовые константы и идентификаторы (слова, начинающиеся не с цифры и не являющиеся служебными). Проверить ошибки в записи идентификаторов и констант, парность служебных слов: «*BEGIN*» и «*END*», «*IF*» и «*THEN*», «*FOR*» и «*DO*», и скобок: «(» и «)», «[» и «]». Проверку констант выполнить с помощью стандартной процедуры *VAL*. Для проверки парности служебных слов и символов-разделителей использовать динамический массив из *KP* целочисленных элементов, где *KP* — количество парных служебных слов и символов-разделителей. Сначала все элементы массива обнуляются. Если встречается первое слово *i*-й пары, то *i*-й элемент массива увеличивается на единицу, а если второе слово — уменьшается. После обработки текста программы все элементы массива должны быть нулевыми. Парные служебные слова и символы-разделители хранить в таблице. Ключ элемента таблицы — парное служебное слово, информационная часть содержит $+i$ для первого слова *i*-й пары и $-i$ для второго слова. Информацию о символах-разделителях и парных служебных словах прочитывать из текстовых файлов.

4. Текст программы на некотором алгоритмическом языке может содержать символы-разделители, служебные слова, числовые константы и идентификаторы (слова, начинающиеся не с цифры и не являющиеся служебными). Проверить ошибки в записи идентификаторов и констант и сочетаемость рядом стоящих служебных слов, символов-разделителей, числовых констант и идентификаторов. Например, пары «*BEGIN x*» и «*ELSE BEGIN*» — сочетаемы, а пары «*x BEGIN*» и «*BEGIN ELSE*» — несочетаемы.

Проверку констант выполнить с помощью стандартной процедуры *VAL*. Проверку сочетаемости — с помощью таблицы сочетаемости. Ключ элемента таблицы — служебное слово или символ-разделитель, информационная часть — код служебного слова или символа-разделителя и множество кодов слов, которые могут непосредственно следовать за ключевым словом. Код константы — 0, идентификатора — 1. Для констант и идентификаторов также сформировать множества кодов слов, которые могут следовать за ними. Информацию для таблицы сочетаемости прочитать из файла.

5. Вычислить алгебраическое выражение, заданное ОПЗ. Выражение может содержать константы, вещественные переменные, знаки операций: «+», «-», «*», «/» и функции «*sin(x)*», «*cos(x)*», «*arctan(x)*», «*abs(x)*», «*exp(x)*», «*ln(x)*», «*sqr(x)*» и «*sqrt(x)*». Для хранения значений переменных использовать таблицу. Ключ элемента таблицы — имя переменной, информационная часть — значение переменной. Если текущей переменной нет в таблице, то ввести значение и включить ее в таблицу. Для вычисления значения выражения использовать стек.

ОПЗ выражения $b \cdot a + 5 \cdot (b + \sin(\text{sqr}(a)))$ имеет вид:

$$b \ a \cdot 5 \ b \ a \ \text{sqr} \ \sin \ + \ * \ +$$

6. Написать интерпретатор языка вычисления арифметических вещественных выражений, заданных ОПЗ. В языке три оператора:

- 1) <оператор ввода> ::= <переменная> *Read*
- 2) <оператор вывода> ::= <переменная> *Write*
- 3) <оператор присваивания> ::= <переменная> = <ОПЗ>

В строке текста программы записывается только один оператор. ОПЗ может содержать константы, вещественные переменные и знаки операций «+», «-», «*» и «/». Для хранения значений переменных использовать таблицу. Ключ элемента таблицы — имя переменной, информационная часть — значение переменной. Если переменной, записанной в операторе вывода или в ОПЗ нет в таблице, то ошибка, иначе определить значение переменной и включить ее в таблицу. Для вычисления выражений использовать стек.

Пример текста программы на языке вычисления арифметических вещественных выражений:

```

a Read
b Read
c Read
d = a b * 3.5 +
e = d d * c * a + 2 /

```

$d = a \ b + 2 \ d \ * \ e - *$
d Write
e Write

7. Написать интерпретатор языка арифметических вычислений. Язык содержит команды ввода и вывода значений вещественных переменных, команду пересылки константы или значения переменной в другую переменную, арифметические команды сложения, вычитания, умножения и деления. Команды ввода (*IN*) и вывода (*OUT*) имеют один операнд, команда пересылки (*MOV*) — два операнда, первый из которых — имя переменной, в которую пересылается второй операнд, арифметические команды (*ADD*, *SUB*, *MUL*, *DIV*) — два операнда, в первом сохраняется результат. В каждой строке программы — одна команда. Команды и операнды разделяются пробелами. Текст программы находится в текстовом файле. Значения переменных хранятся в таблице. Ключ элемента таблицы — имя переменной, информационная часть — значение переменной. Если операнда команды ввода или первого операнда арифметических команд и команды пересылки нет в таблице, то определить его значение и занести в таблицу. Если операнда команды вывода или второго операнда арифметических команд и команды пересылки нет в таблице, то выдать сообщение об ошибке.

Пример текста программы на языке арифметических вычислений:

IN a
IN b
IN c
MOV d a
MUL d b
DIV c a
SUB b c
MUL b 3.7
ADD d b
OUT d

8. Текстовый файл содержит текст на русском языке. В тексте могут встречаться числа, записанные в словесной форме. Преобразовать файл, заменив словесную запись чисел числовой. Например, файл:

Получил триста двадцать пять рублей пятнадцать копеек.
 преобразовать в файл:

Получил 325 рублей 15 копеек.

Для преобразования чисел использовать таблицу. Ключ элемента — словесное название числа («один», «два»,..., «десять», «одиннадцать»,...,

«двадцать»,..., «девяносто», «сто», «двести», «триста»,..., «девятьсот»), информационная часть — числовое значение ключа. Информацию в таблицу загрузить из текстового файла.

9. Текстовый файл содержит текст на русском языке. В тексте могут встречаться числа, записанные в числовой форме. Преобразовать файл, заменив числовую запись чисел словесной. Например, файл:

Получил 325 рублей 15 копеек.

преобразовать в файл:

Получил триста двадцать пять рублей пятнадцать копеек.

Для преобразования чисел использовать таблицу. Ключ элемента — числовое значение числа («1», «2»,..., «10», «11»,..., «12»,..., «20»,..., «90», «100», «200», «300»,..., «900»), информационная часть — словесное название ключа. Информацию в таблицу загрузить из текстового файла.

10. Текстовый файл содержит формулу химического соединения (не обязательно существующего). Название элемента начинается с большой буквы. Если количество некоторых элементов в соединении больше одного, то после названия указывается число элементов. Примеры формул:

H₂O, *HCl*, *O₂*, *H₂SO₄*, *O₄H₂S*, *NaCl*.

Для правильных формул получить молекулярный вес соединения. Использовать таблицу, ключ элемента которой представляет собой название химического элемента, а информационная часть — вес элемента. Информацию в таблицу загрузить из текстового файла.

Модули

1. Неупорядоченная таблица на последовательном линейном списке.

Реализация на языке *Pascal*:

Unit Table1;

Interface

uses list8; {см лаб.раб. №5}

Const TableOk = 0;

TableNotMem = 1;

TableUnder = 2;

Type Table=list;

Func=function(var A,B: pointer):shortint; {Сравнивает ключи элементов таблицы, адреса которых находятся в параметрах A и B. Возвращает -1, если ключ элемента по адресу A меньше ключа элемента по адресу B, 0 — если ключи равны и +1 — если ключ элемента по адресу A больше ключа элемента по адресу B.}

```

Var TableError : 0..2;
Procedure InitTable(var T:Table; SizeMem, SizeEl:Word);
Function EmptyTable(var T:Table):boolean; {Возвращает TRUE , если таб-
лица пуста, иначе — FALSE}
Function PutTable(var T:Table; El:Element;
var F: Func):boolean; {Включение элемента в таблицу. Возвращает
TRUE , если элемент включен в таблицу, иначе — FALSE (если в таблице
уже есть элемент с заданным ключом или нехватает памяти).}
Function GetTable(var T:Table; var El:Element; Key:T_Key;
var F: Func):boolean; {Исключение элемента. Возвра-
щает TRUE , если элемент с ключом Key был в таблице, иначе — FALSE}
Function ReadTable(var T:Table; var El:Element; Key:T_Key;
var F: Func):boolean; {Чтение элемента. Возвращает
TRUE , если элемент с ключом Key есть в таблице, иначе — FALSE}
Function WriteTable(var T:Table; var El:Element; Key:T_Key;
var F: Func):boolean; {Изменение элемента. Воз-
вращает TRUE , если элемент с ключом Key есть в таблице, иначе —
FALSE}
Procedure DoneTable(var T:Table); {Уничтожение таблицы}

```

Реализация на языке C:

```

#ifdef __TABLE1_H
#define __TABLE1_H
#include "list8.h" // Смотреть лаб.раб. №5
const TableOk = 0;
const TableNotMem = 1;
const TableUnder = 2;
typedef List Table;
typedef ... T_Key; // Определить тип ключа
typedef int (*func)(void*, void*); /* Сравнивает ключи элементов таблицы,
адреса которых находятся в параметрах a и b. Возвращает -1, если ключ
элемента по адресу a меньше ключа элемента по адресу b, 0 — если ключи
равны и +1 — если ключ элемента по адресу a больше ключа элемента по
адресу b */
int TableError; // 0..2
void InitTable(Table *T, unsigned SizeMem, unsigned SizeEl);
inline int EmptyTable(Table *T); /* Возвращает 1 , если таблица пуста,
иначе — 0 */
int PutTable(Table *T, void *E, func f); /* Включение элемента в таблицу.
Возвращает 1 , если элемент включен в таблицу, иначе — 0 (если в таблице
уже есть элемент с заданным ключом или нехватает памяти) */

```

```

int GetTable(Table *T, void *E, T_Key Key, func f); /* Исключение элемен-
та. Возвращает 1 , если элемент с ключом Key был в таблице, иначе — 0 */
int ReadTable(Table *T, void *E, T_Key Key, func f); /* Чтение элемента.
Возвращает 1 , если элемент с ключом Key есть в таблице, иначе — 0 */
int WriteTable(Table *T, void *E, T_Key Key, func f); /* Изменение элемен-
та. Возвращает 1 , если элемент с ключом Key есть в таблице, иначе — 0 */
void DoneTable(Table *T); /*Удаление таблицы из динамической памяти */
#endif

```

2. Неупорядоченная таблица на односвязном линейном списке.

Реализация на языке *Pascal*:

Unit Table2;

Interface

uses list3; {см лаб.паб. №5}

Const TableOk = 0;

TableNotMem = 1;

TableUnder = 2;

Type Table=list;

Func=function(var A,B: pointer):shortint; {Сравнивает ключи элементов таблицы, адреса которых находятся в параметрах A и B. Возвращает -1, если ключ элемента по адресу A меньше ключа элемента по адресу B, 0 — если ключи равны и +1 — если ключ элемента по адресу A больше ключа элемента по адресу B.}

Var TableError : 0..2;

Procedure InitTable(var T:Table; SizeMem, SizeEl:Word);

Function EmptyTable(var T:Table):boolean; {Возвращает TRUE — если таблица пуста, иначе FALSE}

Function PutTable(var T:Table; El:Element; var F: Func):boolean; {Включение элемента в таблицу. Возвращает TRUE , если элемент включен в таблицу, иначе — FALSE (если в таблице уже есть элемент с заданным ключом или нехватает памяти).}

Function GetTable(var T:Table; var El:Element; Key:T_Key; var F: Func):boolean; {Исключение элемента. Возвращает TRUE , если элемент с ключом Key был в таблице, иначе — FALSE}

Function ReadTable(var T:Table; var El:Element; Key:T_Key; var F: Func):boolean; {Чтение элемента. Возвращает TRUE , если элемент с ключом Key есть в таблице, иначе — FALSE}

Function WriteTable(var T:Table; var El:Element; Key:T_Key;

var F: Func):boolean; {Изменение элемента. Возвращает *TRUE* , если элемент с ключом *Key* есть в таблице, иначе — *FALSE*}
Procedure DoneTable(var T:Table); {Уничтожение таблицы}

Реализация на языке C:

```
#if !defined(__TABLE1_H)
#define __TABLE1_H
#include "list3.h" // Смотреть лаб.раб. №5
const TableOk = 0;
const TableNotMem = 1;
const TableUnder = 2;
typedef List Table;
typedef ... T_Key; // Определить тип ключа
typedef int (*func)(void*, void*); /* Сравнивает ключи элементов таблицы,
адреса которых находятся в параметрах a и b. Возвращает -1, если ключ
элемента по адресу a меньше ключа элемента по адресу b, 0 — если ключи
равны и +1 — если ключ элемента по адресу a больше ключа элемента по
адресу b */
int TableError; // 0..2
void InitTable(Table *T, unsigned SizeMem, unsigned SizeEl);
inline int EmptyTable(Table *T); /* Возвращает 1 , если таблица пуста,
иначе — 0 */
int PutTable(Table *T, void *E, func f); /* Включение элемента в таблицу.
Возвращает 1 , если элемент включен в таблицу, иначе — 0 (если в таблице
уже есть элемент с заданным ключом или нехватает памяти) */
int GetTable(Table *T, void *E, T_Key Key, func f); /* Исключение элемен-
та. Возвращает 1 , если элемент с ключом Key был в таблице, иначе — 0 */
int ReadTable(Table *T, void *E, T_Key Key, func f); /* Чтение элемента.
Возвращает 1 , если элемент с ключом Key есть в таблице, иначе — 0 */
int WriteTable(Table *T, void *E, T_Key Key, func f); /* Изменение элемен-
та. Возвращает 1 , если элемент с ключом Key есть в таблице, иначе — 0 */
void DoneTable(Table *T); //Удаление таблицы из динамической памяти
#endif
```

3. Упорядоченная таблица на последовательном линейном списке. Использовать алгоритм блочного поиска (см лаб.раб.№4).

Реализация на языке Pascal:

Unit Table3;

Interface

uses list8; {см лаб.раб. №5}

Const TableOk = 0;

TableNotMem = 1;

TableUnder = 2;

Type Table=list;

Func=function(var *A,B: pointer*):shortint; {Сравнивает ключи элементов таблицы, адреса которых находятся в параметрах *A* и *B*. Возвращает -1, если ключ элемента по адресу *A* меньше ключа элемента по адресу *B*, 0 — если ключи равны и +1 — если ключ элемента по адресу *A* больше ключа элемента по адресу *B*.}

Var TableError : 0..2;

Procedure InitTable(var *T:Table*; *SizeMem*, *SizeEl:Word*);

Function EmptyTable(var *T:Table*):boolean; {Возвращает TRUE , если таблица пуста, иначе — FALSE}

Function PutTable(var *T:Table*; *El:Element*; var *F: Func*):boolean; {Включение элемента в таблицу. Возвращает TRUE , если элемент включен в таблицу, иначе — FALSE (если в таблице уже есть элемент с заданным ключом или нехватает памяти).}

Function GetTable(var *T:Table*; var *El:Element*; *Key:T_Key*;

var *F: Func*):boolean; {Исключение элемента. Возвращает TRUE , если элемент с ключом *Key* был в таблице, иначе — FALSE}

Function ReadTable(var *T:Table*; var *El:Element*; *Key:T_Key*;

var *F: Func*):boolean; {Чтение элемента. Возвращает TRUE , если элемент с ключом *Key* есть в таблице, иначе — FALSE}

Function WriteTable(var *T:Table*; var *El:Element*; *Key:T_Key*;

var *F: Func*):boolean; {Изменение элемента. Возвращает TRUE , если элемент с ключом *Key* есть в таблице, иначе — FALSE}

Procedure DoneTable(var *T:Table*); {Уничтожение таблицы}

Реализация на языке C:

```
#if !defined(__TABLE1_H)
```

```
#define __TABLE1_H
```

```
#include "list8.h" // Смотреть лаб.раб. №5
```

```
const TableOk = 0;
```

```
const TableNotMem = 1;
```

```
const TableUnder = 2;
```

```
typedef List Table;
```

```
typedef ... T_Key; // Определить тип ключа
```

```
typedef int (*func)(void*, void*); /* Сравнивает ключи элементов таблицы,
адреса которых находятся в параметрах a и b. Возвращает -1, если ключ
элемента по адресу a меньше ключа элемента по адресу b, 0 — если ключи
```

равны и +1 — если ключ элемента по адресу a больше ключа элемента по адресу b */

```
int TableError; // 0..2
void InitTable(Table *T, unsigned SizeMem, unsigned SizeEl);
inline int EmptyTable(Table *T); /* Возвращает 1 , если таблица пуста,
иначе — 0 */
int PutTable(Table *T, void *E, func f); /* Включение элемента в таблицу.
Возвращает 1 , если элемент включен в таблицу, иначе — 0 (если в таблице
уже есть элемент с заданным ключом или нехватает памяти) */
int GetTable(Table *T, void *E, T_Key Key, func f); /* Исключение элемен-
та. Возвращает 1 , если элемент с ключом Key был в таблице, иначе — 0 */
int ReadTable(Table *T, void *E, T_Key Key, func f); /* Чтение элемента.
Возвращает 1 , если элемент с ключом Key есть в таблице, иначе — 0 */
int WriteTable(Table *T, void *E, T_Key Key, func f); /* Изменение элемен-
та. Возвращает 1 , если элемент с ключом Key есть в таблице, иначе — 0 */
void DoneTable(Table *T); //Удаление таблицы из динамической памяти
#endif
```

4. Упорядоченная таблица на последовательном линейном списке. Использовать алгоритм бинарного поиска(см лаб.раб.№4).

Реализация на языке *Pascal*:

Unit Table4;

Interface

uses list6; {см лаб.раб. №5}

Const TableOk = 0;

TableNotMem = 1;

TableUnder = 2;

Type Table=list;

Func=function(var A,B: pointer):shortint; {Сравнивает ключи элементов таблицы, адреса которых находятся в параметрах A и B. Возвращает -1, если ключ элемента по адресу A меньше ключа элемента по адресу B, 0 — если ключи равны и +1 — если ключ элемента по адресу A больше ключа элемента по адресу B.}

Var TableError : 0..2;

Procedure InitTable(var T:Table; SizeMem, SizeEl:Word);

Function EmptyTable(var T:Table):boolean; {Возвращает TRUE , если таблица пуста, иначе — FALSE}

Function PutTable(var T:Table; El:Element;

var F: Func):boolean; {Включение элемента в таблицу. Возвращает TRUE , если элемент включен в таблицу, иначе — FALSE (если в таблице уже есть элемент с заданным ключом или нехватает памяти).}

```

Function GetTable(var T:Table; var El:Element; Key:T_Key;
                 var F: Func):boolean; {Исключение элемента. Возвращает TRUE, если элемент с ключом Key был в таблице, иначе — FALSE}
Function ReadTable(var T:Table; var El:Element; Key:T_Key;
                  var F: Func):boolean; {Чтение элемента. Возвращает TRUE, если элемент с ключом Key есть в таблице, иначе — FALSE}
Function WriteTable(var T:Table; var El:Element; Key:T_Key;
                   var F: Func):boolean; {Изменение элемента. Возвращает TRUE, если элемент с ключом Key есть в таблице, иначе — FALSE}
Procedure DoneTable(var T:Table); {Уничтожение таблицы}

```

Реализация на языке C:

```

#ifdef __TABLE1_H
#define __TABLE1_H
#include "list6.h" // Смотреть лаб. раб. №5
const TableOk = 0;
const TableNotMem = 1;
const TableUnder = 2;
typedef List Table;
typedef ... T_Key; // Определить тип ключа
typedef int (*func)(void*, void*); /* Сравнивает ключи элементов таблицы,
адреса которых находятся в параметрах a и b. Возвращает -1, если ключ
элемента по адресу a меньше ключа элемента по адресу b, 0 — если ключи
равны и +1 — если ключ элемента по адресу a больше ключа элемента по
адресу b */
int TableError; // 0..2
void InitTable(Table *T, unsigned SizeMem, unsigned SizeEl);
inline int EmptyTable(Table *T); /* Возвращает 1, если таблица пуста,
иначе — 0 */
int PutTable(Table *T, void *E, func f); /* Включение элемента в таблицу.
Возвращает 1, если элемент включен в таблицу, иначе — 0 (если в таблице
уже есть элемент с заданным ключом или нехватает памяти) */
int GetTable(Table *T, void *E, T_Key Key, func f); /* Исключение элемен-
та. Возвращает 1, если элемент с ключом Key был в таблице, иначе — 0 */
int ReadTable(Table *T, void *E, T_Key Key, func f); /* Чтение элемента.
Возвращает 1, если элемент с ключом Key есть в таблице, иначе — 0 */
int WriteTable(Table *T, void *E, T_Key Key, func f); /* Изменение элемен-
та. Возвращает 1, если элемент с ключом Key есть в таблице, иначе — 0 */
void DoneTable(Table *T); // Удаление таблицы из динамической памяти
#endif

```

5. Упорядоченная таблица на односвязном линейном списке.

Реализация на языке *Pascal*:

```

Unit Table5;
  Interface
uses list2; {см лаб.раб. №5}
Const TableOk      = 0;
      TableNotMem   = 1;
      TableUnder    = 2;
Type Table=list;
Var  TableError : 0..2;
Procedure InitTable(var T:Table; SizeMem, SizeEl:Word);
Function EmptyTable(var T:Table):boolean; {Возвращает TRUE , если таб-
лица пуста, иначе — FALSE}
Function PutTable(var T:Table; El:Element; var F: Func):boolean; {Вклю-
чение элемента в таблицу. Возвращает TRUE , если элемент включен в таб-
лицу, иначе — FALSE (если в таблице уже есть элемент с заданным ключом
или нехватает памяти).}
Function GetTable(var T:Table; var El:Element; Key:T_Key;
      var F: Func):boolean; {Исключение элемента. Возвра-
щает TRUE , если элемент с ключом Key был в таблице, иначе — FALSE}
Function ReadTable(var T:Table; var El:Element; Key:T_Key;
      var F: Func):boolean; {Чтение элемента. Возвращает
TRUE , если элемент с ключом Key есть в таблице, иначе — FALSE}
Function WriteTable(var T:Table; var El:Element; Key:T_Key;
      var F: Func):boolean; {Изменение элемента. Возвра-
щает TRUE , если элемент с ключом Key есть в таблице, иначе — FALSE}
Procedure DoneTable(var T:Table); {Уничтожение таблицы}

```

Реализация на языке *C*:

```

#ifndef __TABLE5_H
#define __TABLE5_H
#include "list2.h" // Смотреть лаб.раб. №5
const TableOk = 0;
const TableNotMem = 1;
const TableUnder = 2;
typedef List Table;
typedef ... T_Key; // Определить тип ключа
int TableError; // 0..2
void InitTable(Table *T);
int EmptyTable(Table *T); // Возвращает 1 , если таблица пуста, иначе — 0

```



```

    int PutTable(Table *T, BaseType E); /* Включение элемента в таблицу.
    Возвращает 1, если элемент включен в таблицу, иначе — 0 (если в таблице
    уже есть элемент с заданным ключом или нехватает памяти) */
    int GetTable(Table *T, BaseType *E, T_Key Key); /* Исключение элемента.
    Возвращает 1, если элемент с ключом Key был в таблице, иначе — 0 */
    int ReadTable(Table *T, BaseType *E, T_Key Key); /* Чтение элемента.
    Возвращает 1, если элемент с ключом Key есть в таблице, иначе — 0 */
    int WriteTable(Table *T, BaseType *E, T_Key Key); /* Изменение элемента.
    Возвращает 1, если элемент с ключом Key есть в таблице, иначе — 0 */
    void DoneTable(Table *T); // Уничтожение таблицы
#endif

```

6. Упорядоченная таблица на бинарном дереве.

Реализация на языке *Pascal*:

```

Unit Table6;
Interface
uses Tree1; {см лаб.раб. №7}
Const TableOk      = 0;
      TableNotMem   = 1;
      TableUnder    = 2;
Type Table=tree;
Var TableError : 0..2;

Procedure InitTable(var T:Table; SizeMem, SizeEl:Word);
Function EmptyTable(var T:Table):boolean; {Возвращает TRUE, если таб-
лица пуста, иначе — FALSE}
Function PutTable(var T:Table; El:Element; var F: Func):boolean; {Вклю-
чение элемента в таблицу. Возвращает TRUE, если элемент включен в таб-
лицу, иначе — FALSE (если в таблице уже есть элемент с заданным ключом
или нехватает памяти).}
Function GetTable(var T:Table; var El:Element; Key:T_Key;
      var F: Func):boolean; {Исключение элемента. Возвра-
щает TRUE, если элемент с ключом Key был в таблице, иначе — FALSE}
Function ReadTable(var T:Table; var El:Element; Key:T_Key;
      var F: Func):boolean; {Чтение элемента. Возвращает
TRUE, если элемент с ключом Key есть в таблице, иначе — FALSE}
Function WriteTable(var T:Table; var El:Element; Key:T_Key;
      var F: Func):boolean; {Изменение элемента. Возвра-
щает TRUE, если элемент с ключом Key есть в таблице, иначе — FALSE}
Procedure DoneTable(var T:Table); {Уничтожение таблицы}

```

Реализация на языке C:

```

#if !defined(__TABLE6_H)
#define __TABLE6_H
#include "tree1.h" // Смотреть лаб.раб. №5
const TableOk = 0;
const TableNotMem = 1;
const TableUnder = 2;
typedef List Table;
typedef ... T_Key; // Определить тип ключа
int TableError; // 0..2
void InitTable(Table *T);
int EmptyTable(Table *T); // Возвращает 1, если таблица пуста, иначе — 0
int PutTable(Table *T, BaseType E); /* Включение элемента в таблицу.
Возвращает 1, если элемент включен в таблицу, иначе — 0 (если в таблице
уже есть элемент с заданным ключом или нехватает памяти) */
int GetTable(Table *T, BaseType *E, T_Key Key); /* Исключение элемента.
Возвращает 1, если элемент с ключом Key был в таблице, иначе — 0 */
int ReadTable(Table *T, BaseType *E, T_Key Key); /* Чтение элемента.
Возвращает 1, если элемент с ключом Key есть в таблице, иначе — 0 */
int WriteTable(Table *T, BaseType *E, T_Key Key); /* Изменение элемента.
Возвращает 1, если элемент с ключом Key есть в таблице, иначе — 0 */
void DoneTable(Table *T); // Уничтожение таблицы
#endif

```

7. Упорядоченная таблица на бинарном дереве.**Реализация на языке Pascal:**

```

Unit Table7;
  Interface
    uses Tree6; { см лаб.раб. №7 }
    Const TableOk      = 0;
           TableNotMem = 1;
           TableUnder  = 2;
    Type Table=Tree;
    Func=function(var A,B: pointer):shortint; {Сравнивает ключи элементов
таблицы, адреса которых находятся в параметрах A и B. Возвращает -1, ес-
ли ключ элемента по адресу A меньше ключа элемента по адресу B, 0 — ес-
ли ключи равны и +1 — если ключ элемента по адресу A больше ключа
элемента по адресу B.}
    Var TableError : 0..2;

```

Procedure InitTable(var *T:Table*; *SizeMem*, *SizeEl:Word*);

Function EmptyTable(var *T:Table*):*boolean*; {Возвращает TRUE, если таблица пуста, иначе — FALSE}

Function PutTable(var *T:Table*; *El:Element*; var *F: Func*):*boolean*; {Включение элемента в таблицу. Возвращает TRUE, если элемент включен в таблицу, иначе — FALSE (если в таблице уже есть элемент с заданным ключом или нехватает памяти).}

Function GetTable(var *T:Table*; var *El:Element*; *Key:T_Key*; var *F: Func*):*boolean*; {Исключение элемента. Возвращает TRUE, если элемент с ключом *Key* был в таблице, иначе — FALSE}

Function ReadTable(var *T:Table*; var *El:Element*; *Key:T_Key*; var *F: Func*):*boolean*; {Чтение элемента. Возвращает TRUE, если элемент с ключом *Key* есть в таблице, иначе — FALSE}

Function WriteTable(var *T:Table*; var *El:Element*; *Key:T_Key*; var *F: Func*):*boolean*; {Изменение элемента. Возвращает TRUE, если элемент с ключом *Key* есть в таблице, иначе — FALSE}

Procedure DoneTable(var *T:Table*); {Уничтожение таблицы}

Реализация на языке C:

```
#if !defined(__TABLE7_H)
#define __TABLE7_H
#include "tree6.h" // Смотреть лаб.раб. №5
const TableOk = 0;
const TableNotMem = 1;
const TableUnder = 2;
typedef List Table;
typedef ... T_Key; // Определить тип ключа
typedef int (*func)(void*, void*); /* Сравнивает ключи элементов таблицы,
адреса которых находятся в параметрах a и b. Возвращает -1, если ключ
элемента по адресу a меньше ключа элемента по адресу b, 0 — если ключи
равны и +1 — если ключ элемента по адресу a больше ключа элемента по
адресу b */
int TableError; // 0..2
void InitTable(Table *T, unsigned SizeMem, unsigned SizeEl);
inline int EmptyTable(Table *T); /* Возвращает 1, если таблица пуста,
иначе — 0 */
int PutTable(Table *T, void *E, func f); /* Включение элемента в таблицу.
Возвращает 1, если элемент включен в таблицу, иначе — 0 (если в таблице
уже есть элемент с заданным ключом или нехватает памяти) */
int GetTable(Table *T, void *E, T_Key Key, func f); /* Исключение элемен-
та. Возвращает 1, если элемент с ключом Key был в таблице, иначе — 0 */
```

```

int ReadTable(Table *T, void *E, T_Key Key, func f); /* Чтение элемента.
Возвращает 1, если элемент с ключом Key есть в таблице, иначе — 0 */
int WriteTable(Table *T, void *E, T_Key Key, func f); /* Изменение элемен-
та. Возвращает 1, если элемент с ключом Key есть в таблице, иначе — 0 */
void DoneTable(Table *T); //Удаление таблицы из динамической памяти
#endif

```

8. Хеш-таблица. Метод разрешения коллизий — двойное хеширование.

Реализация на языке *Pascal*:

Unit Table8;

Interface

Const TableOk = 0;

TableNotMem = 1;

TableUnder = 2;

Type ElTable=record

flag:-1..1; {flag = -1 — элемент массива был занят}

{flag = 0 — элемент массива свободен}

{flag = 1 — элемент массива занят}

E:pointer;

end;

Index = 0..65520 div sizeof(ElTable);

Tbuf = array [index] of ElTable;

Tabl=record

Buf: ^Tbuf;

n: word; {количество элементов в таблице}

SizeBuf: word; {количество элементов в массиве}

SizeEl: word; {размер элемента таблицы}

end;

Func=function(var A,B: pointer):shortint; {Сравнивает ключи элементов таблицы, адреса которых находятся в параметрах A и B. Возвращает -1, если ключ элемента по адресу A меньше ключа элемента по адресу B, 0 — если ключи равны и +1 — если ключ элемента по адресу A больше ключа элемента по адресу B.}

Var TableError : 0..2;

Procedure InitTable(var T:Table; SizeBuf,SizeEl:Word);

Function EmptyTable(var T:Table):boolean; {Возвращает TRUE, если таблица пуста, иначе — FALSE}

Function PutTable(var T:Table; El:Element; var F: Func):boolean; {Включение элемента в таблицу. Возвращает TRUE, если элемент включен в таб-

лицу, иначе — *FALSE* (если в таблице уже есть элемент с заданным ключом или нехватает памяти).}

```
Function GetTable(var T:Table; var El:Element; Key:T_Key;
    var F: Func):boolean; {Исключение элемента. Возвращает TRUE, если элемент с ключом Key был в таблице, иначе — FALSE}
Function ReadTable(var T:Table; var El:Element; Key:T_Key;
    var F: Func):boolean; {Чтение элемента. Возвращает TRUE, если элемент с ключом Key есть в таблице, иначе — FALSE}
Function WriteTable(var T:Table; var El:Element; Key:T_Key;
    var F: Func):boolean; {Изменение элемента. Возвращает TRUE, если элемент с ключом Key есть в таблице, иначе — FALSE}
Procedure DoneTable(var T:Table); {Уничтожение таблицы}
```

Реализация на языке C:

```
#if !defined(__TABLE8_H)
#define __TABLE8_H
const TableOk = 0;
const TableNotMem = 1;
const TableUnder = 2;
typedef... T_Key; // Определить тип ключа
typedef struct ElTable
{
    int flag; /* flag = -1 — элемент массива был занят
               flag = 0 — элемент массива свободен
               flag = 1 — элемент массива занят */
    void* E;
};
typedef struct Table
{
    ElTable* Buf;
    unsigned n; // Количество элементов в таблице
    unsigned SizeBuf; // Количество элементов в массиве
    unsigned SizeEl; // Размер элемента таблицы
};
typedef int (*func)(void*, void*); /* Сравнивает ключи элементов таблицы,
адреса которых находятся в параметрах a и b. Возвращает -1, если ключ
элемента по адресу a меньше ключа элемента по адресу b, 0 — если ключи
равны и +1 — если ключ элемента по адресу a больше ключа элемента по
адресу b */
int TableError; // 0..2
```

```

void InitTable(Table *T, unsigned SizeBuf, unsigned SizeEl);
int EmptyTable(Table *T); // Возвращает 1 , если таблица пуста, иначе — 0
int PutTable(Table *T, void *E, func f); /* Включение элемента в таблицу.
Возвращает 1 , если элемент включен в таблицу, иначе — 0 (если в таблице
уже есть элемент с заданным ключом или нехватает памяти) */
int GetTable(Table *T, void *E, T_Key Key, func f); /* Исключение элемен-
та. Возвращает 1 , если элемент с ключом Key был в таблице, иначе — 0 */
int ReadTable(Table *T, void *E, T_Key Key, func f); /* Чтение элемента.
Возвращает 1 , если элемент с ключом Key есть в таблице, иначе — 0 */
int WriteTable(Table *T, void *E, T_Key Key, func f); /* Изменение элемен-
та. Возвращает 1 , если элемент с ключом Key есть в таблице, иначе — 0 */
void DoneTable(Table *T); // Уничтожение таблицы
#endif

```

9. Хеш-таблица. Метод разрешения коллизий — цепочки переполнения.

Реализация на языке *Pascal*:

Unit Table9;

Interface

uses list3; {см лаб.раб. №5}

Const TableOk = 0;

TableNotMem = 1;

TableUnder = 2;

Type Index = 0..65520 div sizeof(List);

Tbuf = array [index] of List;

Tabl=record

Buf: ^Tbuf;

n: word; {количество элементов в таблице}

SizeBuf: word; {количество элементов в массиве}

SizeEl: word; {размер элемента таблицы}

end;

*Func=function(var A,B: pointer):shortint; {Сравнивает ключи элемен-
тов таблицы, адреса которых находятся в параметрах A и B. Возвращает -1,
если ключ элемента по адресу A меньше ключа элемента по адресу B, 0 —
если ключи равны и +1 — если ключ элемента по адресу A больше ключа
элемента по адресу B.}*

Var TableError : 0..2;

Procedure InitTable(var T:Table; SizeBuf,SizeEl:Word);

*Function EmptyTable(var T:Table):boolean; {Возвращает TRUE , если таб-
лица пуста, иначе — FALSE}*

Function PutTable(var T:Table; El:Element; var F: Func):boolean; {Включение элемента в таблицу. Возвращает *TRUE* , если элемент включен в таблицу, иначе — *FALSE* (если в таблице уже есть элемент с заданным ключом или нехватает памяти).}

Function GetTable(var T:Table; var El:Element; Key:T_Key; var F: Func):boolean; {Исключение элемента. Возвращает *TRUE* , если элемент с ключом *Key* был в таблице, иначе — *FALSE*}

Function ReadTable(var T:Table; var El:Element; Key:T_Key; var F: Func):boolean; {Чтение элемента. Возвращает *TRUE* , если элемент с ключом *Key* есть в таблице, иначе — *FALSE*}

Function WriteTable(var T:Table; var El:Element; Key:T_Key; var F: Func):boolean; {Изменение элемента. Возвращает *TRUE* — если элемент с ключом *Key* есть в таблице, иначе — *FALSE*}

Procedure DoneTable(var T:Table); {Уничтожение таблицы}

Реализация на языке C:

```
#if !defined(__TABLE9_H)
#define __TABLE9_H
#include "list3.h" // Смотреть лаб.раб. №5
const TableOk = 0;
const TableNotMem = 1;
const TableUnder = 2;
typedef ... T_Key; // Определить тип ключа
typedef struct Table
{
    List* Buf;
    unsigned n; // Количество элементов в таблице
    unsigned SizeBuf; // Количество элементов в массиве
    unsigned SizeEl; // Размер элемента таблицы
};
typedef int (*func)(void*, void*); /* Сравнивает ключи элементов таблицы,
адреса которых находятся в параметрах a и b. Возвращает -1, если ключ
элемента по адресу a меньше ключа элемента по адресу b, 0 — если ключи
равны и +1 — если ключ элемента по адресу a больше ключа элемента по
адресу b */
int TableError; // 0..2
void InitTable(Table *T, unsigned SizeBuf, unsigned SizeEl);
int EmptyTable(Table *T); // Возвращает 1 — если таблица пуста, иначе 0
int PutTable(Table *T, void *E, func f); /* Включение элемента в таблицу.
Возвращает 1 , если элемент включен в таблицу, иначе — 0 (если в таблице
уже есть элемент с заданным ключом или нехватает памяти) */
```

```

    int GetTable(Table *T, void *E, T_Key Key, func f); /* Исключение элемен-
та. Возвращает 1, если элемент с ключом Key был в таблице, иначе — 0 */
    int ReadTable(Table *T, void *E, T_Key Key, func f); /* Чтение элемента.
Возвращает 1, если элемент с ключом Key есть в таблице, иначе — 0 */
    int WriteTable(Table *T, void *E, T_Key Key, func f); /* Изменение элемен-
та. Возвращает 1, если элемент с ключом Key есть в таблице, иначе — 0 */
    void DoneTable(Table *T); // Уничтожение таблицы
#endif

```

10. Хеш-таблица. Метод разрешения коллизий — цепочки переполнения.

Реализация на языке *Pascal*:

Unit Table10;

Interface

uses list5; { см лаб. раб. №5 }

Const TableOk = 0;

TableNotMem = 1;

TableUnder = 2;

Type Index = 0..65520 div sizeof(List);

Tbuf = array [index] of List;

Tabl = record

Buf: ^Tbuf;

n: word; { количество элементов в таблице }

SizeBuf: word; { количество элементов в массиве }

end;

Var TableError : 0..2;

Procedure InitTable(var T:Table; SizeBuf:Word);

Function EmptyTable(var T:Table):boolean; { Возвращает TRUE, если таблица пуста, иначе — FALSE }

Function PutTable(var T:Table; El:Element):boolean; { Включение элемента в таблицу. Возвращает TRUE, если элемент включен в таблицу, иначе — FALSE (если в таблице уже есть элемент с заданным ключом или не хватает памяти). }

Function GetTable(var T:Table; var El:Element; Key:T_Key):boolean; { Исключение элемента. Возвращает TRUE, если элемент с ключом Key был в таблице, иначе — FALSE }

Function ReadTable(var T:Table; var El:Element; Key:T_Key):boolean; { Чтение элемента. Возвращает TRUE, если элемент с ключом Key есть в таблице, иначе — FALSE }

Function WriteTable(var T:Table; var El:Element; Key:T_Key):boolean;
 {Изменение элемента. Возвращает *TRUE* , если элемент с ключом *Key* есть
 в таблице, иначе — *FALSE*}

Procedure DoneTable(var T:Table); {Уничтожение таблицы}

Реализация на языке C:

```
#if !defined(__TABLE10_H)
#define __TABLE10_H
#include "list5.h" // Смотреть лаб.раб. №5
const TableOk = 0;
const TableNotMem = 1;
const TableUnder = 2;
typedef ... T_Key; // Определить тип ключа
typedef struct Table
{
    List* Buf;
    unsigned n; // Количество элементов в таблице
    unsigned SizeBuf; // Количество элементов в массиве Buf
};
int TableError; // 0..2
void InitTable(Table *T, unsigned SizeBuf);
int EmptyTable(Table *T); // Возвращает 1 , если таблица пуста, иначе — 0
int PutTable(Table *T, BaseType E); /* Включение элемента в таблицу.
Возвращает 1 , если элемент включен в таблицу, иначе — 0 (если в таблице
уже есть элемент с заданным ключом или нехватает памяти) */
int GetTable(Table *T, BaseType *E, T_Key Key); /* Исключение элемента.
Возвращает 1 , если элемент с ключом Key был в таблице, иначе — 0 */
int ReadTable(Table *T, BaseType *E, T_Key Key); /* Чтение элемента.
Возвращает 1 , если элемент с ключом Key есть в таблице, иначе — 0 */
int WriteTable(Table *T, BaseType *E, T_Key Key); /* Изменение элемента.
Возвращает 1 , если элемент с ключом Key есть в таблице, иначе — 0 */
void DoneTable(Table *T); // Уничтожение таблицы
#endif
```

11. Хеш-таблица. Метод разрешения коллизий – дерево переполнения.

Реализация на языке Pascal:

```
Unit Table11;

Interface

uses Tree6; {см лаб.раб. №7}
Const TableOk      = 0;
```

```

TableNotMem    = 1;
TableUnder     = 2;
Type Index = 0..65520 div sizeof(Tree);
Tbuf = array [index] of List;
Tabl=record
    Buf: ^Tbuf;
    n: word; { количество элементов в таблице }
    SizeBuf: word; { количество элементов в массиве }
    SizeEl: word; { размер элемента таблицы }
end;

```

Func=function(var A,B: pointer):shortint; {Сравнивает ключи элементов таблицы, адреса которых находятся в параметрах A и B. Возвращает -1, если ключ элемента по адресу A меньше ключа элемента по адресу B, 0 — если ключи равны и +1 — если ключ элемента по адресу A больше ключа элемента по адресу B.}

```
Var TableError : 0..2;
```

```
Procedure InitTable(var T:Table; SizeBuf,SizeEl:Word);
```

Function EmptyTable(var T:Table):boolean; {Возвращает TRUE , если таблица пуста, иначе — FALSE}

Function PutTable(var T:Table; El:Element; var F: Func):boolean; {Включение элемента в таблицу. Возвращает TRUE , если элемент включен в таблицу, иначе — FALSE (если в таблице уже есть элемент с заданным ключом или нехватает памяти).}

```
Function GetTable(var T:Table; var El:Element; Key:T_Key;
```

var F: Func):boolean; {Исключение элемента. Возвращает TRUE , если элемент с ключом Key был в таблице, иначе — FALSE}

```
Function ReadTable(var T:Table; var El:Element; Key:T_Key;
```

var F: Func):boolean; {Чтение элемента. Возвращает TRUE , если элемент с ключом Key есть в таблице, иначе — FALSE}

```
Function WriteTable(var T:Table; var El:Element; Key:T_Key;
```

var F: Func):boolean; {Изменение элемента. Возвращает TRUE , если элемент с ключом Key есть в таблице, иначе — FALSE}

```
Procedure DoneTable(var T:Table); {Уничтожение таблицы}
```

Реализация на языке C:

```

#ifndef __TABLE11_H
#define __TABLE11_H
#include "tree6.h" // Смотреть лаб.раб. №7
const TableOk = 0;
const TableNotMem = 1;
const TableUnder = 2;

```

```

typedef ... T_Key; // Определить тип ключа
typedef struct Table
{
    Tree* Buf;
    unsigned n; // Количество элементов в таблице
    unsigned SizeBuf; // Количество элементов в массиве Buf
    unsigned SizeEl; // Размер элемента таблицы
};

typedef int (*func)(void*, void*); /* Сравнивает ключи элементов таблицы,
адреса которых находятся в параметрах a и b. Возвращает -1, если ключ
элемента по адресу a меньше ключа элемента по адресу b, 0 — если ключи
равны и +1 — если ключ элемента по адресу a больше ключа элемента по
адресу b */
int TableError; // 0..2
void InitTable(Table *T, unsigned SizeBuf, unsigned SizeEl);
int EmptyTable(Table *T); // Возвращает 1, если таблица пуста, иначе — 0
int PutTable(Table *T, void *E, func f); /* Включение элемента в таблицу.
Возвращает 1, если элемент включен в таблицу, иначе — 0 (если в таблице
уже есть элемент с заданным ключом или нехватает памяти) */
int GetTable(Table *T, void *E, T_Key Key, func f); /* Исключение элемен-
та. Возвращает 1, если элемент с ключом Key был в таблице, иначе — 0 */
int ReadTable(Table *T, void *E, T_Key Key, func f); /* Чтение элемента.
Возвращает 1, если элемент с ключом Key есть в таблице, иначе — 0 */
int WriteTable(Table *T, void *E, T_Key Key, func f); /* Изменение
элемента. Возвращает 1, если элемент с ключом Key есть в таблице,
иначе — 0 */
void DoneTable(Table *T); // Уничтожение таблицы
#endif

```

Содержание отчета

1. Тема лабораторной работы.
2. Цель работы.
3. Характеристика СД типа «таблица» (п.1 постановки задачи).
4. Индивидуальное задание.
5. Текст модуля для реализации СД типа «таблица», текст программы для отладки модуля, тестовые данные результат работы программы.
6. Текст программы для решения задачи с использованием модуля, тестовые данные, результат работы программы.

Теоретические сведения

Таблица — это набор элементов одинаковой организации, каждый из которых можно представить в виде двойки $\langle K, V \rangle$, где K — ключ, а V — тело (информационная часть) элемента. Ключ уникален для каждого элемента, т.е. в таблице нет двух элементов с одинаковыми ключами. Ключ используется для доступа к элементам при выполнении операций.

Таблица — динамическая структура. Над таблицей определены следующие основные операции:

1. Инициализация.
2. Включение элемента.
3. Исключение элемента с заданным ключом.
4. Чтение элемента с заданным ключом.
5. Изменение элемента с заданным ключом.
6. Проверка пустоты таблицы.
7. Уничтожение таблицы.

Кардинальное число СД БД определяется по формуле

$$CAR(Table) = CAR(BaseType)^0 + CAR(BaseType)^1 + \dots + CAR(BaseType)^{max},$$

где $CAR(BaseType)$ — кардинальное число тела элемента таблицы типа $BaseType$, max — количество различных ключей в таблице.

На *абстрактном* уровне таблица представляет собой множество.

На *физическом* уровне таблица реализуется последовательной или связанной схемой хранения. Располагаться таблица может в статической или динамической памяти. В зависимости от способа размещения элементов таблицы классифицируются на неупорядоченные, упорядоченные и хеш-таблицы.

Для хранения элементов неупорядоченной таблицы используется дополнительная структура данных — линейный список (последовательный или односвязный). Элементы списка неупорядочены по значению ключа. Для поиска элемента с заданным ключом применяется алгоритм линейного или быстрого линейного поиска (в случае реализации таблицы на основе последовательного списка). При реализации таблицы на основе ПЛС включать элемент таблицы эффективнее в конец списка. Включить элемент можно только в том случае, если в таблице нет элемента с таким же ключом, поэтому операцию включения можно выполнить только после применения алгоритма поиска. Если элемента с заданным ключом в ОЛС нет, алгоритм поиска заканчивается после обработки последнего элемента ОЛС, поэтому элемент может быть включен в конец ОЛС.

Исключение элемента таблицы, реализованной на основе ОЛС выполняется также, как и исключение элемента ОЛС. Время выполнения опе-

рации исключения элемента в ПЛС зависит от расположения элемента. Для достижения большей эффективности при выполнении операции исключения элемента таблицы, реализованной на основе ПЛС, можно исключать последний элемент ПЛС, предварительно скопировав его на место исключаемого элемента таблицы.

Для хранения элементов упорядоченной таблицы можно использовать дополнительные структуры данных — линейный список (последовательный или односвязный) или бинарное дерево. Если элементы таблицы расположены в линейном списке, то они упорядочиваются по возрастанию значений ключа. Для поиска элемента с заданным ключом применяются алгоритмы линейного, бинарного или блочного (в случае реализации таблицы на основе последовательного списка) поиска. После выполнения операции включения элементы списка должны остаться упорядоченными, поэтому перед выполнением операции включения необходимо найти элемент списка, после которого нужно включить элемент. При реализации таблицы на основе ПЛС включение элемента требует перемещения всех элементов списка, расположенных правее элемента, за которым включается элемент, на одну позицию вправо. Сократить время выполнения операции включения можно путем сочетания перемещения и поиска элемента, за которым включается элемент (аналогично алгоритму сортировки вставками (см лаб.раб. №4)). Включение элемента в ОЛС не требует перемещения элементов, поэтому операцию включения элемента в таблицу можно достаточно эффективно реализовать, выполнив последовательно алгоритмы поиска и включения элемента в ОЛС.

Для хранения элементов упорядоченной таблицы может быть использовано бинарное дерево. БД должно обладать следующим основным свойством: ключ любого элемента, расположенного в левом поддереве должен быть меньше ключа элемента, расположенного в корне, а ключ любого элемента правого поддерева — больше ключа корневого элемента. Включаемый элемент всегда подключается к вершине, у которой хотя бы одно из поддеревьев пусто (алгоритм включения см в лаб.раб. №7).

При исключении элемента основное свойство БД должно сохраняться. Алгоритм исключения элемента зависит от типа вершины, в которой находится исключаемый элемент таблицы. Возможны три типа вершин:

1) вершина не имеет потомков (лист). В этом случае родитель теряет соответствующего сына.

2) вершина имеет одного потомка. В результате исключения такой вершины сыном родителя станет потомок удаляемой вершины.

3) вершина имеет двух потомков. Для исключения вершины нужно найти такую подходящую легко удаляемую вершину (типа 1 или 2), значение которой можно было бы переписать в исключаемую вершину без

нарушения основного свойства. Такая вершина всегда существует: это либо самая правая вершина в левом поддереве, либо самая левая вершина в правом поддереве (объясните, почему).

Алгоритм исключения:

1. Поиск подходящей легко удаляемой вершины.
2. Запись значения найденной вершины в исключаемую.
3. Исключение из БД найденной вершины.

Хеш-таблица — это таблица, в которой положение a_i (адрес) элемента в памяти определяется с помощью некоторой функции H (хеш-функции), аргументом которой является значение ключа k_j элемента. Функция хеширования определяется как отображение

$H: K \rightarrow A$, где

$K = \{k_1, k_2, \dots, k_m\}$ — множество значений ключа;

$A = \{a_1, a_2, \dots, a_n\}$ — адресное пространство;

$m \leq n$.

Для хранения элементов хеш-таблицы может быть использована дополнительная СД — массив, тогда $a_i \in A$ представляет собой индекс элемента массива. Если определена функция $H(k)$, такая, что для любого k_j , $j = 1, 2, \dots, m$, $H(k_j)$ имеет целочисленное значение из множества A , причем $H(k_i) \neq H(k_j)$, то каждый элемент таблицы с ключом k взаимнооднозначно отображается в элемент массива с индексом $H(k)$. Доступ к записи с ключом k осуществляется в этом случае непосредственно путем вычисления значения $H(k)$. Таблицы, для которых существует и известна такая хеш-функция, называют хеш-таблицами с прямым доступом. Время выполнения операций поиска, включения, исключения, чтения и изменения не зависит от количества элементов в таблице и определяется временем вычисления значения хеш-функции и временем доступа к элементу массива.

Подбор хеш-функции, обеспечивающей взаимную однозначность преобразования значения ключа в индекс элемента массива, в общем случае весьма трудная задача. На практике ее можно решить только для постоянных таблиц с заранее известным набором значений ключа. Такие таблицы широко используются в трансляторах.

Поскольку взаимную однозначность преобразования значения ключа в индекс элемента массива в общем случае обеспечить практически невозможно (при соизмеримых значениях n и m), от требования взаимной однозначности отказываются. Это приводит к тому, что для некоторых различных значений ключа значение хеш-функции может быть одно и то же, т.е. $H(k_i) = H(k_j)$. Это означает, что два элемента таблицы должны быть размещены в одном элементе массива, что невозможно. Такую ситуацию назы-

вают коллизией. Для разрешения коллизий используют различные методы. Наиболее часто используют для этого алгоритмы из двух основных групп, а именно: открытая адресация (методы двойного хеширования и линейного опробования) и метод цепочек.

В методе открытой адресации в качестве дополнительной СД используется массив, элементы которого могут содержать элементы таблицы. Каждый элемент массива имеет признак: содержит ли элемент массива элемент таблицы, содержал ли элемент массива элемент таблицы или элемент массива свободен. Для разрешения коллизий используется две хеш-функции: $H_1(k)$ и $H_2(a)$.

Алгоритм выполнения операций поиска, включения, исключения, чтения и изменения элемента таблицы с ключом k_j :

1. Вычислить $a_i = H_1(k_j)$.
2. Если при включении в таблицу новой записи элемент массива a_i или содержал ранее элемент таблицы, но в настоящее время не содержит, а при исключении содержит элемент таблицы с ключом k_j , то поиск завершен. В противном случае перейти к следующему пункту.
3. Вычислить $a_i = H_2(a_i)$ и перейти к пункту 2.

При включении новых элементов таблицы алгоритм остается конечным до тех пор, пока есть хотя бы один свободный элемент массива. При исключении элемента из таблицы алгоритм конечен, если таблица содержит элемент с заданным ключом. При невыполнении этих условий произойдет заикливание, против которого нужно принимать специальные меры. Например, можно ввести счетчик числа просмотренных позиций, если это число станет больше n , то закончить алгоритм.

Время выполнения операций над хеш-таблицей зависит от числа коллизий и времени вычисления значения хеш-функции, уменьшить которое можно путем использования «хорошей» хеш-функции. В случае целочисленного ключа в качестве хеш-функции часто применяется функция $H_1(k) = (C_1 * k + C_2) \bmod n$, где C_1 и C_2 — константы, взаимно простые числа. Если ключ нецелочисленный, то одним из простых методов получения хеш-функции является выделение части цифрового кода ключа. Например, пусть $m \leq 256$, тогда $H_1(k)$ в качестве значения может иметь целочисленное представление первого или последнего байта ключа или какие-либо восемь разрядов из середины ключа. Для улучшения равномерности распределения значений хеш-функции применяют сложение кода ключа: первая половина кода складывается со второй и из результата выбираются какие-либо восемь разрядов. Можно также разделить код ключа на байты, а затем сложить их по модулю 2^8 .

В качестве функции $H_2(a)$, называемой функцией рехеширования, может быть использована $H_2(a) = (a + C) \bmod n$, где C — целочисленная константа. Если $C = 1$, то метод разрешения коллизий называется методом линейного опробования, иначе — методом двойного хеширования. Недостатком метода линейного опробования является эффект первичного сгущения, который заключается в том, что при возникновении коллизии заполняются соседние элементы массива, увеличивая число проб при выполнении операции поиска. Особенностью хеш-таблиц является то, что время поиска элемента с заданным ключом не зависит от количества элементов в таблице, а зависит только от заполненности массива.

В методе цепочек элемент массива T с индексом a_i содержит цепочку элементов таблицы, ключи которых отображаются хеш-функцией в значение a_i . Для хранения таких цепочек элементов может быть использована дополнительная СД — линейный список (ПЛС или ОЛС), т.о. элемент массива T представляет собой линейный список. Алгоритмы включения и исключения элементов в такую таблицу очень просты.

Алгоритм включения/исключения элемента с ключем k_j в таблицу:

1. Вычислить $a_i = H(k_j)$.
2. Включить/исключить элемент в линейный список $T[a_i]$.

Для ускорения поиска элементов таблицы, ключи которых отображаются хеш-функцией в одно и то же значение a_i , в качестве элемента массива может быть использована СД типа БД.

К о н т р о л ь н ы е в о п р о с ы

1. Что такое таблица? Какие операции определены над таблицами?
2. Как классифицируются таблицы в зависимости от способа размещения их элементов?
3. Определите порядок функции временной сложности операций включения и исключения элементов в неупорядоченные и упорядоченные таблицы.
4. Как исключить элемент из упорядоченной таблицы, реализованной с использованием бинарного дерева?
5. Что такое хеш-таблица, хеш-функция, коллизия?
6. Какие существуют методы разрешения коллизий?
7. При каком методе разрешения коллизий возможно заикливание и как его избежать?
8. Определите порядок функции временной сложности алгоритмов выполнения операций над хеш-таблицами.

Библиографический список

1. *Кнут, Д.* Искусство программирования. Том 1. Основные алгоритмы: Пер. с англ. / Д. Кнут. – 3-е изд. – М.: Изд. дом «Вильямс», 2000. – 830 с.
2. *Кнут, Д.* Искусство программирования. Том 3. Сортировка и поиск: Пер. с англ. / Д. Кнут. – 3-е изд. – М.: Изд. дом «Вильямс», 2000. – 843 с.
3. *Вирт, Н.* Алгоритмы + структуры данных = программы / Н. Вирт – М.: Мир, 1985. – 360 с.
4. *Ахо, А.* Структуры данных и алгоритмы / А. Ахо, Д. Хонкрофт, Д. Ульман – М.: Изд. дом «Вильямс», 2001. – 536 с.
5. *Кармен, Т.* Алгоритмы: построение и анализ / Кармен Т., Лейзерсон Ч., Ривест Р. – 2-е изд. – М.: МЦНМО, 2009. – 350 с.
6. *Хусаинов, Б.С.* Структуры и алгоритмы обработки данных. Примеры на языке С / Б.С. Хусаинов – М.: Финансы и статистика, 2004. – 463 с.