

Министерство сельского хозяйства РФ  
ФГБОУ ВПО «Кубанский государственный  
аграрный университет»

**В. И. Лойко, С. В. Лаптев**

# **СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ**

**Учебное пособие**

**2-е издание переработанное и дополненное**

**Рекомендовано  
Министерством сельского хозяйства  
Российской Федерации  
в качестве учебного пособия  
для студентов высших учебных заведений  
по агроэкономическим специальностям**

**Краснодар  
2013**

**УДК 004.021 (075.8)**

**ББК 73**

**Л73**

**Рецензенты:**

проф. д-р экон. наук **Т. П. Барановская**  
(зав. каф. СА и ОИ, КубГАУ)

проф. д-р техн. наук **Г. А. Аршинов**  
(проф. кафедры КТС, КубГАУ)

**Лойко В. И.**

**Л 73** Структуры и алгоритмы обработки данных: учеб. пособие для вузов / В .И. Лойко, С. В. Лаптев. – 2-е изд., перераб. и доп. – Краснодар: КубГАУ, 2013. - 345 с.

**ISBN 978-5-94672-679-5**

Учебное пособие разработано на основе лекций по курсу «Структуры и алгоритмы обработки данных», преподаваемых авторами студентам различных специальностей. В теоретической части пособия изложены основные положения теории алгоритмов и структур данных для персональных ЭВМ. Главное внимание в пособии уделено оперативным структурам.

В практической части пособия приведены указания к лабораторным работам и курсовому проектированию.

Учебное пособие предназначено для студентов всех специальностей факультета «Прикладная информатика» и других специальностей, изучающих информатику и информационные технологии.

**УДК 004.021 (075.8)**

**ББК 73**

© Лойко В.И., Лаптев С.В., 2013

© ФГБОУ ВПО «Кубанский государственный аграрный университет», 2013

**ISBN 978-5-94672-679-5**

## СОДЕРЖАНИЕ

<b>ВВЕДЕНИЕ .....</b>	<b>9</b>
<b>ЧАСТЬ 1. ВВЕДЕНИЕ В ТЕОРИЮ СТРУКТУР ДАННЫХ И АЛГОРИТМОВ ИХ ОБРАБОТКИ .</b>	<b>12</b>
<b>1.ТИПЫ ДАННЫХ .....</b>	<b>13</b>
1.1 ЦЕЛЫЙ ТИП - INTEGER.....	14
1.2 ВЕЩЕСТВЕННЫЙ ТИП - REAL .....	15
1.3 ЛОГИЧЕСКИЙ ТИП - BOOLEAN .....	16
1.4 СИМВОЛЬНЫЙ ТИП - CHAR .....	16
1.5 УКАЗАТЕЛЬНЫЙ ТИП - POINTER .....	17
<b>2. СТАТИЧЕСКИЕ И ПОЛУСТАТИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ .....</b>	<b>19</b>
2.1 УРОВНИ ПРЕДСТАВЛЕНИЯ ДАННЫХ .....	20
2.2 КЛАССИФИКАЦИЯ СТРУКТУР ДАННЫХ .....	21
2.3 СТАТИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ.....	22
2.3.1 <i>Векторы</i> .....	22
2.3.2 <i>Массивы</i> .....	23
2.3.3 <i>Записи</i> .....	23
2.3.4 <i>Таблицы</i> .....	26
2.4 ПОЛУСТАТИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ.....	27
2.4.1 <i>Стеки</i> .....	28
2.4.2 <i>Очередь</i> .....	31
2.4.3 <i>Дек</i> .....	43
<b>3. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ</b>	<b>46</b>
3.1 СВЯЗНЫЕ СПИСКИ.....	47
3.1.1 <i>Односвязные списки</i> .....	47
3.1.2 <i>Кольцевой односвязный список</i> .....	48
3.1.3 <i>Двусвязный список</i> .....	50

3.1.4 Кольцевой двусвязный список.....	51
3.2 РЕАЛИЗАЦИЯ СТЕКОВ С ПОМОЩЬЮ ОДНОСВЯЗНЫХ СПИСКОВ.....	51
3.3 ОРГАНИЗАЦИЯ ОПЕРАЦИЙ GETNODE, FREENODE И УТИЛИЗАЦИЯ ОСВОБОДИВШИХСЯ ЭЛЕМЕНТОВ.....	54
3.3.1 Операция $P=GetNode$ .....	55
3.3.2 Операция $FreeNode(P)$ .....	55
3.3.3 Утилизация освобожденных элементов в многосвязных списках .....	56
3.4 ОДНОСВЯЗНЫЙ СПИСОК, КАК САМОСТОЯТЕЛЬНАЯ СТРУКТУРА ДАННЫХ .....	56
3.4.1 Вставка и извлечение элементов из списка.....	58
3.4.2 Примеры типичных операций над списками .....	60
3.4.3 Элементы заголовков в списках .....	62
3.5 НЕЛИНЕЙНЫЕ СВЯЗАННЫЕ СТРУКТУРЫ .....	63
<b>4. РЕКУРСИВНЫЕ СТРУКТУРЫ ДАННЫХ.....</b>	<b>67</b>
4.1 ДЕРЕВЬЯ .....	67
4.1.1 Представление деревьев .....	69
4.2 БИНАРНЫЕ ДЕРЕВЬЯ.....	69
4.2.1 Сведение $m$ -арного дерева к бинарному.....	71
4.2.2 Основные операции с деревьями .....	73
4.2.3 Создание дерева бинарного поиска .....	75
4.2.4 Прохождение (обход) бинарных деревьев .....	76
<b>5. ПОИСК .....</b>	<b>80</b>
5.1 ПОСЛЕДОВАТЕЛЬНЫЙ ПОИСК .....	81
5.2 ИНДЕКСНО-ПОСЛЕДОВАТЕЛЬНЫЙ ПОИСК.....	84
5.3 ЭФФЕКТИВНОСТЬ ПОСЛЕДОВАТЕЛЬНОГО ПОИСКА .....	86
5.4 ЭФФЕКТИВНОСТЬ ИНДЕКСНО-ПОСЛЕДОВАТЕЛЬНОГО ПОИСКА .....	86
5.5 МЕТОДЫ ОПТИМИЗАЦИИ ПОИСКА .....	87
5.5.1 Переупорядочивание таблицы поиска путем перестановки найденного элемента в начало списка ...	88

5.5.2 Метод транспозиции .....	90
5.5.3 Дерево оптимального поиска.....	91
5.6 Бинарный поиск (метод деления пополам).....	93
5.7 Поиск по бинарному дереву .....	95
5.8 Поиск со вставкой (с включением) .....	97
5.9 Поиск с удалением .....	99
<b>6. СОРТИРОВКА .....</b>	<b>103</b>
6.1 Сортировка методом прямого включения .....	104
6.2 Сортировка методом прямого выбора.....	107
6.3 Сортировка с помощью прямого обмена (пузырьковая сортировка).....	108
6.4 Улучшенные методы сортировки.....	111
6.4.1 Быстрая сортировка ( <i>Quick Sort</i> ).....	111
6.4.2 Сортировка Шелла (сортировка с уменьшающимся шагом).....	113
<b>7. ПРЕОБРАЗОВАНИЕ КЛЮЧЕЙ (РАССТАНОВКА).....</b>	<b>116</b>
7.1. Выбор функции преобразования.....	116
7.2. Алгоритм.....	118
<b>ЧАСТЬ 2. ПРАКТИКУМ ПО СТРУКТУРАМ И АЛГОРИТМАМ ОБРАБОТКИ ДАННЫХ В ЭВМ</b>	<b>122</b>
<b>МЕТОДИЧЕСКОЕ РУКОВОДСТВО К ЛАБОРАТОРНЫМ РАБОТАМ .....</b>	<b>123</b>
Организационно-методические указания.....	123
<b>ПОЛУСТАТИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ</b>	<b>125</b>
Лабораторная работа №1 (4 часа). Полустатические структуры данных.....	125
Краткая теория .....	125
Алгоритмы.....	127
Примеры алгоритмов конкретных задач (стеки и операции над ними).....	130

Задания.....	146
<b>ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ ...</b>	<b>147</b>
ЛАБОРАТОРНАЯ РАБОТА №2 (4 часа). СПИСКОВЫЕ СТРУКТУРЫ ДАННЫХ.....	147
КРАТКАЯ ТЕОРИЯ .....	148
АЛГОРИТМЫ.....	149
<i>Линейные однонаправленные списки (односвязные списки)</i> .....	<i>149</i>
<i>Кольцевые списки</i> .....	<i>153</i>
ПРИМЕРЫ АЛГОРИТМОВ КОНКРЕТНЫХ ЗАДАЧ .....	157
Задания.....	161
<b>РЕКУРСИВНЫЕ СТРУКТУРЫ ДАННЫХ.....</b>	<b>164</b>
ЛАБОРАТОРНАЯ РАБОТА №3 (4 часа). БИНАРНЫЕ ДЕРЕВЬЯ (СОЗДАНИЕ И ОБХОД) .....	164
КРАТКАЯ ТЕОРИЯ .....	164
АЛГОРИТМЫ.....	167
ПРИМЕРЫ АЛГОРИТМОВ КОНКРЕТНЫХ ЗАДАЧ .....	172
Задания.....	175
<b>ПОИСК .....</b>	<b>178</b>
ЛАБОРАТОРНАЯ РАБОТА №4 (4 часа). ИССЛЕДОВАНИЕ МЕТОДОВ ЛИНЕЙНОГО И БИНАРНОГО ПОИСКА.....	178
КРАТКАЯ ТЕОРИЯ .....	178
АЛГОРИТМЫ.....	179
ПРИМЕРЫ АЛГОРИТМОВ КОНКРЕТНЫХ ЗАДАЧ .....	187
Задания.....	197
ЛАБОРАТОРНАЯ РАБОТА №5 (4 часа). ИССЛЕДОВАНИЕ МЕТОДОВ ОПТИМИЗАЦИИ ПОИСКА .....	199
КРАТКАЯ ТЕОРИЯ .....	199
АЛГОРИТМЫ.....	201
АЛГОРИТМЫ НА ЯЗЫКЕ C++ .....	205
Задания.....	208
ЛАБОРАТОРНАЯ РАБОТА №6 (4 часа). ПОИСК ПО ДЕРЕВУ С ВКЛЮЧЕНИЕМ И ИСКЛЮЧЕНИЕМ.....	210

КРАТКАЯ ТЕОРИЯ .....	210
АЛГОРИТМЫ.....	211
ПРИМЕРЫ АЛГОРИТМОВ КОНКРЕТНЫХ ЗАДАЧ (ПОИСК ПО ДЕРЕВУ С ВКЛЮЧЕНИЕМ, ИСКЛЮЧЕНИЕМ).....	220
ЗАДАНИЯ.....	230
<b>СОРТИРОВКА .....</b>	<b>232</b>
ЛАБОРАТОРНАЯ РАБОТА №7 (4 ЧАСА). СОРТИРОВКИ МЕТОДАМИ ПРЯМОГО ВКЛЮЧЕНИЯ И ВЫБОРА .....	232
КРАТКАЯ ТЕОРИЯ .....	233
<i>Сортировка методом прямого включения .....</i>	<i>234</i>
<i>Алгоритм сортировки прямым включением .....</i>	<i>235</i>
СОРТИРОВКА МЕТОДОМ ПРЯМОГО ВЫБОРА .....	236
<i>Алгоритм сортировки прямым выбором.....</i>	<i>239</i>
ПРИМЕРЫ АЛГОРИТМОВ И ПРИЛОЖЕНИЙ.....	242
ЗАДАНИЯ.....	247
ЛАБОРАТОРНАЯ РАБОТА №8 (4 ЧАСА). СОРТИРОВКИ МЕТОДОМ ПРЯМОГО ОБМЕНА И С ПОМОЩЬЮ ДЕРЕВА.....	250
КРАТКАЯ ТЕОРИЯ .....	250
АЛГОРИТМЫ.....	251
ПРИМЕРЫ АЛГОРИТМОВ КОНКРЕТНЫХ ЗАДАЧ .....	256
ЗАДАНИЯ.....	260
ЛАБОРАТОРНАЯ РАБОТА №9 (4 ЧАСА). УЛУЧШЕННЫЕ МЕТОДЫ СОРТИРОВКИ .....	263
КРАТКАЯ ТЕОРИЯ .....	263
АЛГОРИТМЫ.....	266
ПРИМЕРЫ АЛГОРИТМОВ КОНКРЕТНЫХ ЗАДАЧ .....	270
ЗАДАНИЯ.....	274
ТЕСТЫ К ЛАБОРАТОРНЫМ РАБОТАМ .....	276
<b>МЕТОДИЧЕСКОЕ РУКОВОДСТВО К КУРСОВОЙ РАБОТЕ .....</b>	<b>290</b>
1 ТРЕБОВАНИЯ К КУРСОВОЙ РАБОТЕ .....	290
2. ПРИМЕРНЫЙ ПЕРЕЧЕНЬ КУРСОВЫХ РАБОТ .....	291
3. ПРИМЕР ВЫПОЛНЕНИЯ КУРСОВОЙ РАБОТЫ.....	292

<i>3.1 Постановка задачи.....</i>	<i>292</i>
<i>3.2 Краткая теория .....</i>	<i>292</i>
<i>3.3 Метод исследования .....</i>	<i>298</i>
<i>3.4 Результаты исследования .....</i>	<i>298</i>
<i>3.5 Контрольный пример .....</i>	<i>300</i>
<i>3.6 Выводы .....</i>	<i>302</i>
<i>3.7 Приложение .....</i>	<i>302</i>
<b>ЗАКЛЮЧЕНИЕ.....</b>	<b>328</b>
<b>ЛИТЕРАТУРА.....</b>	<b>330</b>
<b>ПРИЛОЖЕНИЕ. ТЕСТЫ С ОТВЕТАМИ .....</b>	<b>331</b>



## **ВВЕДЕНИЕ**

Компьютер - это машина, которая обрабатывает информацию. Изучение науки об ЭВМ предполагает изучение того, каким образом эта информация организована внутри ЭВМ, как она обрабатывается и как может быть использована. Следовательно, для изучения предмета студенту особенно важно понять концепции организации информации и работы с ней.

Так как вычислительная техника базируется на изучении информации, то первый возникающий вопрос заключается в том, что такое информация. К сожалению, несмотря на то, что концепция информации является краеугольным камнем всей науки о вычислительной технике, на этот вопрос не может быть дано однозначного ответа. В этом контексте понятие "информация" в вычислительной технике сходно с понятием "точка", "прямая" и "плоскость" в геометрии - все это неопределенные термины, о которых могут быть сделаны некоторые утверждения и выводы, но которые не могут быть объяснены в терминах более элементарных понятий.

Базовой единицей информации является бит, который может принимать два взаимоисключающих значения. Если устройство может находиться более чем в двух состояниях, то тот факт, что оно находится в одном из этих состояний, уже требует нескольких битов информации.

Для представления двух возможных состояний некоторого бита используются двоичные цифры - ноль и единица.

Число битов, необходимых для кодирования символа в конкретной вычислительной машине, называется размером байта, а группа битов в этом числе называется байтом. Размер байта в большинстве ЭВМ равен 8.

Память вычислительной машины представляет собой совокупность битов. В любой момент функционирования в

ЭВМ каждый из битов памяти имеет значение 0 или 1 (сброшен или установлен). Состояние бита называется его значением или содержимым.

Биты в памяти ЭВМ группируются в элементы большего размера, например в байты. В некоторых ЭВМ несколько байтов объединяются в группы, называемые словами. Каждому такому элементу назначается адрес, который представляет собой имя, идентифицирующее конкретный элемент памяти среди аналогичных элементов. Этот адрес обычно числовой. Он называется ячейкой, а содержимое ячейки есть значение битов, которые ее составляют.

Итак, мы видим, что информация в ЭВМ сама по себе не имеет конкретного смысла. С некоторой конкретной битовой комбинацией может быть связано любое смысловое значение. Именно интерпретация битовой комбинации придает ей заданный смысл.

Метод интерпретации битовой информации часто называется типом данных. Каждая ЭВМ имеет свой набор типов данных.

Важно осознавать роль, выполняемую спецификацией типа в языках высокого уровня. Именно посредством подобных объявлений программист указывает на то, каким образом содержимое памяти ЭВМ интерпретируется программой. Эти объявления детерминируют объем памяти, необходимый для размещения отдельных элементов, способ интерпретации этих элементов и другие важные детали. Объявления также сообщают интерпретатору точное значение используемых символов операций.

Под компьютерной программой понимается алгоритм обработки данных, представленный в понятной исполнителю, т.е. ЦЭВМ, форме. По существу объявления сообщают интерпретатору точное значение используемых символов машинных операций.

В учебном пособии рассмотрены алгоритмы реализации и обработки основных структур данных в оперативной памя-

ти ЭВМ. В первой части пособия алгоритмы представлены в псевдокоде, а во второй – на языке программирования C++. Представлены примеры решения конкретных задач с использованием алгоритмов обработки базовых оперативных структур.

**ЧАСТЬ 1.**  
**ВВЕДЕНИЕ В ТЕОРИЮ СТРУКТУР**  
**ДАнных И АЛГОРИТМОВ ИХ ОБ-**  
**РАБОТКИ**

## 1.ТИПЫ ДАННЫХ

В математике принято классифицировать переменные в соответствии с некоторыми важными характеристиками. Мы различаем вещественные, комплексные и логические переменные, переменные, представляющие собой отдельные значения, множества значений или множества множеств. В обработке данных понятие классификации играет такую же, если не большую роль. Мы будем придерживаться того принципа, что любая константа, переменная, выражение или функция относятся к некоторому типу.

Фактически тип характеризует множество значений, которые может принимать некоторая переменная или выражение и которые может формировать функция.

В большинстве языков программирования различают стандартные типы данных и типы, заданные пользователем. Также в большинстве языков программирования в стандартных типах присутствуют следующие:

- a) целый (INTEGER);*
- b) вещественный (REAL) ;*
- c) логический (BOOLEAN);*
- d) символьный (CHAR);*
- e) указательный (POINTER).*

Пользовательские типы могут варьироваться в зависимости от языка программирования и предоставляемых программисту возможностей решений конкретных задач.

Любой тип данных должен быть охарактеризован областью значений и допустимыми операциями над этим типом данных.

## 1.1 Целый тип - INTEGER

Этот тип включает некоторое подмножество целых, размер которого варьируется от машины к машине. Если для представления целых чисел в машине используется  $n$  разрядов, причем используется дополнительный код, то допустимые числа должны удовлетворять условию  $-2^{n-1} \leq x < 2^{n-1}$ .

Считается, что все операции над данными этого типа выполняются точно и соответствуют обычным правилам арифметики. Если результат выходит за пределы представимого множества, то вычисления будут прерваны. Такое событие называется переполнением.

Числа делятся на знаковые и беззнаковые. Для каждого из них имеется свой диапазон значений:

а)  $(0..2^n-1)$  для беззнаковых чисел

б)  $(-2^{N-1}.. 2^{N-1}-1)$  для знаковых.

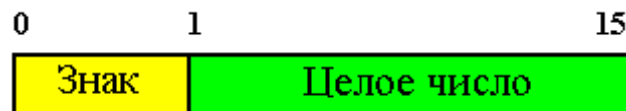


Рис. 1.1

При обработке машиной чисел, используется формат со знаком. Если же машинное слово используется для записи и обработки команд и указателей, то в этом случае используется формат без знака.

Операции над целым типом:

- а) Сложение.
- б) Вычитание.
- с) Умножение.
- д) Целочисленное деление.
- е) Нахождение остатка по модулю.
- ф) Нахождение экстремума числа (минимума и максимума)

g) Реляционные операции (операции сравнения) (<,>,<=,>=,=,<>)

Примеры:

$$A \text{ div } B = C$$

$$A \text{ mod } B = D$$

$$C * B + D = A$$

$$7 \text{ div } 3 = 2$$

$$7 \text{ mod } 3 = 1$$

Во всех операциях, кроме реляционных, в результате получается целое число.

## 1.2 Вещественный тип - REAL

Вещественные типы образуют ряд подмножеств вещественных чисел, которые представлены в машинных форматах с плавающей точкой. Числа в формате с плавающей точкой характеризуются целочисленными значениями мантииссы и порядка, которые определяют диапазон изменения и количество верных знаков в представлении чисел вещественного типа.

$X = +/- M * q^{(+/-P)}$  - полулогарифмическая форма представления числа, показана на рисунке 1.2.

$$937,56 = 93756 * 10^{-2} = 0,93756 * 10^3$$

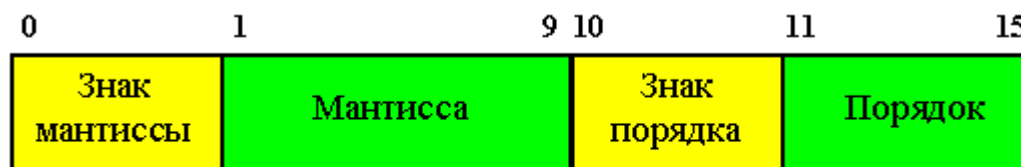


Рис. 1.2

Удвоенная точность необходима для того, чтобы увеличить точность мантииссы.

### 1.3 Логический тип - BOOLEAN

Стандартный логический тип Boolean (размер-1 байт) представляет собой тип данных, любой элемент которого может принимать лишь 2 значения: True и False.

Над логическими элементами данных выполняются логические операции. Основные из них:

- a) Отрицание (NOT)
- b) Конъюнкция (AND)
- c) Дизъюнкция (OR)

**Таблица истинности основных логических функций.**

A	B	not A	A or B	A and B
1	1	0	1	1
1	0	0	1	0
0	1	1	1	0
0	0	1	0	0

Рис. 1.3

Логические значения получаются также при реляционных операциях с целыми числами.

### 1.4 Символьный тип - CHAR

Тип CHAR содержит 26 прописных латинских букв и 26 строчных, 10 арабских цифр и некоторое число других графических символов, например, знаки пунктуации.

Подмножества букв и цифр упорядочены и "соприкасаются", т.е.

$("A" \leq x) \& (x \leq "Z")$  - x представляет собой прописную букву



("0"<= x)&(x <= "9") - x представляет собой цифру

Тип CHAR содержит некоторый непечатаемый символ, пробел, его можно использовать как разделитель.

Операции:

- a) Присваивания
- b) Сравнения
- c) Определения номера данной литеры в системе кодирования. ORD( $W_i$ )
- d) Нахождение литеры по номеру. CHR(i)
- e) Вызов следующей литеры. SUCC( $W_i$ )
- f) Вызов предыдущей литеры. PRED( $W_i$ )

## 1.5 Указательный тип - POINTER

Переменная типа указатель является физическим носителем адреса величины базового типа. Стандартный тип-указатель Pointer дает указатель, не связанный ни с каким конкретным базовым типом. Этот тип совместим с любым другим типом-указателем.

Операции:

- a) Присваивания
- b) Операции с беззнаковыми целыми числами.

При помощи этих операций можно вычислить адрес данных. В машинном виде эти типы занимают максимально возможную длину.

**Например:**

ABCD:1234 - значение указателя в шестнадцатеричной системе счисления - относительный адрес.

Первое число (ABCD) - адрес сегмента

Второе число (1234) - адрес внутри сегмента.

Для получения абсолютного адреса необходимо произвести сдвиг адреса сегмента влево, и к полученному числу прибавить адрес внутреннего сегмента.

**Например:**

- 1) Сдвигаем ABCD на один разряд влево. Получаем ABCD0.
- 2) Прибавляем 1234. Полученный результат и является абсолютным адресом.

ABCD0

12 3 4

-----

ACF04 - абсолютный адрес данного числа.

**Контрольные вопросы:**

1. Какие типы данных вы знаете?
2. Как представляются целые числа?
3. Как представляются вещественные числа?
4. Что представляют собой данные логического типа?
5. Какому условию должны удовлетворять допустимые числа типа INTEGER?
6. Какие операции можно производить над целыми числами?
7. Перечислите булевские операции.
8. Какова структура типа CHAR?
9. Какие операции возможны над данными этого типа?
10. Что можно вычислить с помощью данных указательного типа?

## 2. СТАТИЧЕСКИЕ И ПОЛУСТАТИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Структуры данных - это совокупность элементов данных и отношений между ними. При этом под элементами данных может подразумеваться как простое данное так и структура данных. Под отношениями между данными понимают функциональные связи между ними и указатели на то, где находятся эти данные.

Графическое представление элемента структуры данных.

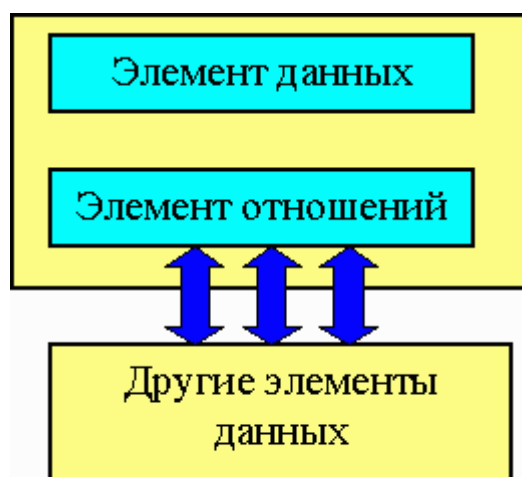


Рис.2.1

Элемент отношений - это совокупность всех связей элемента с другими элементами данных, рассматриваемой структуры.

$$S:=(D,R)$$

Где  $S$  - структура данных,  $D$  - данные и  $R$  - отношения.

Как бы сложна ни была структура данных, в конечном итоге она состоит из простых данных (см. рис. 2.2, 2.3).

### *Одномерный массив*

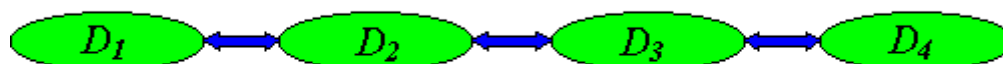


Рис.2.2

### *Двумерный массив*

$\begin{smallmatrix} i \\ j \end{smallmatrix}$	1	2	...	n
1	элемент данных	элемент данных	...	элемент данных
2	элемент данных	элемент данных	...	элемент данных
...	...	...	...	
n	элемент данных	элемент данных	...	элемент данных

Рис.2.3

## 2.1 Уровни представления данных

Память машины состоит из миллионов триггеров, которые обрабатывают поступающую информацию. Мы, занося информацию в компьютер, представляем ее в каком-то виде, который на наш взгляд упорядочивает данные и придает им смысл. Машина отводит поле для поступающей информации и задает ей какой-то адрес. Таким образом, получается, что мы обрабатываем данные на логическом уровне, как бы абстрактно, а машина делает это на физическом уровне.

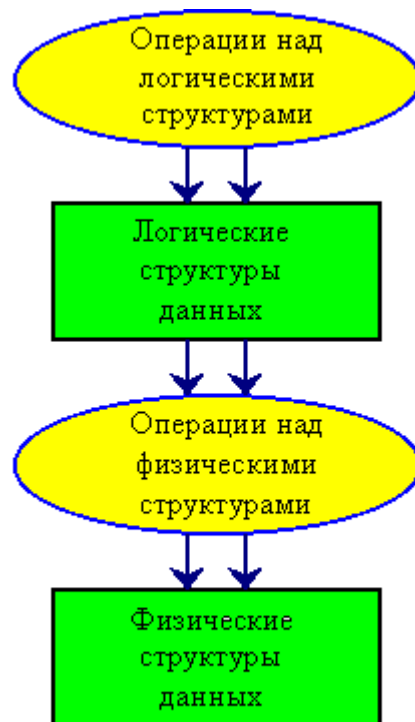


Рис. 2.4

Последовательность переходов от логической организации к физической показана на рис. 2.4.

## 2.2 Классификация структур данных

Структуры данных классифицируются:

1. По связанности данных в структуре:

- если данные в структуре связаны очень слабо, то такие структуры называются несвязанными (вектор, массив, строки, стеки)

- если данные в структуре связаны, то такие структуры называются связанными (связанные списки)

2. По изменчивости структуры во времени или в процессе выполнения программы:

- статические структуры - структуры, неменяющиеся до конца выполнения программы (записи, массивы, строки, вектора)

- полустатические структуры (стеки, деки, очереди)

- динамические структуры - происходит полное изменение при выполнении программы

3. По упорядоченности структуры:

- линейные (вектора, массивы, стеки, деки, записи)
- нелинейные (многосвязные списки, древовидные структуры, графы)

Наиболее важной характеристикой является изменчивость структуры во времени.

## 2.3 Статические структуры данных

### 2.3.1 Векторы

Самая простая статическая структура - это вектор. Вектор - это чисто линейная упорядоченная структура, где отношение между ее элементами есть строго выраженная последовательность элементов структуры (рис. 2.5).

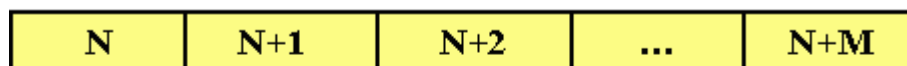


Рис. 2.5

Каждый элемент вектора имеет свой индекс, определяющий положение данного элемента в векторе. Поскольку индексы являются целыми числами, над ними можно производить операции и, таким образом, вычислять положение элемента в структуре на логическом уровне доступа. Для доступа к элементу вектора, достаточно просто указать имя вектора (элемента) и его индекс.

Для доступа к этому элементу используется функция адресации, которая формирует из значения индекса адрес слота, где находится значение исходного элемента. Для объявления в программе вектора необходимо указать его имя, количество элементов и их тип (тип данных)

Пример:

M1: Array [1..100] of integer;

M2: Array [1..10] of real;

Вектор состоит из однотипных данных и количество их строго определено.

### 2.3.2 Массивы

В общем случае элемент массива - это есть элемент вектора, который сам по себе тоже является элементом структуры (рис. 2.6).

<b><math>V_{11}</math></b>	<b><math>V_{12}</math></b>	<b>...</b>	<b><math>V_{1N}</math></b>
<b><math>V_{21}</math></b>	<b><math>V_{22}</math></b>	<b>...</b>	<b><math>V_{2N}</math></b>
<b><math>V_{31}</math></b>	<b><math>V_{32}</math></b>	<b>...</b>	<b><math>V_{3N}</math></b>
<b><math>V_{41}</math></b>	<b><math>V_{42}</math></b>	<b>...</b>	<b><math>V_{4N}</math></b>

Рис. 2.6

Для доступа к элементу двумерного массива необходимы значения пары индексов (номер строки и номер столбца, на пересечении которых находится элемент). На физическом уровне двумерный массив выглядит также, как и одномерный (вектор), причем трансляторы представляют массивы либо в виде строк, либо в виде столбцов.

### 2.3.3 Записи

Запись представляет из себя структуру данных последовательного типа, где элементы структуры расположены один за другим как в логическом, так и в физическом представлении. Запись предполагает множество элементов разного типа. Элементы данных в записи часто называют полями записи.

Пример:

### *Запись студентов*

621	Иванов И.И.	ОАПП	95-ОА-21
-----	-------------	------	----------

Рис. 2.7

Логическая структура записи может быть представлена как в графическом виде, так и в табличном.

### *Логическая структура*

Номер	Фамилия	Факультет	Группа
-------	---------	-----------	--------

Рис. 2.8

### *Графическая структура*



Рис.2.9

Элемент записи может включать в себя записи. В этом случае возникает сложная иерархическая структура данных.

### *Пример:*

Необходимо заполнить запись о студенте, содержащую следующую информацию: N - порядковый номер студента; Имя студента, в составе которого должны быть: Фамилия, Имя, Отчество; Анкетные данные студента: год рождения, место рождения, родители: мать, отец; Факультет; Группа;



Оценки, полученные в сессию: по английскому языку и микропроцессорам.

Ниже приведены два логических представления структуры этой записи.

### Графическая структура

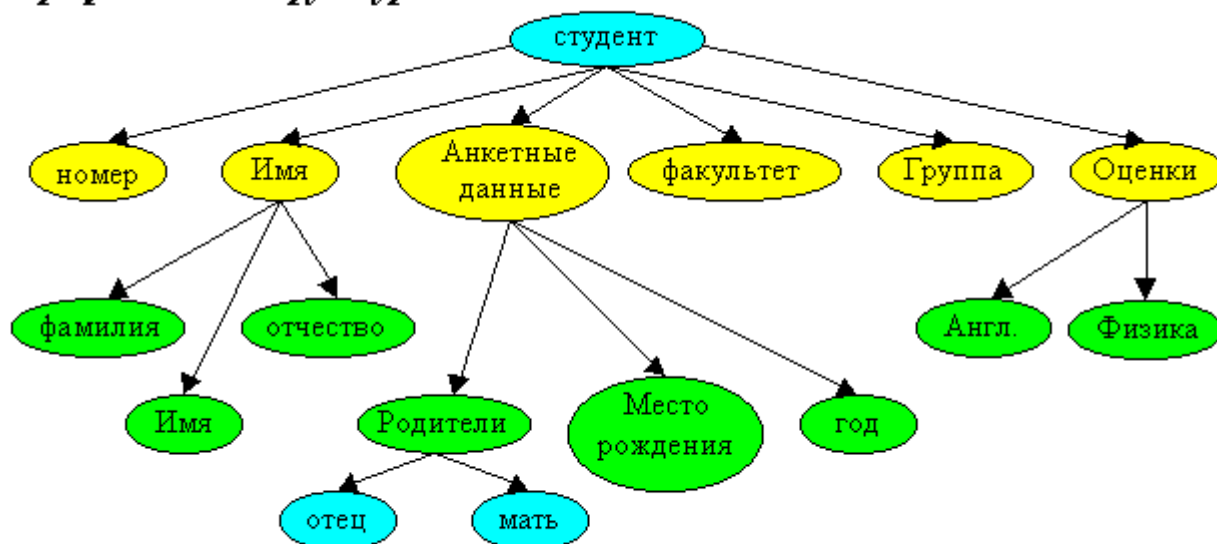


Рис.2.10

Получена четырехуровневая иерархическая структура данных. Информация содержится в листьях, остальные узлы служат для указания пути к листьям.

1-ый уровень	Студент = запись
2-ой уровень	Номер
2-ой уровень	Имя = запись
3-ий уровень	Фамилия
3-ий уровень	Имя
3-ий уровень	Отчество
2-ой уровень	Анкетные данные = запись
3-ий уровень	Место рождения
3-ий уровень	Год рождения
3-ий уровень	Родители = запись
4-ый уровень	Мать
4-ый уровень	Отец
2-ой уровень	Факультет

2-ой уровень	Группа
2-ой уровень	Оценки = запись
3-ий уровень	Английский
3-ий уровень	Физика

Эта структура называется вложенной записью.

### ***Операции над записями:***

1. Прочтение содержимого поля записи.
2. Занесение информации в поле записи.
3. Все операции, которые разрешаются над полем записи, соответствующего типа.

### **2.3.4 Таблицы**

Таблица - это конечный набор записей (рис. 2.11).

N	ФИО	ГРУППА	ФАКУЛЬТЕТ	АНГЛ.	ФИЗИКА
1					
2					
...					
n					

Рис. 2.11.

При задании таблицы указывается количество содержащихся в ней записей.

Пример:

```
Type ST = Record
  Num: Integer;
  Name: String[15];
  Fak: String[5];
  Group: String[10];
```

Angl: Integer;  
Physic: Integer;

var

Table: Array [1..19] of St;

Элементом данных таблицы является запись. Поэтому операции, которые производятся с таблицей - это операции, производимые с записью.

### ***Операции с таблицами:***

1. Поиск записи по заданному ключу.
2. Занесение новой записи в таблицу.

Ключ - это идентификатор записи. Для хранения этого идентификатора отводится специальное поле.

Составной ключ - ключ, содержащий более двух полей.

## **2.4 Полустатические структуры данных**

К полустатическим структурам данных относят стеки, деки и очереди.

### **Списки**

Это набор связанных элементов данных, которые в общем случае могут быть разного типа.

Пример списка:

$E_1, E_2, \dots, E_n, \dots$   $n > 1$  и не зафиксировано.

Количество элементов списка может меняться в процессе выполнения программы. Различают 2 вида списков:

- 1) Несвязные
- 2) Связные

В несвязных списках связь между элементами данных выражена неявно. В связных списках в элемент данных занос-

сится указатель связи с последующим или предыдущим элементом списка.

Стеки, деки и очереди - это несвязные списки. Кроме того они относятся к последовательным спискам, в которых неявная связь отображается их последовательностью.

### 2.4.1 Стеки

Очередь вида LIFO (Last In First Out - Последним пришел, первым ушел), при которой на обслуживание первым выбирается тот элемент очереди, который поступил в нее последним, называется стеком. Это одна из наиболее употребляемых структур данных, которая оказывается весьма удобной при решении различных задач.

В силу указанной дисциплины обслуживания, в стеке доступна единственная его позиция, которая называется вершиной стека - это позиция, в которой находится последний по времени поступления в стек элемент. Когда мы заносим новый элемент в стек, то он помещается поверх прежней вершины и теперь уже сам находится в вершине стека. Выбрать элемент можно только из вершины стека; при этом выбранный элемент исключается из стека, а в его вершине оказывается элемент, который был занесен в стек перед выбранным из него элементом (структура с ограниченным доступом к данным).

Графически стек можно представить следующим образом:

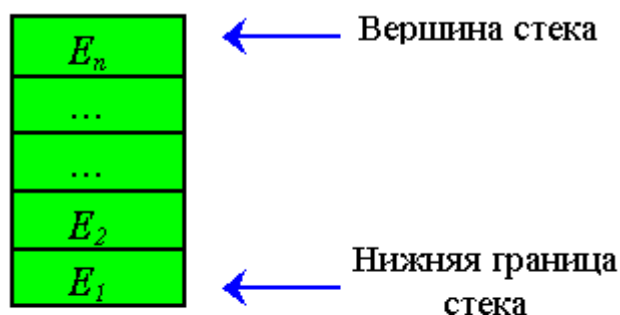


Рис. 2.12

Первый элемент заносится вниз стека. Выборка из стека осуществляется с вершины, поэтому стек является структурой с ограниченным доступом.

***Операции, производимые над стеками:***

1. Занесение элемента в стек.

Push(S,x), где S - идентификатор стека, x - заносимый элемент.

2. Выборка элемента из стека.

Pop(S)

3. Определение пустоты стека.

Empty(S)

4. Прочтение элемента без его выборки из стека.

StackTop(S)

5 Определение переполнения стека (для полустатических структур)

Full(S)

Для реализации полустатического стека в качестве вспомогательной структуры используется массив.

**Алгоритмы основных операций с полустатическим стеком:**

i – указатель вершины стека.

Push(S,x)

$i = i + 1$

$S(i) = x$

return

Pop(S)

$x = S(i)$

$i = i - 1$

*return*

Empty(S)

*if i = 0*

*then “пусто”*

*Stop*

*return*

*endif*

*условие  $i=0$  означает, что стек пуст*

Full(S)

*if i = maxS*

*then “переполнение”*

*Stop*

*return*

*endif*

StackTop(S)

*x = S(i)*

*return*

При выполнении операции выборки из стека сначала необходимо осуществить проверку на пустоту стека. Если он пуст, то Empty возвращает значение True. Если Empty возвращает False, то это означает, что стек не пуст и из него еще можно выбирать элементы.

Если при выборке проверять стек на пустоту, а при занесении элемента проверять стек на переполнение, то алгоритмы операций считывания и занесения элемента будут следующими:

Pop(S)

*if i = 0 then “пусто”*

*Stop*

```

return
endif
 $x = S(i)$ 
 $i = i - 1$ 
return

Push(S,i)
if  $i = \max S$ 
then “переполнение”
Stop
return

endif
 $i = i + 1$ 
 $S(i) = x$ 
return

```

### 2.4.2 Очередь

Понятие очереди всем хорошо известно из повседневной жизни. Элементами очереди в общем случае являются заказы на то или иное обслуживание.

В программировании имеется структура данных, которая называется очередью. Эта структура данных используется, например, для моделирования реальных очередей с целью определения их характеристик при данном законе поступления заказов и дисциплине их обслуживания. По своему существу очередь является полустатической структурой — с течением времени и длина очереди, и состав могут изменяться.

На рис. 2. 13 приведена очередь, содержащая четыре элемента — А, В, С и D. Элемент А расположен в начале очереди, а элемент D — в ее конце. Элементы могут удаляться только из начала очереди, то есть первый помещаемый в очередь элемент удаляется первым. По этой причине очередь часто называют списком, организованным по принципу «первый размещенный первым удаляется» в противоположность

принципу стековой организации — «последний размещенный первым удаляется».

Дисциплину обслуживания, в которой заказ, поступивший в очередь первым, выбирается первым для обслуживания (и удаляется из очереди), называется FIFO (First In First Out - Первым пришел, первым ушел). Очередь открыта с обеих сторон.

Таким образом, изъятие компонент происходит из начала очереди, а запись — в конец. В этом случае вводят два указателя: один - на начало очереди, второй - на ее конец.

Реальная полустатическая очередь создается в памяти ЭВМ в виде одномерного массива с конечным числом элементов, при этом необходимо указать тип элементов очереди, а также необходима переменная в работе с очередью.

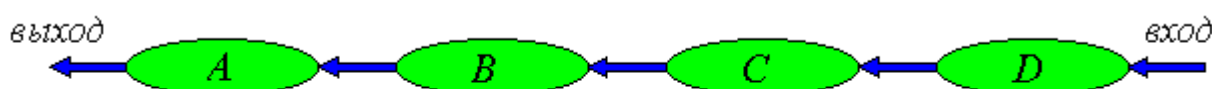


Рис. 2.13.

Физически очередь занимает сплошную область памяти, и элементы следуют друг за другом, как в последовательном списке.

### ***Операции, производимые над очередью:***

Для очереди определены три примитивные операции. Операция `insert (q,x)` помещает элемент `x` в конец очереди `q`. Операция `remove(q)` удаляет элемент из начала очереди `q` и присваивает его значение переменной `x`. Третья операция, `empty (q)`, возвращает значение `true` или `false` в зависимости от того, является ли данная очередь пустой или нет. Учитывая то, что полустатическая очередь реализуется на одномерном массиве, необходимо следить за возможностью его переполнения. С этой целью вводится операция `full(q)`.



Операция *insert* для динамической очереди может быть выполнена всегда, поскольку на количество элементов, которые может содержать очередь, никаких ограничений не накладывается. В полустатической очереди при проведении операции вставки необходимо осуществление проверки на переполнение, поскольку массив, с помощью которого она реализуется, состоит из конечного числа элементов. Операция *remove* применима только к непустой очереди, поскольку невозможно удалить элемент из очереди, не содержащей элементов. Результатом попытки удалить элемент из пустой очереди является возникновение исключительной ситуации, которая носит название **потеря значимости**. Операция *empty*, разумеется, выполнима всегда.

### **Алгоритмы основных операций с полустатической очередью.**

Для работы с очередью вводят два указателя: один – на начало очереди (F), второй – на ее конец (R).

*Full (q)*

*if*  $R = \max Q$  *then* *full* = *true*

*else* *full* = *false*

*endif*

*return*

*Empty (q)*

*if*  $R < F$  *then* *empty* = *true*

*else* *empty* = *false*

*endif*

*return*

*Insert (q, x)*

*Full (q)*

*if* *full* = *true* *then* ‘переполнение’

*stop*

```

return
endif
R = R + 1
q(R) = x
return

Remove (q)
Empty (q)
if empty = true then 'пусто'
stop
return
endif
x = q(F)
F = F + 1
return

```

**Пример работы с очередью с использованием стандартных процедур.**

Пусть очередь реализуется с помощью массива из 5 элементов.

$\text{MaxQ} = 5$

$R = 0, F = 1$

Условие *пустоты* очереди  $R < F$  ( $0 < 1$ )

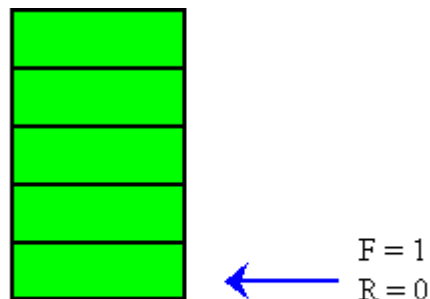


Рис. 2.14.

Производим вставку элементов А, В и С в очередь.

Insert(q, A);  
Insert(q, B);  
Insert(q, C);

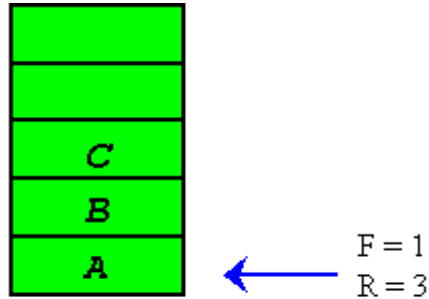


Рис. 2.15.

Убираем элементы А и В из очереди.

Remove(q);  
Remove(q);

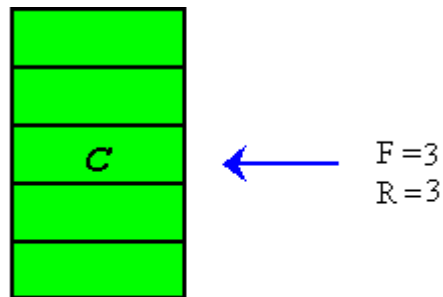


Рис. 2.16.

Добавляем элементы D и E:

Insert (q, D);  
Insert (q, E);

Убираем элементы C,D и E из очереди:

Remove (q);  
Remove (q);  
Remove (q).

К сожалению, при подобном представлении очереди возникла абсурдная ситуация, при которой очередь является пустой, однако новый элемент разместить в ней нельзя, так как  $R = \max Q$ .

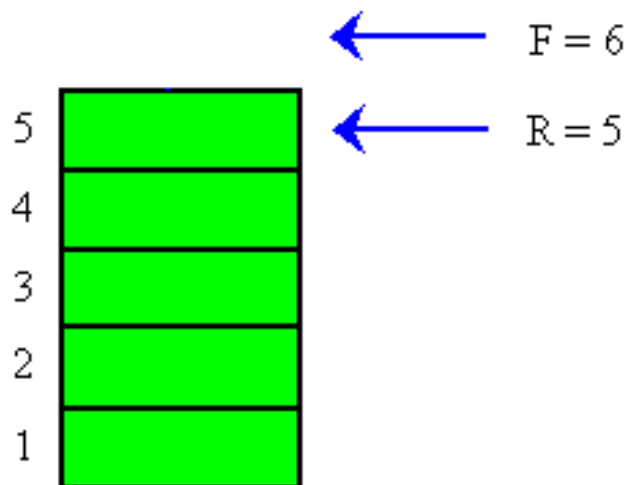


Рис. 2.16`

Одним из решений возникшей проблемы может быть модификация операции remove таким образом, что при удалении очередного элемента вся очередь смещается к началу массива.

Переменная  $F$  больше не требуется, поскольку первый элемент массива всегда является началом очереди.

Пустая очередь представлена очередью, для которой значение  $R$  равно нулю.

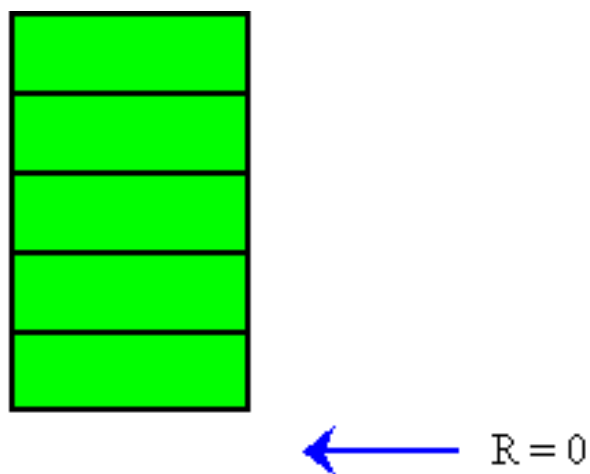


Рис. 2.16''

Произведем вставку элементов А, В и С в очередь:

Insert (q, A);

Insert (q, B);

Insert (q, C);

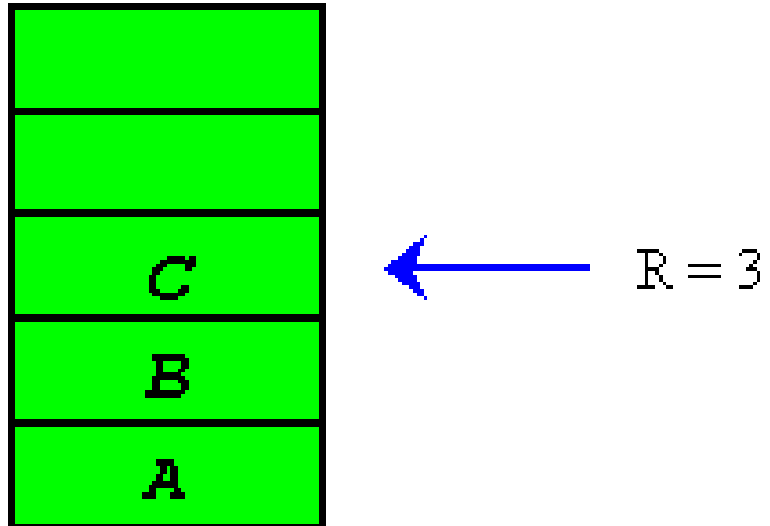


Рис. 2.16'''

Убираем элемент А из очереди:

Remove (q)

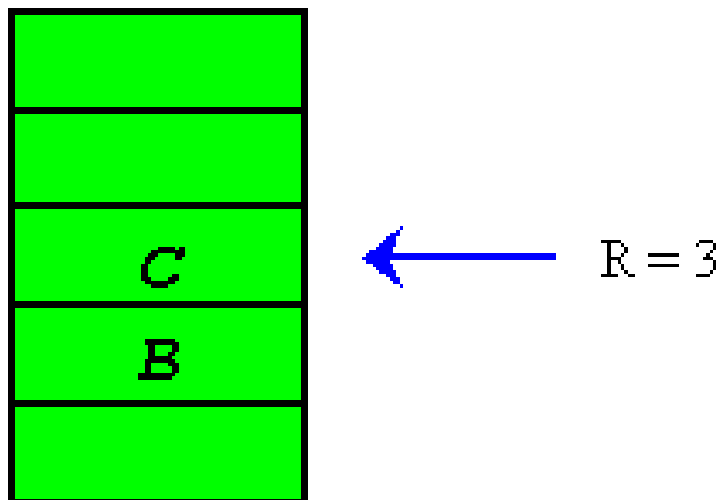


Рис. 2.16''''

Операция remove (q) может быть в этом случае реализована следующим образом:

```
Remove (q)  
x = q(1)  
for i = 1 to R-1  
    q(i) = q(i+1)  
next i  
R = R-1  
return
```

При такой модификации элементы очереди смещаются к началу массива, R=2.

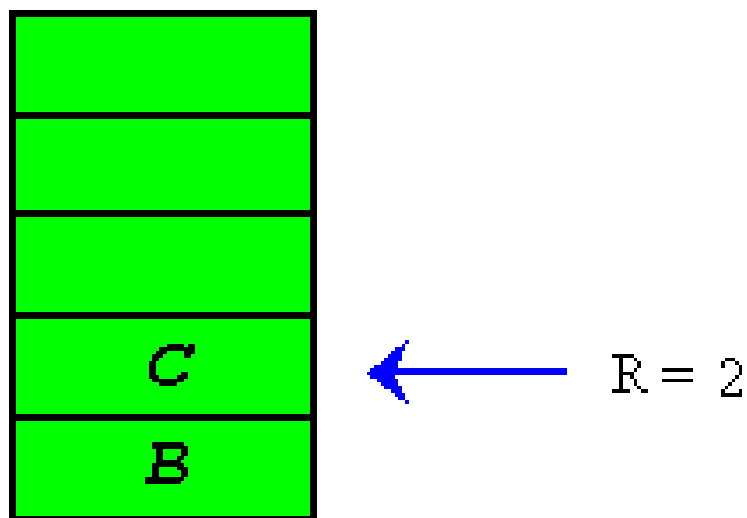


Рис 2.16''''.

Однако этот метод весьма непроизводителен. Каждое удаление требует перемещения всех оставшихся в очереди элементов. Если очередь содержит 500 или 1000 элементов, то очевидно, что это весьма неэффективный способ. Кроме того, операция удаления элемента из очереди логически предполагает манипулирование только с одним элементом, т. е. с тем, который расположен в начале очереди. Реализация

данной операции должна отражать именно этот факт, не производя при этом множества дополнительных действий.

Другой способ предполагает рассматривать массив, который содержит очередь в виде замкнутого кольца, а не линейной последовательности, имеющей начало и конец. Это означает, что первый элемент очереди следует сразу же за последним. Это означает, что даже в том случае, если последний элемент занят, новое значение может быть размещено сразу же за ним на месте первого элемента, если этот первый элемент пуст.

### Организация кольцевой очереди.

Рассмотрим пример. Предположим, что очередь содержит три элемента - в позициях 3, 4 и 5 пятиэлементного массива. Хотя массив и не заполнен, последний элемент очереди занят.

Если теперь делается попытка поместить в очередь элемент G, то он будет записан в первую позицию массива, как это показано на рис. 2.17. Первый элемент очереди есть Q(3), за которым следуют элементы Q(4), Q(5) и Q(1).

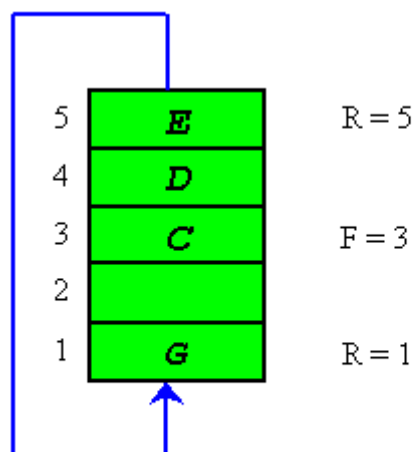


Рис. 2.17

К сожалению, при таком представлении довольно трудно определить, когда очередь пуста. Условие  $R < F$  больше не

годится для такой проверки, поскольку на рис. 2. 17 показан случай, при котором данное условие выполняется, но очередь при этом не является пустой.

Одним из способов решения этой проблемы является введение соглашения, при котором значение **F** есть индекс элемента массива, немедленно предшествующего первому элементу очереди, а не индексу самого первого элемента. В этом случае, поскольку R содержит индекс последнего элемента очереди, условие  $F = R$  подразумевает, что очередь пуста.

Отметим, что перед началом работы с очередью, в F и R устанавливается значение последнего индекса массива  $\max Q$ , а не 1 и 0, поскольку при таком представлении очереди последний элемент массива немедленно предшествует первому элементу. Поскольку  $R = F$ , то очередь изначально пуста.

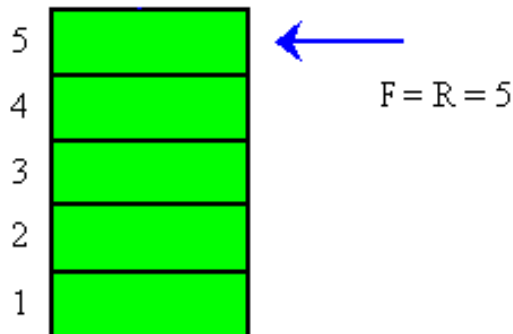


Рис. 2.17'

### Основные операции с кольцевой очередью:

1. Вставка элемента  $q$  в очередь  $x$ .  
 $\text{Insert}(q, x)$
2. Извлечение элемента из очереди  $x$ .  
 $\text{Remove}(q)$   
Проверка очереди на пустоту.



Empty(q)

4.Проверка очереди на переполнение Full(q).

Алгоритм проверки на пустоту кольцевой очереди следующий:

Empty(q)

if  $F = R$

then  $empty = true$

else  $empty = false$

endif

return

Алгоритм извлечения элемента из кольцевой очереди следующий:

Remove(q)

empty (q)

if  $empty = true$

then “нельзя”

stop

endif

if  $F = maxQ$

then  $F = 1$

else  $F = F + 1$

endif

$x = q(F)$

return

Отметим, что значение F должно быть модифицировано до момента извлечения элемента.

Переполнение очереди происходит в том случае, если весь массив уже занят элементами очереди, и при этом делается попытка разместить в ней еще один элемент.

Для того чтобы запрограммировать операцию вставки, должна быть проанализирована ситуация, при которой возникает переполнение. Переполнение происходит в том случае, если весь массив уже занят элементами очереди и при этом делается попытка разместить в ней еще один элемент. Рассмотрим, например, очередь на рис. 2. 17. В ней находятся три элемента — C, D и E, соответственно расположенные в Q (3), Q (4) и Q (5). Поскольку последний элемент в очереди занимает позицию Q (5), значение R равно 5. Так как первый элемент в очереди находится в Q (3), значение F равно 2. На рис. 2. 17 в очередь помещается элемент G, что приводит к соответствующему изменению значения R. Если произвести следующую вставку, то массив становится целиком заполненным, и попытка произвести еще одну вставку приводит к переполнению. Это регистрируется тем фактом, что  $F = R$ , а это как раз и указывает на переполнение. Очевидно, что при такой реализации нет возможности сделать различие между пустой и заполненной очередью.

Одно из решений состоит в том, чтобы пожертвовать одним элементом массива и позволить очереди расти **до объема на единицу меньшего максимального**. Рисунок 2.17'' иллюстрирует данное соглашение при реализации очереди с помощью массива из 5 элементов.

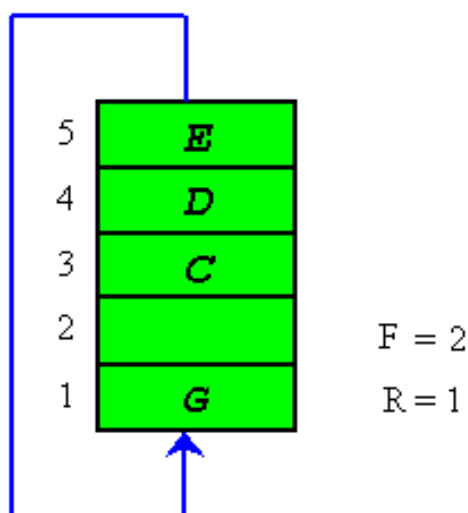


Рис. 2.17''

Если, например, очередь реализуется на массиве из 100 элементов, то очередь может содержать до 99 элементов. Попытка разместить в очереди 100-й элемент приведет к переполнению.

Проверка на переполнение в подпрограмме *insert* производится *после* установления нового значения для *R*, в то время как проверка на потерю значимости в подпрограмме *remove* производится сразу же после входа в подпрограмму *до* момента обновления значения *F*.

Алгоритм операции вставки элемента в кольцевую очередь при такой реализации следующий:

```

Insert(q,x)
if R = maxQ
    then R = 1
    else R = R+1
endif
‘проверка на переполнение’
if R = F
    then print «переполнение очереди»
    stop
endif
q (R) = x
return
    
```

### 2.4.3 Дек

От английского DEQ - Double Ended Queue (Очередь с двумя концами)

Особенностью деков является то, что запись и считывание элементов может производиться с двух концов (рис. 2.18).

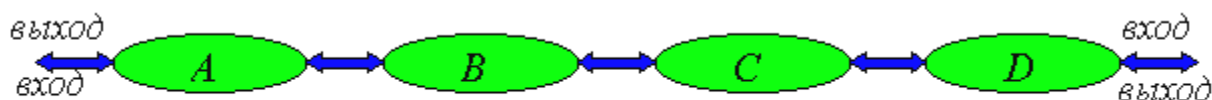


Рис. 2.18.

Дек можно рассматривать и в виде двух стеков, соединенных нижними границами. Возьмем пример, иллюстрирующий принцип построения дека. Рассмотрим состав на железнодорожной станции. Новые вагоны к составу можно добавлять либо к его концу, либо к началу. Аналогично, чтобы отсоединить от состава вагон, находящийся в середине, нужно сначала отсоединить все вагоны или вначале, или в конце состава, отсоединить нужный вагон, а затем присоединить их снова.

### ***Операции над деками:***

1. Insert - вставка элемента.
2. Remove - извлечение элемента из дека.
3. Empty - проверка на пустоту.
4. Full - проверка на переполнение.

### **Контрольные вопросы**

1. Что такое структуры данных?
2. Назовите уровни представления данных?
3. Какова классификация структур данных?
4. Какая статическая структура является самой простой?
5. Перечислите основные элементы таблицы.
6. Назовите их основные особенности.
7. Что такое вектор?
8. Что представляет из себя запись?
9. Какова структура записи?
10. К каким структурам данных относятся очереди и стеки?
11. Каково правило выборки элемента из стека?

12. Какая операция читает верхний элемент стека без его удаления?
13. Какую дисциплину обслуживания принято называть FIFO, а какую - LIFO?
14. Признак заполнения кольцевой очереди?
15. Признак пустой очереди?
16. Что называется списком?
17. Перечислите виды списков.
18. Назовите элементы очереди.
19. Как организуется кольцевая очередь?
20. Какова особенность деков?

### 3. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

До сих пор мы рассматривали только статические программные объекты. Однако использование при программировании только статических объектов может вызвать определенные трудности, особенно с точки зрения получения эффективной машинной программы. Дело в том, что иногда мы заранее не знаем не только размера значения того или иного программного объекта, но также и того, будет ли существовать этот объект или нет. Такого рода программные объекты, которые возникают уже в процессе выполнения программы или размер значений которых определяется при выполнении программы, называются динамическими объектами.

Динамические структуры данных имеют 2 особенности:

1) Заранее не определено количество элементов в структуре.

2) Элементы динамических структур не имеют жесткой линейной упорядоченности. Они могут быть разбросаны по памяти.

Чтобы связать элементы динамических структур между собой в состав элементов помимо информационных полей входят поля указателей (рис. 3.1) (связок с другими элементами структуры).

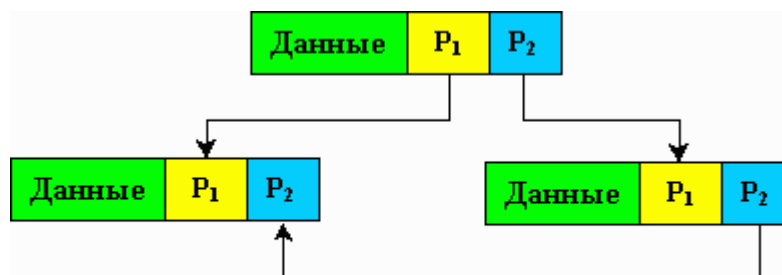


Рис. 3.1.

$P_1$  и  $P_2$  это указатели, содержащие адреса элементов, с которыми связаны соответствующие элементы структуры.

### 3.1 Связные списки

Наиболее распространенными динамическими структурами являются связанные списки. С точки зрения логического представления различают линейные и нелинейные списки.

В линейных списках связи строго упорядочены: указатель предыдущего элемента содержит адрес последующего элемента или наоборот.

К линейным спискам относятся односвязные и двусвязные списки. К нелинейным - многосвязные.

Элемент списка в общем случае представляет собой информационное поле и одно или несколько полей указателей.

#### 3.1.1 Односвязные списки

Элемент односвязного списка содержит, как минимум, два поля (рис. 3.2): информационное поле (INFO) и поле указателя (PTR).

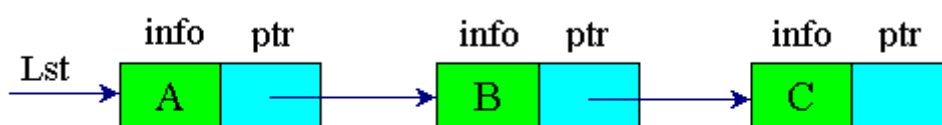


Рис. 3.2

Особенностью указателя является то, что он дает только адрес последующего элемента списка. Поле указателя последнего элемента в списке является пустым (NIL). LST - указатель на начало списка. Список может быть пустым, тогда LST будет равен NIL.

Доступ к элементу списка осуществляется только от его начала, то есть обратной связи в этом списке нет.

В дальнейшем при изучении списков будем использовать следующую терминологию:

$p$  - указатель

$node(p)$  – узел, на который ссылается указатель  $p$  (при этом неважно в какое место изображения элемента (узла) списка он направлен на рисунке)

$ptr(p)$  – ссылка на последующий элемент узла  $node(p)$

$ptr(ptr(p))$  – ссылка последующего для  $node(p)$  узла на последующий для него элемент

### 3.1.2 Кольцевой односвязный список

Кольцевой односвязный список получается из обычного односвязного списка путем присваивания указателю последнего элемента списка значение указателя начала списка (рис 3.3).

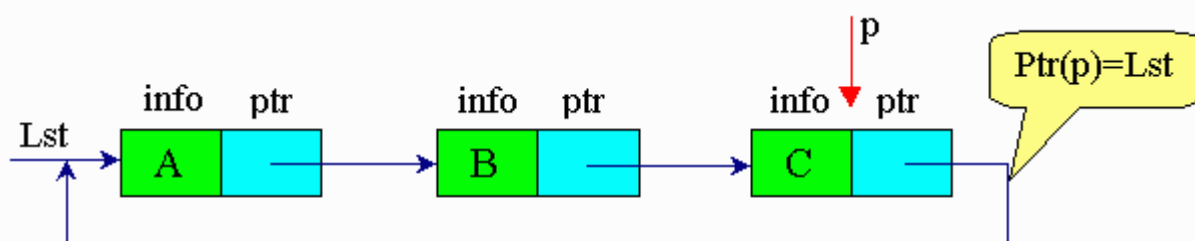


Рис. 3.3

### Простейшие операции, производимые над односвязными списками

#### Вставка в начало односвязного списка.

Надо вставить в начало односвязного списка элемент с информационным полем  $x$ . Для этого необходимо сгенерировать пустой элемент ( $P=GetNode$ ). Информационному полю этого элемента присвоить значение  $x$  ( $INFO(P)=x$ ), значению указателя на этот элемент присвоить значение указателя на



начало списка ( $\text{Ptr}(P) = \text{Lst}$ ), значению указателя начала списка присвоить значение указателя  $P$  ( $\text{Lst} = P$ ).

Алгоритм операции вставки в начало списка следующий:

*$P = \text{GetNode}$*

*$\text{Info}(P) = x$*

*$\text{Ptr}(P) = \text{Lst}$  ‘( $\text{Lst}$  уже не указывает на начало списка)’*

*$\text{Lst} = P$  ‘(новое начало)’*

*return*

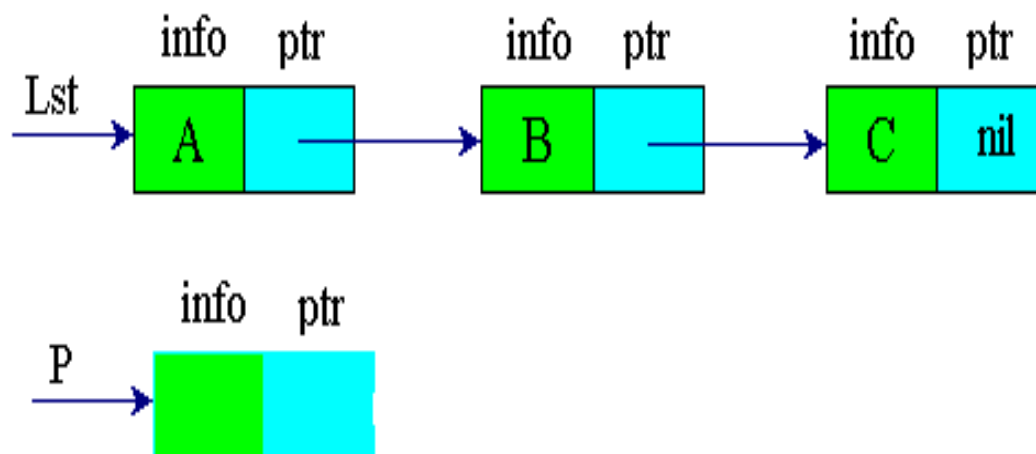


Рис. 3.3'

### **Удаление элемента из начала односвязного списка.**

Надо удалить первый элемент списка, но запомнить информацию, содержащуюся в поле Info этого элемента. Для этого введем указатель  $P$ , который будет указывать на удаляемый элемент ( $P = \text{Lst}$ ). В переменную  $x$  занесем значение информационного поля Info удаляемого элемента ( $x = \text{Info}(P)$ ). Значению указателя на начало списка присвоим значение указателя следующего за удаляемым элементом ( $\text{Lst} = \text{Ptr}(P)$ ). Удалим элемент ( $\text{FreeNode}(P)$ ).

Алгоритм операции удаления из начала списка следующий:

*$P = \text{Lst}$*

```

x=Info(P)
Lst = Ptr(P)
FreeNode(P)
return

```

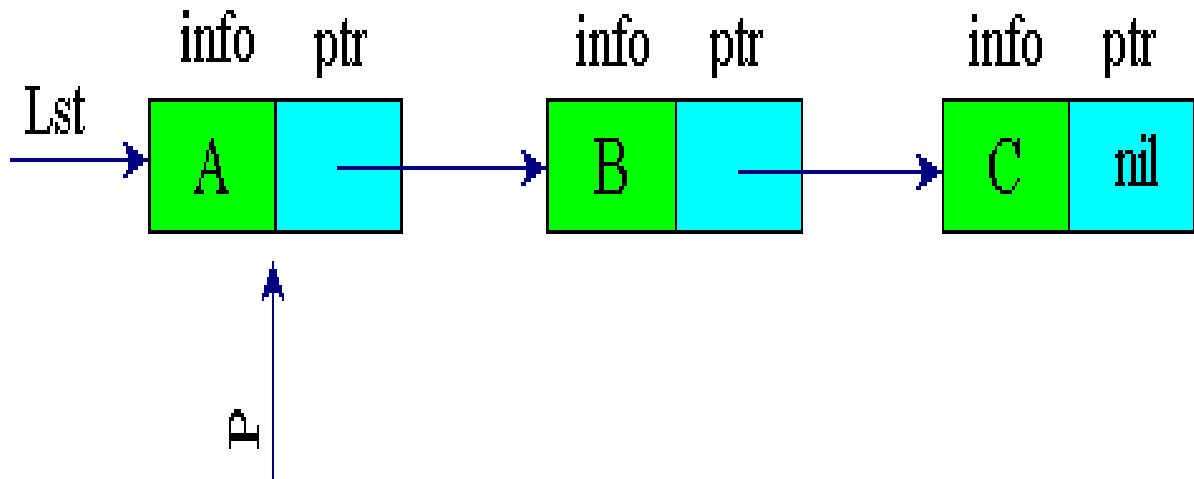


Рис. 3.3''

### 3.1.3 Двусвязный список

Использование однонаправленных списков при решении ряда задач может вызвать определенные трудности. Дело в том, что по однонаправленному списку можно двигаться только в одном направлении, от первого звена к последнему звену списка. Между тем нередко возникает необходимость произвести какую-либо обработку элементов, предшествующих элементу с заданным свойством. Однако после нахождения элемента с этим свойством в односвязном списке у нас нет возможности получить достаточно удобный и быстрый доступ к предыдущим элементам - для достижения этой цели придется усложнить алгоритм, что неудобно и нерационально.

Для устранения этого неудобства добавим в каждое звено списка еще одно поле, значением которого будет ссылка на предыдущее звено списка. Динамическая структура, со-

стоящая из элементов такого типа, называется двунаправленным или двусвязным списком.

Двусвязный список характеризуется тем, что у любого элемента есть два указателя. Один указывает на предыдущий элемент (обратный), другой указывает на следующий элемент (прямой) (рис. 3.4).



Фактически двусвязный список это два односвязных списка с одинаковыми элементами, записанных в противоположной последовательности.

### 3.1.4 Кольцевой двусвязный список

В программировании двусвязные списки часто обобщают следующим образом: в качестве значения поля Rptr последнего звена принимают ссылку на первое звено, а в качестве значения поля Lptr первого звена - ссылку на последнее звено. Список замыкается в своеобразное кольцо: двигаясь по ссылкам, можно от последнего звена переходить к первому и наоборот.



## 3.2 Реализация стеков с помощью односвязных списков

Любой односвязный список может рассматриваться в виде стека. Однако односвязный список имеет преимущество

по сравнению со стеком, реализованном на одномерном массиве, так как заранее не задан его размер.

### **Стековые операции, применимые к спискам**

1). Чтобы добавить элемент в стек, надо в алгоритме вставки в начало списка заменить указатель Lst на указатель S вершины стека (операция Push(S, X)).

```
P = GetNode  
Info(P) = x  
Ptr(P) = S  
S = P  
return
```

2) . Проверка стека на пустоту (Empty(S))

```
if S = Nil  
    then print "Стек пуст"  
        return  
endif  
return
```

3) . Выборка элемента из стека (POP(S))

```
Empty(S)  
P = S  
X = Info(P)  
S = Ptr(P)  
FreeNode(P)  
return
```

### **Операции с очередью, применимые к спискам.**

Указатель начала списка принимаем за указатель начала очереди F, а указатель R, указывающий на последний элемент списка - за указатель конца очереди.

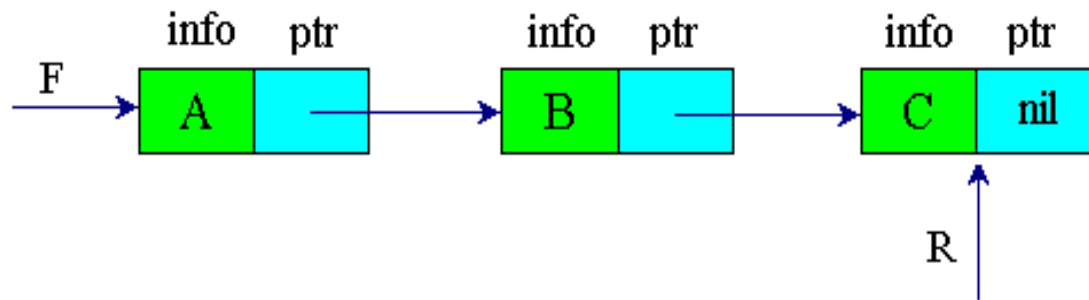


Рис. 3.5'

1). Алгоритм операции проверки очереди на пустоту (Empty(Q)) следующий:

```

If  $F = nil$ 
    then print "Очередь пуста"
    return
endif
return

```

2). Операция удаления из очереди должна проходить из ее начала. Алгоритм операции удаления из очереди (Remove(Q)) следующий:

```

If  $F = nil$ 
    then print "Очередь пуста"
    return
endif
 $P = F$ 
 $F = Ptr(P)$ 
 $X = Info(P)$ 
FreeNode(P)

```

*return*

3). Операция вставки в очередь должна осуществляться к ее концу. Алгоритм операции вставки в очередь. ( $\text{Insert}(Q, X)$ ) следующий:

$P = \text{GetNode}$

$\text{Info}(P) = x$

$\text{Ptr}(P) = \text{Nil}$

$\text{Ptr}(R) = P$

$R = P$

*return*

### **3.3 Организация операций *Getnode*, *Freenode* и утилизация освободившихся элементов**

Для более эффективного использования памяти компьютера (для исключения *мусора*, то есть неиспользуемых элементов) при работе его со списками создается свободный список, имеющий тот же формат полей, что и у функциональных списков.

Если у функциональных списков разный формат, то надо создавать свободный список каждого функционального списка.

Количество элементов в свободном списке должно быть определено задачей, которую решает программа. Как правило, свободный список создается в памяти машины как стек. При этом создание (*GetNode*) нового элемента эквивалентно выборке элемента свободного стека, а операция *FreeNode* - добавлению в свободный стек освободившегося элемента.

Пусть нам необходимо создать пустой список по типу стека (рис. 3.6) с указателем начала списка - *AVAIL*.

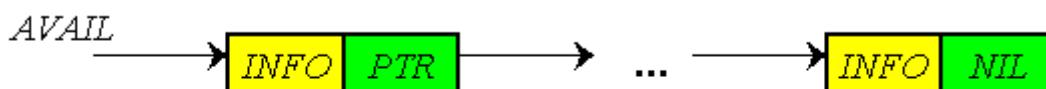


Рис. 3.6

Необходимы функции, которые позволят нам создавать пустой элемент списка и освобождать элемент из списка.

### 3.3.1 Операция $P=GetNode$ .

Разработаем функцию, которая будет создавать пустой элемент списка с указателем  $P$ .

Для реализации операции  $GetNode$  необходимо указателю сгенерированного элемента присвоить значение указателя начала свободного списка, а указатель начала перенести к следующему элементу. При этом надо проверить, есть ли элементы в списке. Пустота свободного списка ( $Avail = Nil$ ), эквивалентна переполнению функционального списка.

```

If Avail = Nil
    then Print "Переполнение"
        return
    else    P = Avail
           Avail = Ptr(Avail)
endif
return

```

### 3.3.2 Операция $FreeNode(P)$ .

При освобождении элемента из функционального списка, он заносится в свободный список.

```

Ptr(P) = Avail
Avail = P
return

```

### **3.3.3 Утилизация освободившихся элементов в многосвязных списках**

Стандартные операции возвращения освободившегося элемента списка в пул свободных элементов не всегда дают эффект, если используются нелинейные многосвязные списки.

Первый способ утилизации - метод счетчиков. В каждый элемент многосвязного списка вставляется поле счетчика, который считает количество ссылок на данный элемент. Когда счетчик элемента оказывается в нулевом состоянии, а поля указателей элемента находятся в состоянии nil, этот элемент может быть возвращен в пул свободных элементов.

Второй способ - метод сборки мусора (метод маркера). Если с каким-то элементом установлена связь, то однобитовое поле элемента (маркер) устанавливается в "1", иначе - в "0". По сигналу переполнения ищутся элементы, у которых маркер установлен в ноль, т. е. включается программа сборки мусора, которая просматривает всю отведенную память и возвращает в список свободных элементов все элементы, не помеченные маркером.

### **3.4 Односвязный список, как самостоятельная структура данных**

Просмотр односвязного списка может производиться только последовательно, начиная с головы (с начала) списка. Если необходимо просмотреть предыдущий элемент, то надо снова возвращаться к началу списка. Это - недостаток односвязных списков по сравнению со статическими структурами типа массива. Списковая структура проявляет свои достоинства, когда число элементов списка велико, а вставку или удаление необходимо произвести внутри списка.

Пример:



Необходимо вставить в существующий массив элемент  $X$  между 5 и 6 элементами.

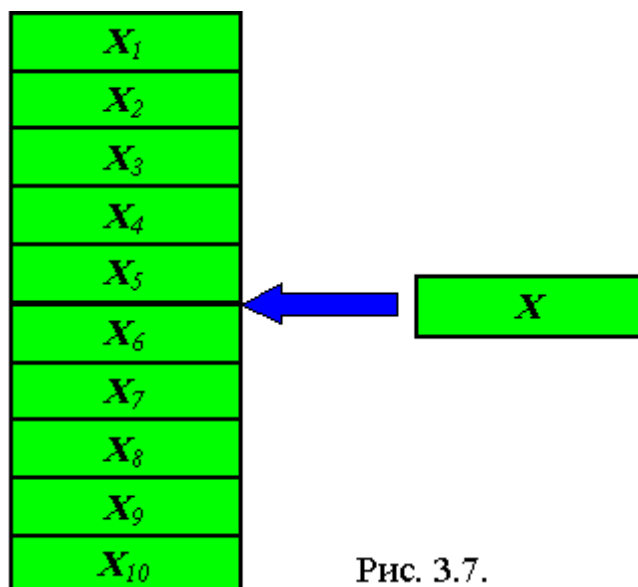


Рис. 3.7.

Для проведения данной операции в массиве нужно “опустить” все элементы, начиная с  $X_6$  - увеличить их индексы на единицу. В результате вставки получаем следующий массив (рис. 3.7, 3.8):

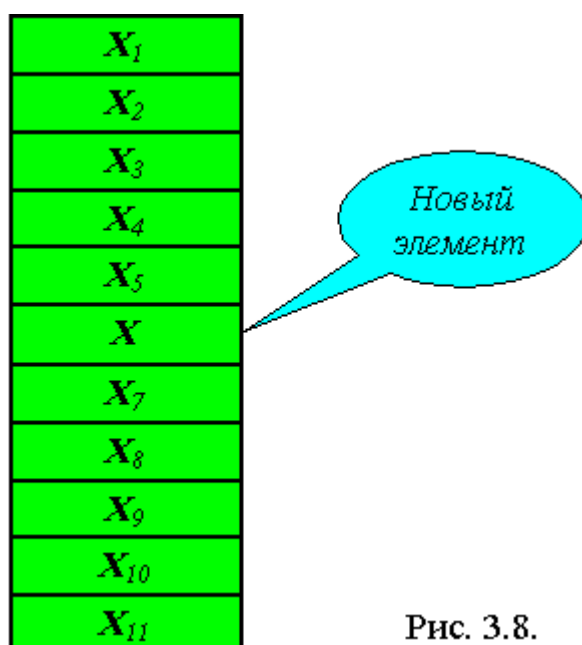


Рис. 3.8.

Данная процедура может занимать очень значительное время. В противоположность этому, в связанном списке операция вставки состоит в изменении значения 2-х указателей и

генерации свободного элемента. Причём время, затраченное на выполнение этой операции, является постоянным и не зависит от количества элементов в списке.

### 3.4.1 Вставка и извлечение элементов из списка

Сначала определяем элемент, **после** которого необходимо вставить элемент в список. Вставка производится с помощью процедуры  $InsAfter(P, x)$ , а удаление -  $DelAfter(P)$ .

При этом рабочий указатель  $P$  будет указывать на элемент, **после** которого необходимо произвести вставку или удаление (рис 3.9).

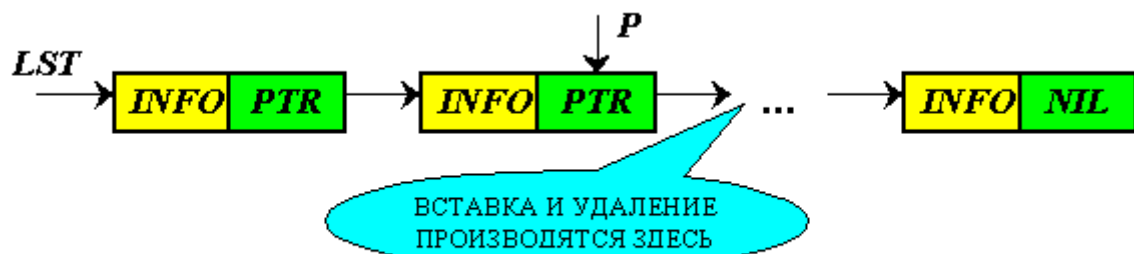


Рис. 3.9.

Пусть необходимо вставить новый элемент с информационным полем  $x$  после элемента, на который указывает рабочий указатель  $P$ .

Для этого:

1) Необходимо сгенерировать новый элемент.

$Q = GetNode$

2) Информационному полю этого элемента присвоить значение  $x$ .

$Info(Q) = x$

3) Вставить полученный элемент.

$Ptr(Q) = Ptr(P)$

$Ptr(P) = Q$

Окончательно алгоритм функции  $InsAfter(P, x)$  будет следующий:

$Q = GetNode$

```

info(Q) = x
ptr(Q) = ptr(P)
ptr(P) = Q
return

```

Пусть необходимо удалить элемент списка, который следует после элемента, на который указывает рабочий указатель Р.

Для этого:

1) Присваиваем Q значение указателя на удаляемый элемент.

$Q = \text{Ptr}(P)$

2) В переменную x сохраняем значение информационного поля удаляемого элемента.

$X = \text{Info}(Q)$

3) Меняем значение указателя на удаляемый элемент на значение указателя на следующий элемент и производим удаление .

$\text{Ptr}(P) = \text{Ptr}(Q)$

$\text{FreeNode}(Q)$

Окончательно алгоритм функции DelAfter(P) будет следующий:

```

Q = ptr(P)
X = info(Q)
ptr(P) = ptr(Q)
FreeNode(Q)
return

```

Для вставки или удаления элемента в списке (не первого) необходимо совершить «проход» по списку до элемента, после которого осуществляется вставка или удаление. Данную операцию называют просмотром односвязного списка при вставке или удалении. Без данной операции невозможно

осуществлять работу со списком, соответственно для нее необходим алгоритм.

Обозначим через  $P$  - рабочий указатель; в начале просмотра  $P = Lst$ .

Введем также указатель  $Q$ , который отстает на один элемент от  $P$ ; в начале просмотра  $Q = nil$ .

Когда указатель  $P$  получит значение  $nil$ , цикл просмотра заканчивается.

Алгоритм просмотра односвязного списка при вставке и удалении следующий:

```
 $Q = Nil$   
 $P = Lst$   
while ( $P \neq nil$ ) do  
     $Q = P$   
     $P = ptr(P)$   
endwhile  
return
```

### 3.4.2 Примеры типичных операций над списками

#### **Задача 1.**

Требуется просмотреть список и удалить элементы, у которых информационные поля равны 4.

Обозначим  $P$  - рабочий указатель, в начале просмотра  $P = Lst$ . Введем также указатель  $Q$ , который отстает на один элемент от  $P$ . Когда указатель  $P$  найдет элемент, последний будет удален относительно указателя  $Q$  как последующий элемент.

Алгоритм решения данной конкретной задачи следующий:

$x = 4$

```

Q = nil
P = Lst
while P <> nil do
    if info(P) = x then
        if Q = nil then
            Pop(Lst)
            P = Lst
        else
            DelAfter(Q)
    endif
    else
        Q = P
        P = Ptr(P)
    endif
endwhile
return

```

### Задача 2.

Дан упорядоченный по возрастанию Info - полей список. Необходимо вставить в этот список элемент со значением *x*, не нарушив упорядоченности списка.

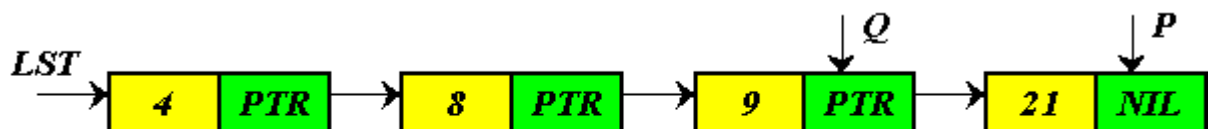


Рис. 3.10.

Пусть  $x = 16$ . Начальное условие:  $Q = Nil$ ,  $P = Lst$ . Вставка элемента должна произойти между 3 и 4 элементом (рис.3.10).

Алгоритм решения данной задачи следующий:

```

X = 16
Q = Nil
P = Lst
while (P <> nil) and (X > info(P)) do
    Q = P
    P = Ptr(P)

```

```

endwhile
if  $Q = nil$  then
     $Push(Lst, X)$ 
endif
InsAfter( $Q, X$ )
return

```

### 3.4.3 Элементы заголовков в списках

Для создания списка с заголовком в начало списка вводится дополнительный элемент, который может содержать информацию о списке (рис. 3.11).

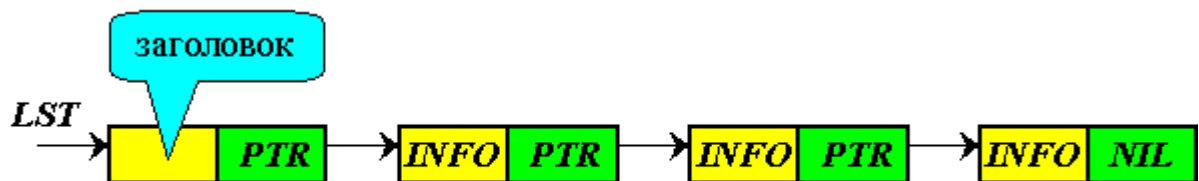


Рис. 3.11.

В заголовок списка часто помещают динамическую переменную, содержащую количество элементов в списке (не считая самого заголовка).

Если список пуст, то остается только заголовок списка (рис. 3.12).

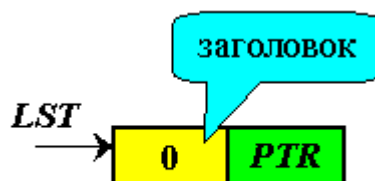


Рис. 3.12.

Также удобно занести в информационное поле заголовка значение указателя конца списка. Тогда, если список используется как очередь, то  $F = Lst$ , а  $R = Info(Lst)$ .

Информационное поле заголовка можно использовать для хранения рабочего указателя при просмотре списка  $P = Info(Lst)$ . То есть заголовок - это дескриптор (описатель) структуры данных.

### 3.5 Нелинейные связанные структуры

Двусвязный список может быть нелинейной структурой данных, если вторые указатели задают произвольный порядок следования элементов (рис.3.13).

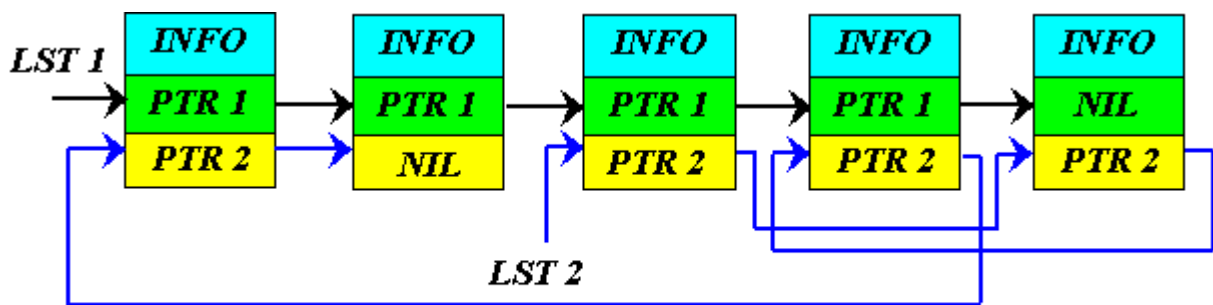


Рис. 3.13

LST1 - указатель на начало первого списка (ориентированного указателями PTR1). Он линейный и состоит из 5-и элементов.

2-ой список образован из этих же самых элементов, но в произвольной последовательности. Началом 2-ого списка является 3-ий элемент, а концом 2-ой элемент.

В общем случае элемент списочной структуры может содержать сколь угодно много указателей, то есть может указывать на любое количество элементов.

Можно выделить 3 признака отличия нелинейной структуры:

1) Любой элемент структуры может ссылаться на любое число других элементов структуры, то есть может иметь любое число полей-указателей.

2) На данный элемент структуры может ссылаться любое число других элементов этой структуры.

3) Ссылки могут иметь вес, то есть подразумевается иерархия ссылок.

Представим, что имеется дискретная система, в графе состояния которой узлы - это состояния, а ребра - переходы из состояния в состояние (рис. 3.14).

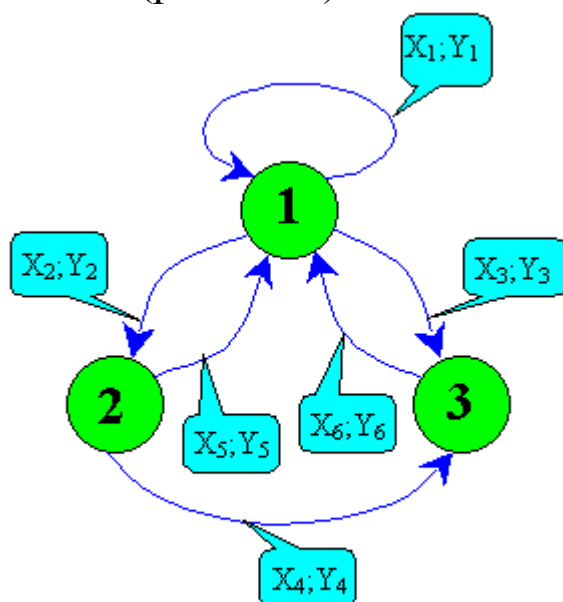


Рис. 3.14.

Входной сигнал в систему это  $X$ . Реакцией на входной сигнал является выработка выходного сигнала  $Y$  и переход в соответствующее состояние.

Граф состояния дискретной системы можно представить в виде комбинации одного двусвязного и трех односвязных списков, которые вместе составляют нелинейный двусвязный список. При этом в информационных полях должна записываться информация о состояниях системы и ребрах. Указатели элементов должны формировать логические ребра системы (рис. 3.15).

Для реализации вышесказанного:

- 1) должен быть создан список, отображающий состояния системы (1, 2, 3);
- 2) должны быть созданы списки, отображающие переходы по ребрам из соответствующих состояний.



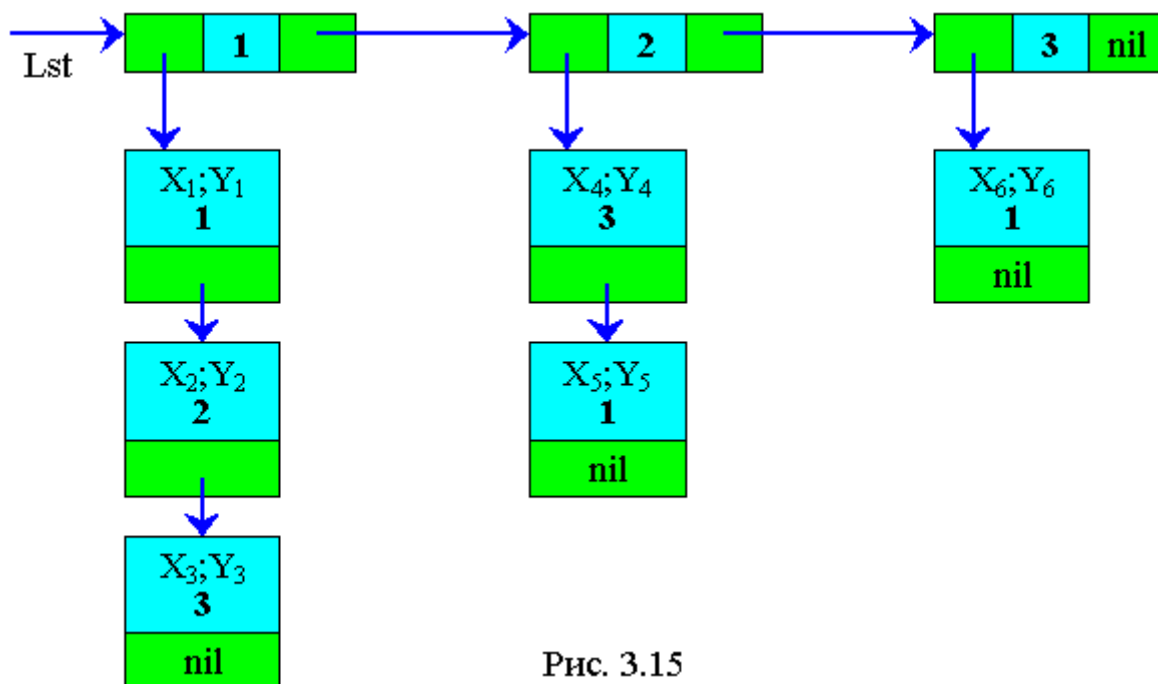


Рис. 3.15

В общем случае при реализации многосвязной структуры получается сеть.

### Контрольные вопросы

1. Какие динамические структуры Вам известны?
2. В чем отличительная особенность динамических объектов?
3. Какой тип представлен для работы с динамическими объектами?
4. Как связаны элементы в динамической структуре?
5. Назовите основные особенности односвязного списка?
6. В чем отличие линейных списков от кольцевых?
7. Зачем были введены двусвязные списки?
8. В чем разница в операциях, производимых над односвязными и двусвязными списками?
9. Какой список является более удобным в обращении, односвязный или двусвязный?
10. Что такое указатель?
11. Какие стековые операции можно производить над списками?

12. Какие операции, производимые над очередью, можно производить над списками?
13. Почему можно производить все эти операции над списками?
14. Для чего предназначены операции Getnode и Freenode?
15. Какие методы утилизации вы знаете?
16. Перечислите элементы заголовков в списках.
17. Зависит ли время, затраченное на вставку элемента в односвязный список, от количества элементов в списке?
18. Где процесс вставки и удаления эффективнее, в списке или в массиве?
19. Как можно производить просмотр односвязного списка?
20. Что означает AVAIL?
21. Какой недостаток односвязных списков по сравнению с
22. массивом?
23. Какие структуры являются нелинейными?
24. Каковы признаки отличия нелинейных структур?
25. Как можно создать нелинейную связную структуру?
26. Что такое граф состояния?

## 4. РЕКУРСИВНЫЕ СТРУКТУРЫ ДАННЫХ

Рассмотрим рекурсивные алгоритмы и рекурсивные структуры данных.

Рекурсия - процесс, протекание которого связано с обращением к самому себе (к этому же процессу).

Пример рекурсивной структуры данных - структура данных, элементы которой являются такими же структурами данных (рис. 4.1).

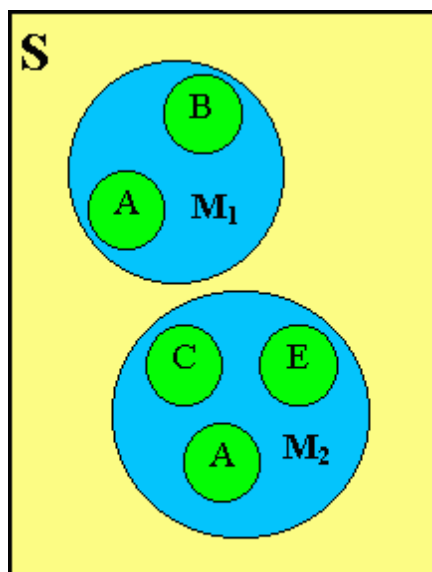


Рис. 4.1

### 4.1 Деревья

Дерево - нелинейная связанная структура данных (рис. 4.2).

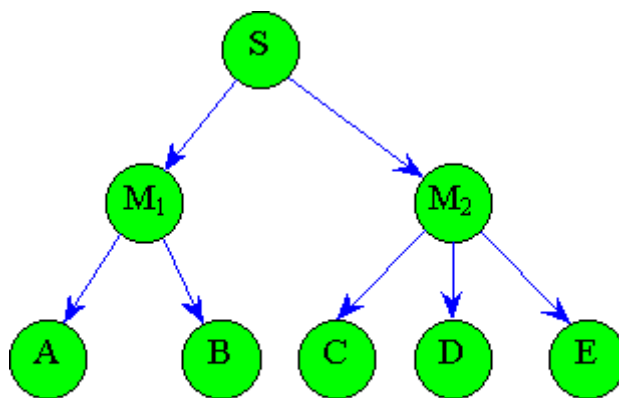


Рис. 4.2

Дерево характеризуется следующими признаками:

- дерево имеет 1 элемент, на которого нет ссылок от других элементов. Этот элемент называется **корнем** дерева;
- в дереве можно обратиться к любому элементу путем прохождения конечного числа **ссылок** (указателей);
- каждый элемент дерева связан **только с одним** предыдущим элементом.

Любой узел дерева может быть **промежуточным** либо **терминальным (листом)**.

На рис. 4.2 промежуточными являются элементы M1, M2, листьями - A, B, C, D, E. Характерной особенностью терминального узла является отсутствие ветвей.

**Высота** дерева - это количество уровней дерева. У дерева на рис. 4.2 высота равна двум.

Количество ветвей, растущих из узла дерева, называется **степенью исхода** узла (на рис. 4.2 для M1 степень исхода 2, для M2 - 3).

Для описания связей между узлами дерева применяют также следующую терминологию: M1 - “**отец**” для элементов A и B. A и B - “**сыновья**” узла M1.

Деревья могут классифицироваться по степени исхода:

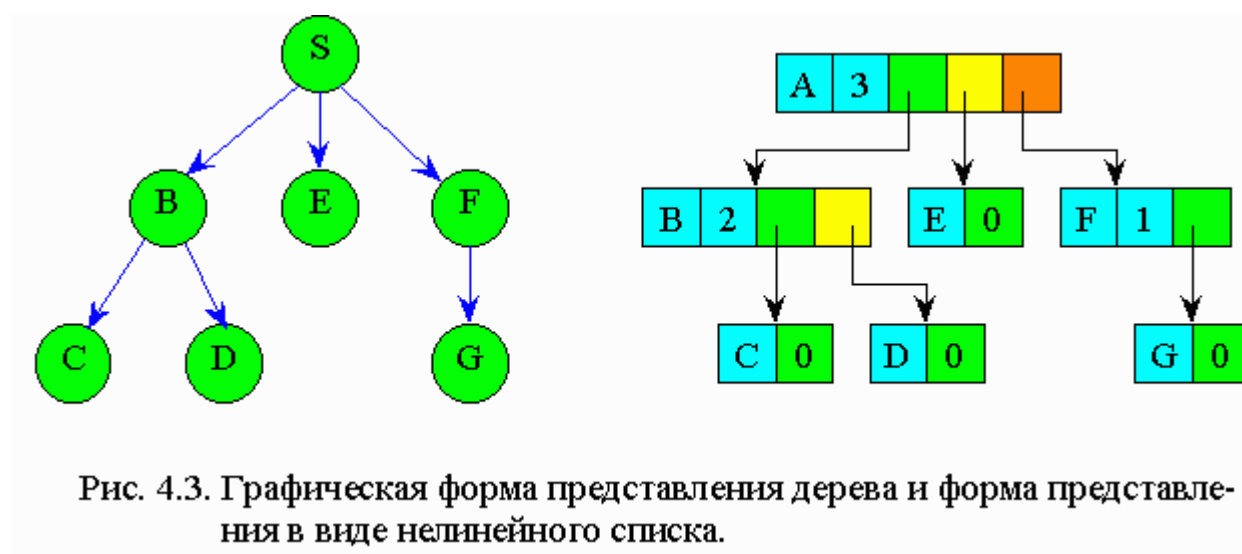
- 1) если максимальная степень исхода равна  $m$ , то это -  $m$ -арное дерево;

2) если степень исхода равна либо 0, либо  $m$ , то это - полное  $m$ -арное дерево;

3) если максимальная степень исхода равна 2, то это - бинарное дерево;

4) если степень исхода равна либо 0, либо 2, то это - полное бинарное дерево.

#### 4.1.1 Представление деревьев



Наиболее удобно деревья представлять в памяти ЭВМ в виде связанных списков. Элемент списка должен содержать информационные поля, в которых содержится значение ключа и другая хранимая информация, а также - поля-указатели, число которых равно степени исхода (рис.4.3). Любой указатель элемента ориентирует данный элемент-узел с сыновьями этого узла.

#### 4.2 Бинарные деревья

Бинарные деревья являются наиболее используемой разновидностью деревьев.

Согласно представлению деревьев в памяти ЭВМ каждый элемент будет записью, содержащей 4 поля. Значения этих полей будут соответственно ключ записи, текст записи, ссылка на элемент влево-вниз, на элемент вправо-вниз.

Для создания элемента бинарного дерева надо создавать в памяти элементы следующего формата (рис. 4.5):



Рис. 4.5

Функция  $V = \text{MakeTree}(\text{Key}, \text{Rec})$  - создает элемент (узел дерева) с двумя указателями и двумя полями (ключевым и информационным).

Алгоритм функции следующий:

```

p = getnode
r(p) = rec
k(p) = key
v = p
left(v) = nil
right(v) = nil
return
    
```

При построении необходимо помнить, что левый сын имеет ключ меньший, чем у отца, а значение ключа правого сына больше значения ключа отца. Например, построим бинарное дерево из следующих элементов: 50, 46, 61, 48, 29, 55, 79. Оно имеет следующий вид:

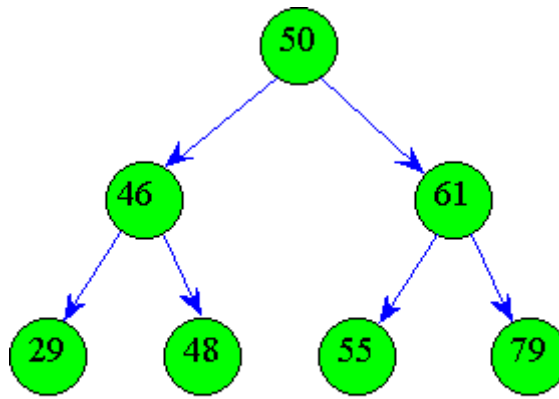


Рис. 4.4 .

Получили упорядоченное бинарное дерево с одинаковым числом уровней в левом и правом поддеревьях. Это - идеально сбалансированное дерево, то есть дерево, в котором левое и правое поддеревья имеют уровни, отличающиеся не более чем на единицу.

#### 4.2.1 Сведение m-арного дерева к бинарному

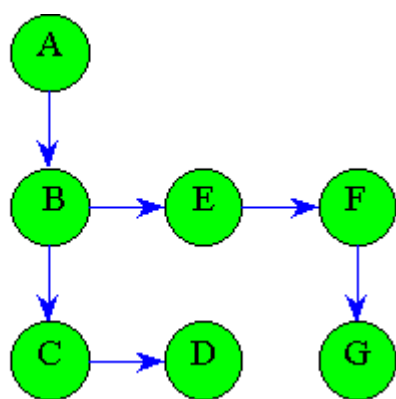
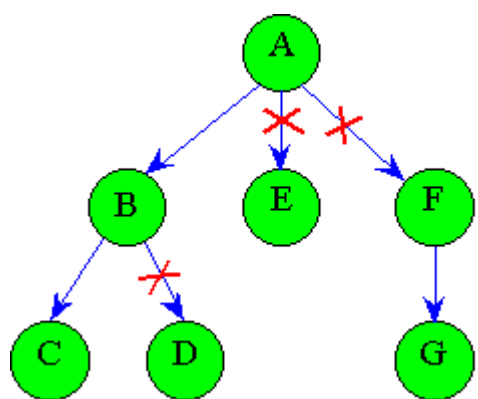
Неформальный алгоритм:

1. В любом узле дерева отсекаются все ветви, кроме крайней **левой**, соответствующей старшим сыновьям.

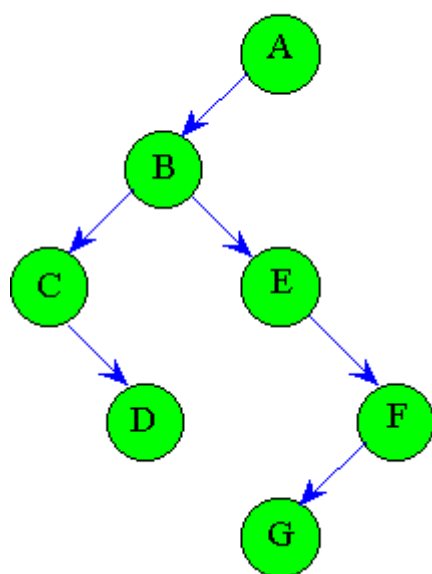
2. Соединяется горизонтальными линиями все сыновья **одного** родителя.

3. Старшим (левым) сыном в любом узле полученной структуры будет узел, находящийся **под** данным узлом (если он есть).

Последовательность действий алгоритма приведена на рисунках ниже, а рис. 4.6 демонстрирует полученное бинарное дерево с полями ключей и указателей.



ИЛИ





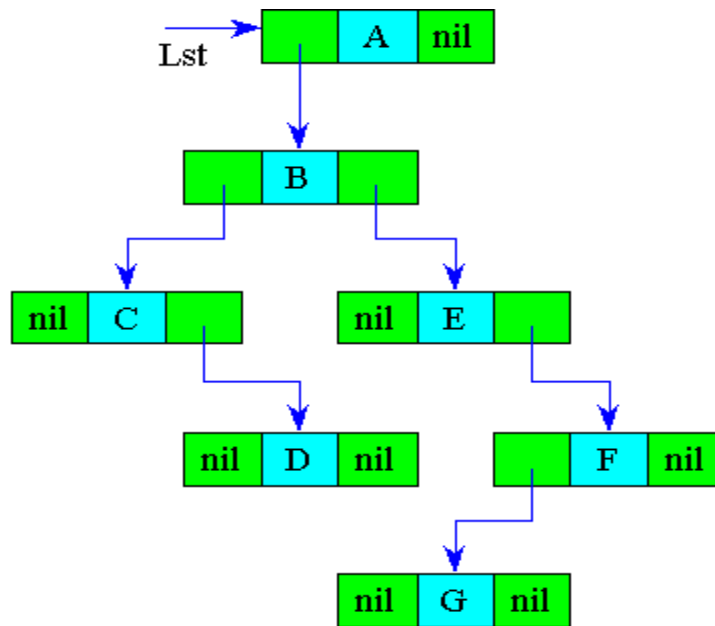


Рис. 4.6. Сведение  $m$ -арного дерева к бинарному.

## 4.2.2 Основные операции с деревьями

1. Обход дерева.
2. Удаление поддерева.
3. Вставка поддерева.

Для выполнения *обхода* дерева необходимо выполнить три операции:

1. Обработка корня.
2. Обработка левой ветви.
3. Обработка правой ветви.

В зависимости от того, в какой последовательности выполняются эти три операции, различают три вида обхода.

1. Обход сверху вниз. Операции выполняются в последовательности

1- 2 - 3.

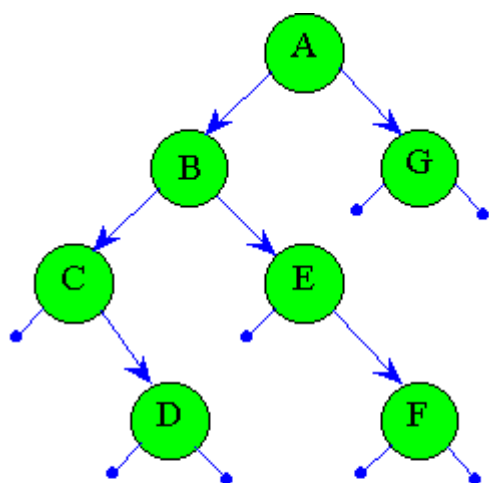
2. Обход слева направо. Операции выполняются в последовательности

2 - 1- 3.

3.Обход снизу вверх. Операции выполняются в последовательности

2 - 3 -1.

В зависимости от того, какой по счету заход в узел приводит к обработке узла, получается реализация одного из трех видов обхода. Если обработка идет после первого захода в узел, то сверху вниз, если после второго, то слева направо, если после третьего, то снизу вверх (см. рис. 4. 7).



Порядок обхода дерева:  
A, B, C, D, E, F, G - сверху вниз;  
C, D, B, E, F, A, G - слева направо;  
D, C, F, E, B, G, A - снизу вверх.

Точно так же можно совершить обход , начав с отростка дерева .

Рис. 4.7. Различные направления обхода дерева.

*Операция исключения поддеревя.* Необходимо указать узел, к которому подсоединяется исключаемое поддерево и указатель этого поддерева. Исключение поддерева состоит в том, что разрывается связь с исключаемым поддеревом, т. е. указатель элемента, связанного с узлом-корнем удаляемого поддерева, устанавливается в nil, а степень исхода данного узла уменьшается на единицу.

*Вставка поддерева* - операция, обратная исключению или удалению. Надо знать указатель корня включаемого поддерева и узел, к которому подвешивается поддерево. Устано-

вить указатель этого узла на корень поддерева, а степень исхода данного узла увеличить на единицу. При этом, в общем случае, необходимо произвести перенумерацию сыновей узла, к которому подвешивается поддерево.

Алгоритм вставки и удаления рассмотрен в главе 5.

### 4.2.3 Создание дерева бинарного поиска

Пусть заданы элементы с ключами: 14, 18, 6, 21, 1, 13, 15. После выполнения нижеприведенного алгоритма получится дерево, изображенное на рис.4.6. Если обойти полученное дерево слева направо, то получим упорядочивание: 1, 6, 13, 14, 15, 18, 21.

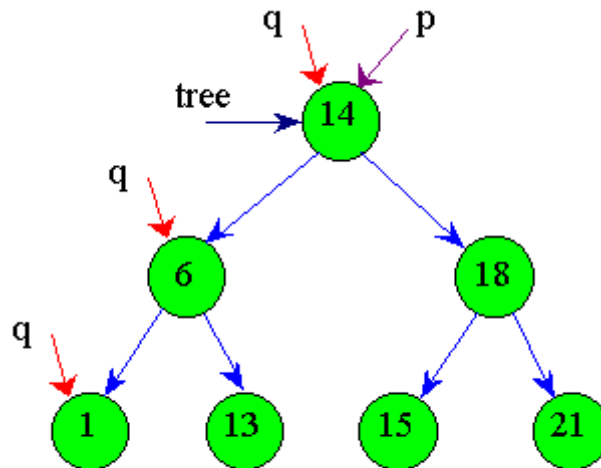


Рис. 4.8.

Алгоритм создания бинарного дерева:

```

read (key, rec)
tree = maketree(key, rec)
while not eof do
    p = tree
    q = tree
    read (key, rec)
    v = maketree(key, rec)
    while p <> nil do

```

```

q = p
if key < k(p) then
    p = left(p)
else
    p = right(p)
endif
endwhile
if key < k(q) then
    left(q) = v
else
    right(q) = v
endif
endwhile
return

```

#### 4.2.4 Прохождение (обход) бинарных деревьев

В зависимости от последовательности обхода узлов различают три вида обхода (прохождения) деревьев (рис.4.9):

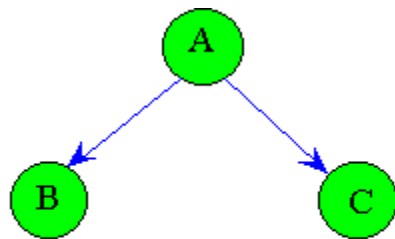


Рис. 4.9. Прохождение бинарных деревьев

1. Сверху вниз **А, В, С**.
2. Слева направо или симметричное прохождение **В, А, С**.
3. Снизу вверх **В, С, А**.

Наиболее часто применяется второй способ.

Ниже приведены рекурсивные алгоритмы прохождения бинарных деревьев.

*subroutine pretrave (tree)*

‘сверху вниз’

*if tree <> nil then*

*print info(tree)*

*pretrave(left(tree))*

*pretrave(right(tree))*

*endif*

*return*

*subroutine intrave (tree)*

‘симметричный или слева  
направо’

*if tree <> nil then*

*intrave(left(tree))*

*print info(tree)*

*intrave(right(tree))*

*endif*

*return*

*subroutine postrave (tree)*

‘снизу вверх’

*if tree<>nil then*

*postrave (left(tree))*

*postrave (right(tree))*

*print info(tree)*

*endif*

*return*

### Обход дерева A, B, C

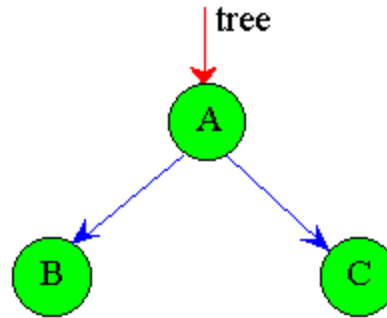


Рис. 4.10

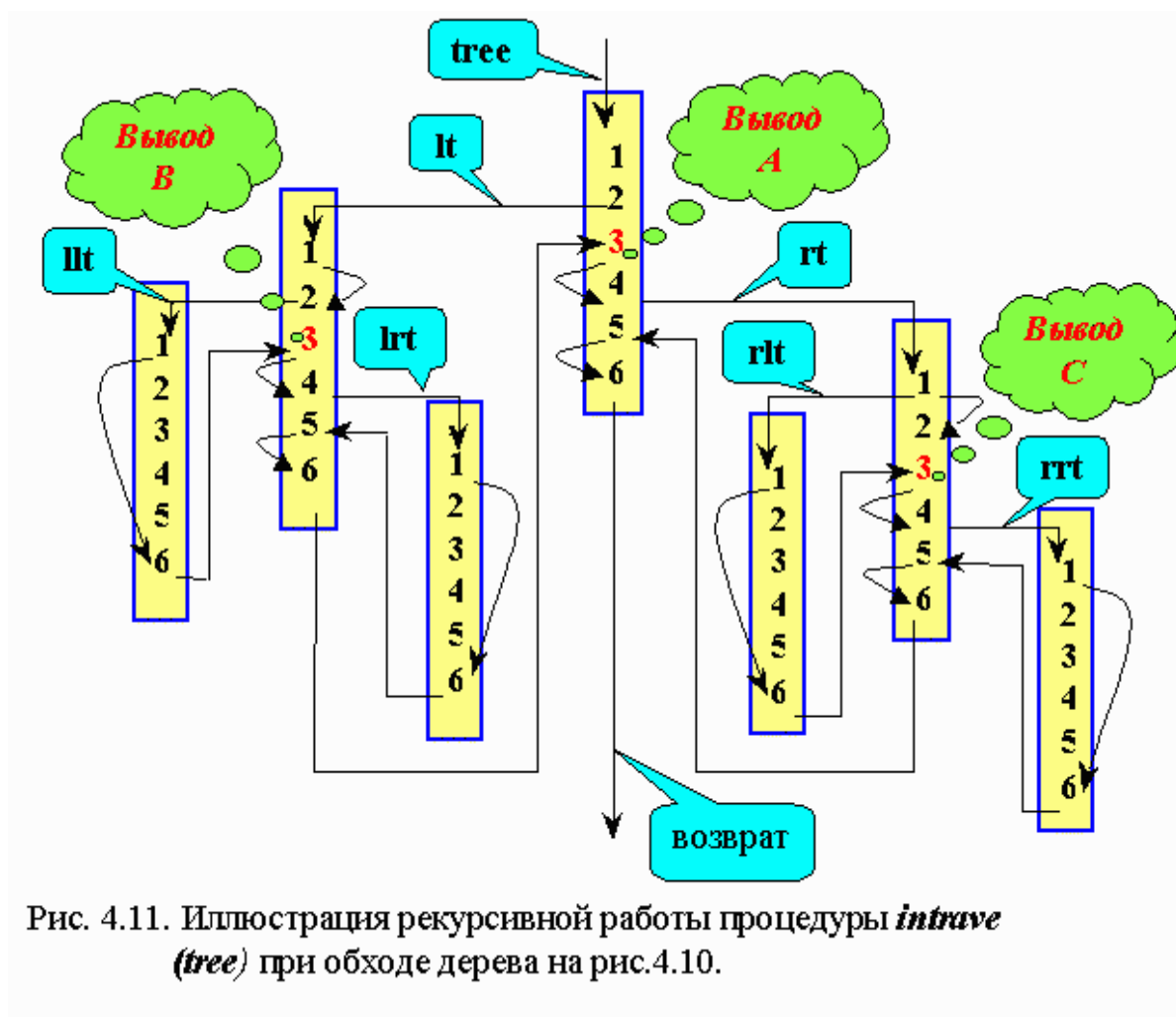
Поясним подробнее рекурсию алгоритма обхода дерева слева направо.

Пронумеруем строки алгоритма *intrave (tree)*:

```
1  if tree <> nil
2      then intrave (left(tree))
3          print info (tree)
4          intrave (right (tree))
5      endif
6  return
```

Обозначим указатели: **t** → **tree**; **l** → **left**; **r** → **right**

На приведенном рис. 4.11 проиллюстрирована последовательность вызова функции *intrave (tree)* по мере обхода узлов простейшего дерева, изображенного на рис. 4. 10.



## 5. ПОИСК

Поиск является одной из основных операций при обработке информации в ЭВМ. Ее назначение - по заданному аргументу найти среди массива данных те данные, которые соответствуют этому аргументу.

Набор данных (любых) будем называть таблицей или файлом. Любое данное (или элемент структуры) отличается каким-то признаком от других данных. Этот признак называется ключом. Ключ может быть уникальным, т. е. в таблице существует только одно данное с этим ключом. Такой уникальный ключ называется первичным. Вторичный ключ в одной таблице может повторяться, но по нему тоже можно организовать поиск. Ключи данных могут быть собраны в одном месте (в другой таблице) или представлять собой запись, в которой одно из полей - это ключ. Ключи, которые выделены из таблицы данных и организованы в свой файл, называются внешними ключами. Если ключ находится в записи, которую он определяет, то он называется внутренним.

Поиском по заданному аргументу называется алгоритм, определяющий соответствие ключа заданному аргументу. Результатом работы алгоритма поиска может быть нахождение этого ключа или отсутствие его в таблице. В случае отсутствия ключа (и информации, ему соответствующей) возможны две операции:

- индикация того, что ключа нет;
- вставка ключа (и информации) в таблицу.

В случае нахождения ключа (и информации ему соответствующей) возможны две операции:

- обработка найденной информации;
- удаление найденной информации.

Пусть  $k$  - массив ключей. Для каждого  $k(i)$  существует  $r(i)$  - данное.  $Key$  - аргумент поиска. Ему соответствует ин-



формационная запись гес. В зависимости от того, какова структура данных в таблице, различают несколько видов поиска.

## 5.1 Последовательный поиск

Применяется в том случае, если неизвестна организация данных или данные неупорядочены. Тогда производится последовательный просмотр по всей таблице начиная от младшего адреса в оперативной памяти и кончая самым старшим.

Ниже представлен алгоритм последовательного поиска в массиве (переменная *search* хранит номер найденного элемента).

```

for i = 1 to n
    if k(i) = key then
        search = i
        return
    endif
next i
search = 0
return

```

i	k	r
1	8	...
2	136	...
3	4	...
...	...	...
n-1	17	...
n	234	...

Рис. 5.1

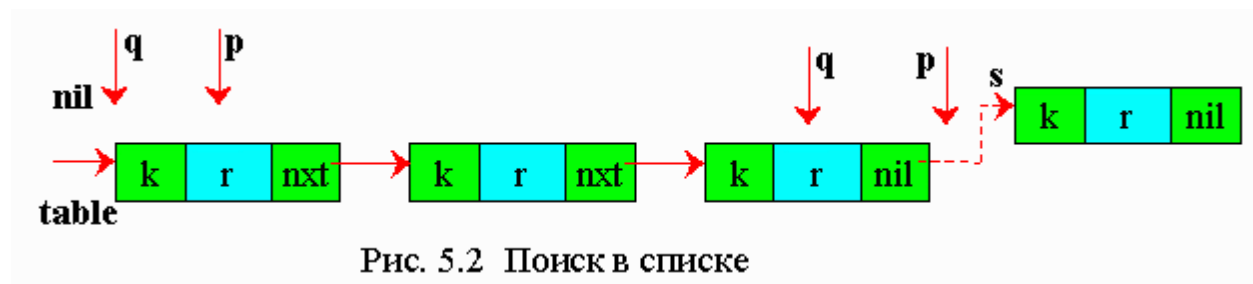
Если элемент не найден в таблице и необходимо произвести вставку, то последние 2 оператора заменяются на

```

n = n + 1
k(n) = key
r(n) = rec
search = n
return

```

Если таблица данных задана в виде односвязного списка, то производится последовательный поиск в списке (рис. 5.2).



Ниже представлен алгоритм последовательного поиска в списке.

```

q = nil
p = table
while (p <> nil) do
    if k(p) = key then
        search = p
        return
    endif
    q = p
    p = nxt(p)
endwhile
s = getnode
k(s) = key
r(s) = rec

```

```

nxt(s) = nil
if q = nil then
    table = s
else
    nxt(q) = s
endif
search = s
return

```

Достоинством списковой структуры является ускоренный алгоритм удаления или вставки элемента в список, причем время вставки или удаления не зависит от количества элементов, а в массиве каждая вставка или удаление требуют передвижения примерно половины элементов. Эффективность поиска в списке примерно такая же, как и в массиве.

Эффективность любого поиска может оцениваться по количеству сравнений  $C$  аргумента поиска с ключами таблицы данных. Чем меньше количество сравнений, тем эффективнее алгоритм поиска.

Эффективность последовательного поиска в *массиве*

$$C_{min} = 1, C_{max} = n.$$

Если данные расположены равновероятно во всех ячейках массива, то

$$C_{cp} \approx (n + 1)/2.$$

Эффективность последовательного поиска в *списке* - то же самое.

Порядок эффективности последовательного поиска  $O(n)$  (величина одного порядка с  $n$ ).

Эффективность последовательного поиска можно увеличить.

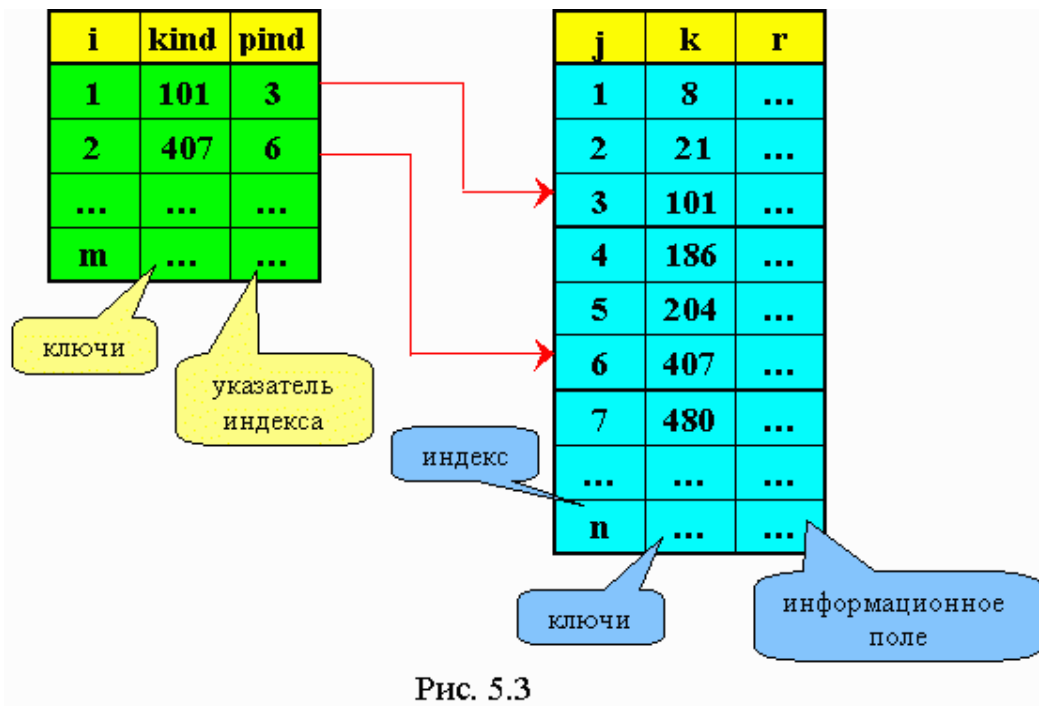
## 5.2 Индексно-последовательный поиск

Пусть имеется возможность накапливать запросы за день, а ночью их обрабатывать. Когда запросы собраны, происходит их сортировка.

При таком поиске организуется две таблицы: таблица данных со своими ключами (**упорядоченными по возрастанию**) и таблица **индексов**, которая тоже состоит из ключей данных, но эти ключи взяты из основной таблицы через определенный интервал (рис. 5.3).

Сначала производится последовательный поиск в таблице индексов по заданному аргументу поиска. Как только мы проходим ключ, который оказался меньше заданного, то этим мы устанавливаем нижнюю границу поиска в основной таблице -  $low$ , а затем - верхнюю -  $hi$ , на которой ( $kind > key$ ).

Например, интервал равен 3, а  $key=186$ , тогда в таблицах ключей и индексов на рисунке ниже  $low=101$ ,  $hi=407$ . Поиск идет вначале с шагом 3 относительно основной таблицы, затем после установления верхней и нижней границ с шагом 1 не по всей таблице, а от  $low$  до  $hi$ .



Алгоритм индексно-последовательного поиска:

```

i = 1
while (i ≤ m) and (kind(i) ≤ key) do
    i = i + 1
endwhile
if i = 1 then low = 1
    else low = pind(i-1)
endif
if i = m + 1 then hi = n
    else hi = pind(i) - 1
endif
for j = low to hi
    if key = k(j) then
        search = j
        return
    endif
endfor
search = 0
return
  
```

### 5.3 Эффективность последовательного поиска

Эффективность любого поиска может оцениваться по количеству сравнений  $C$  аргумента поиска с ключами таблицы данных. Чем меньше количество сравнений, тем эффективнее алгоритм поиска.

Эффективность последовательного поиска в массиве:

$$C = 1 \div n, C = (n + 1)/2.$$

Эффективность последовательного поиска в списке - то же самое. Хотя по количеству сравнений поиск в списке и в массиве имеет одинаковую эффективность, организация данных в виде массива и списка имеет свои достоинства и недостатки. Целью поиска является выполнение следующих операций:

- 1) Найденную запись считать.
- 2) При отсутствии записи произвести ее вставку в таблицу.
- 3) Найденную запись удалить.

Первая операция (собственно поиск) занимает для них одно время. Вторая и третья операции для списочной структуры более эффективна (т. к. у массивов надо сдвигать элементы).

Если  $k$  - число передвижений элементов в массиве, то  $k = (n + 1)/2$ .

### 5.4 Эффективность индексно-последовательного поиска

Если считать равновероятным появление всех случаев, то эффективность поиска можно рассчитать следующим образом:

Введем обозначения:

$n$  – размер основной таблицы;

$m$  - размер индексной таблицы;

$p$  - размер шага;

$m = n / p$ ;

Тогда количество сравнений  $C$  считается так:

$$C = (m+1)/2 + (p+1)/2 = (n/p+1)/2 + (p+1)/2 = n/2p + p/2 + 1$$

Продифференцируем  $C$  по  $p$  и приравняем производную нулю:

$$dC/dp = (d/dp) (n/2p + p/2 + 1) = -n/2p^2 + 1/2 = 0$$

Отсюда

$$p^2 = n;$$

$$p_{opt} = \sqrt{n}$$

Подставив  $p_{opt}$  в выражение для  $C$ , получим следующее количество сравнений:

$$C = \sqrt{n} + 1$$

Порядок эффективности индексно-последовательного поиска  $O(\sqrt{n})$ .

## 5.5 Методы оптимизации поиска

Всегда можно говорить о некотором значении вероятности поиска того или иного элемента в таблице. Считаем, что в таблице данный элемент существует. Тогда вся таблица поиска может быть представлена как система с дискретными состояниями, а вероятность нахождения там искомого элемента - это вероятность  $p(i)$   $i$ -го состояния системы.

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

Количество сравнений при поиске в таблице, представленной как дискретная система, представляет собой математическое ожидание значения дискретной случайной величины, определяемой вероятностями состояний и номерами состояний системы.

$$Z=C=1p(1)+2p(2)+3p(3)+...+np(n)$$

Желательно, чтобы  $p(1) \geq p(2) \geq p(3) \geq ... \geq p(n)$ .

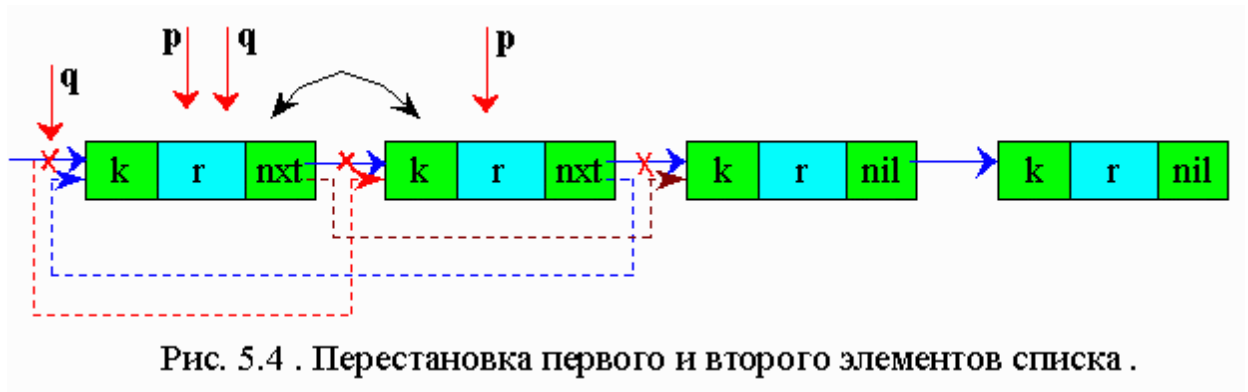
Это минимизирует количество сравнений, то есть увеличивает эффективность. Так как последовательный поиск начинается с первого элемента, то на это место надо поставить элемент, к которому чаще всего обращаются (с наибольшей вероятностью поиска).

Наиболее часто используются два основных способа автономного переупорядочивания таблиц поиска. Рассмотрим их на примере односвязных списков.

### **5.5.1 Переупорядочивание таблицы поиска путем перестановки найденного элемента в начало списка**

Суть этого метода заключается в том, что элемент списка с ключом, равным аргументу поиска, становится первым элементом списка, исходя из предположения, что к этому элементу будут обращаться чаще всего.





Этот алгоритм применим как для списков, так и для массивов. Однако не рекомендуется применять его для массивов, так как на перестановку элементов массива затрачивается гораздо больше времени, чем на перестановку указателей.

Алгоритм переупорядочивания в начало списка:

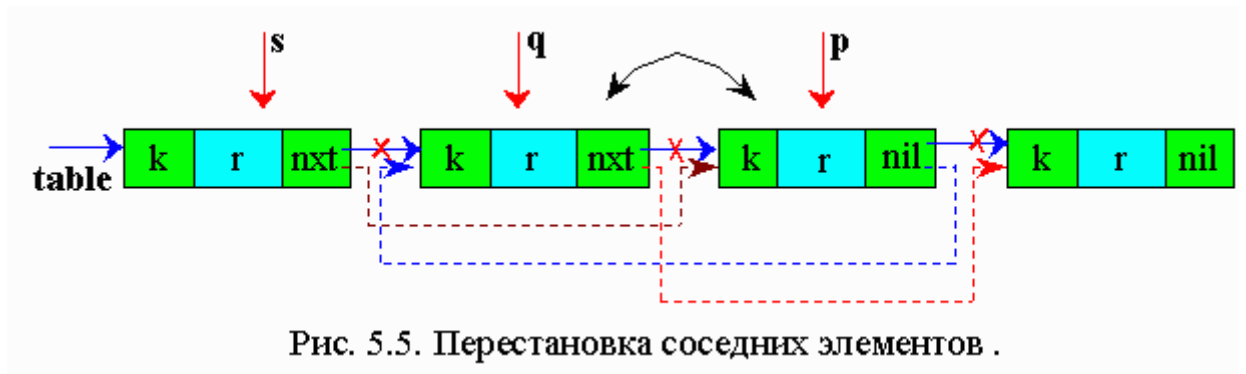
```

q = nil
p = table
while (p <> nil) do
  if key = k(p) then
    search = p
    if q = nil
      then 'перестановка не нужна'
      return
    endif
    nxt(q) = nxt(p)
    nxt(p) = table
    table = p
    return
  endif
  q = p
  p = nxt(p)
endwhile
search = 0
return

```

### 5.5.2 Метод транспозиции

В данном методе найденный элемент переставляется на один элемент к голове списка. И если к этому элементу обращаются часто, то, перемещаясь к голове списка, он скоро окажется на первом месте.



$p$  - рабочий указатель

$q$  - вспомогательный указатель, отстает на один шаг от  $p$

$s$  - вспомогательный указатель, отстает на два шага от  $q$

Алгоритм метода транспозиции:

```

s = nil
q = nil
p = table
while (p <> nil) do
    if key = k(p) then
        ' нашли, транспонируем
        if q = nil then
            ' переставлять не надо
            return
        endif
        nxt(q) = nxt(p)
        nxt(p) = q
        if s = nil then
            table = p
        else

```

```

        nxt(s) = p
        search = p
    endif
    return
endif
s = q
q = p
p = nxt(p)
endwhile
search = nil
return

```

Этот метод удобен при поиске не только в списках, но и в массивах (так как меняются только два стоящих рядом элемента).

### 5.5.3 Дерево оптимального поиска

Если извлекаемые элементы сформировали некоторое постоянное множество, то может быть выгодным настроить дерево бинарного поиска для большей эффективности последующего поиска.

Рассмотрим деревья бинарного поиска, приведенные на рисунках 5.6а и 5.6б. Оба дерева содержат три элемента -  $K1$ ,  $K2$ ,  $K3$ , где  $K1 < K2 < K3$ . Поиск элемента  $K3$  требует двух сравнений для рисунка 5.6 а), и только одного - для рисунка 5.6 б).

Число сравнений ключей, которые необходимо сделать для извлечения некоторой записи, равно уровню этой записи в дереве бинарного поиска плюс 1.

Предположим, что:

- $p1$  - вероятность того, что аргумент поиска  $key = K1$
- $p2$  - вероятность того, что аргумент поиска  $key = K2$
- $p3$  - вероятность того, что аргумент поиска  $key = K3$
- $q0$  - вероятность того, что  $key < K1$

$q1$  - вероятность того, что  $K2 > key > K1$

$q2$  - вероятность того, что  $K3 > key > K2$

$q3$  - вероятность того, что  $key > K3$

$C1$  - число сравнений в первом дереве рисунка 5.6 а)

$C2$  - число сравнений во втором дереве рисунка 5.6 б)

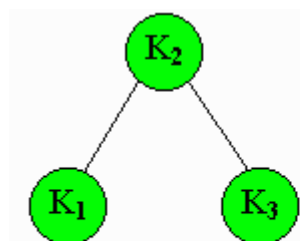


Рис. 5.6 а)

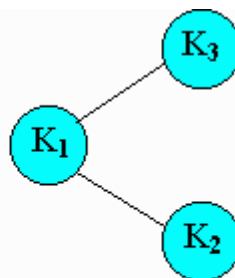


Рис. 5.6 б)

Тогда  $p1+p2+p3+q0+q1+q2+q3 = 1$

Ожидаемое число сравнений в некотором поиске есть сумма произведений вероятности того, что данный аргумент имеет некоторое заданное значение, на число сравнений, необходимых для извлечения этого значения, где сумма берется по всем возможным значениям аргумента поиска. Поэтому

$$C1 = 2p1 + 1p2 + 2p3 + 2q0 + 2q1 + 2q2 + 2q3$$

$$C2 = 2p1 + 3p2 + 1p3 + 2q0 + 2q1 + 3q2 + 1q3$$

Это ожидаемое число сравнений может быть использовано как некоторая мера того, насколько "хорошо" конкретное дерево бинарного поиска подходит для некоторого данного множества ключей и некоторого заданного множества вероятностей. Так, для вероятностей, приведенных далее слева, дерево из а) является более эффективным, а для вероятностей, приведенных справа, дерево из б) является более эффективным:

$$P1 = 0.1$$

$$P2 = 0.3$$

$$P3 = 0.1$$

$$P1 = 0.1$$

$$P2 = 0.1$$

$$P3 = 0.3$$

$$q0 = 0.1$$

$$q1 = 0.2$$

$$q2 = 0.1$$

$$q3 = 0.1$$

$$q0 = 0.1$$

$$q1 = 0.1$$

$$q2 = 0.1$$

$$q3 = 0.2$$

Тогда

$$C1 = 1.6$$

$$C2 = 2.2$$

$$C1 = 1.8$$

$$C2 = 1.7$$

Дерево бинарного поиска, которое минимизирует ожидаемое число сравнений некоторого заданного множества ключей и вероятностей, называется оптимальным. Хотя алгоритм создания дерева может быть очень трудоемким, дерево, которое он создает, будет работать эффективно во всех последующих поисках. К сожалению, однако, заранее вероятности аргументов поиска редко известны.

## 5.6 Бинарный поиск (метод деления пополам)

Будем предполагать, что имеем упорядоченный по возрастанию массив чисел. Основная идея - выбрать случайно некоторый элемент  $A_M$  и сравнить его с аргументом поиска  $X$ . Если  $A_M = X$ , то поиск закончен; если  $A_M < X$ , то мы заключаем, что все элементы с индексами, меньшими или равными  $M$ , можно исключить из дальнейшего поиска. Аналогично, если  $A_M > X$ .

Выбор  $M$  совершенно произволен в том смысле, что корректность алгоритма от него не зависит. Однако на его эффективность выбор влияет. Ясно, что наша задача - исключить как можно больше элементов из дальнейшего поиска. Оптимальным решением будет выбор среднего элемента, т.е. середины массива.

Рассмотрим пример, представленный на рис. 5.7. Допустим нам необходимо найти элемент с ключом 52. Первым сравниваемым элементом будет 49. Так как  $49 < 52$ , то ищем

следующую середину среди элементов, расположенных выше 49. Это число 86.  $86 > 52$ , поэтому теперь ищем 52 среди элементов, расположенных ниже 86, но выше 49. На следующем шаге обнаруживаем, что очередное значение середины равно 52. Мы нашли элемент в массиве с заданным ключом.

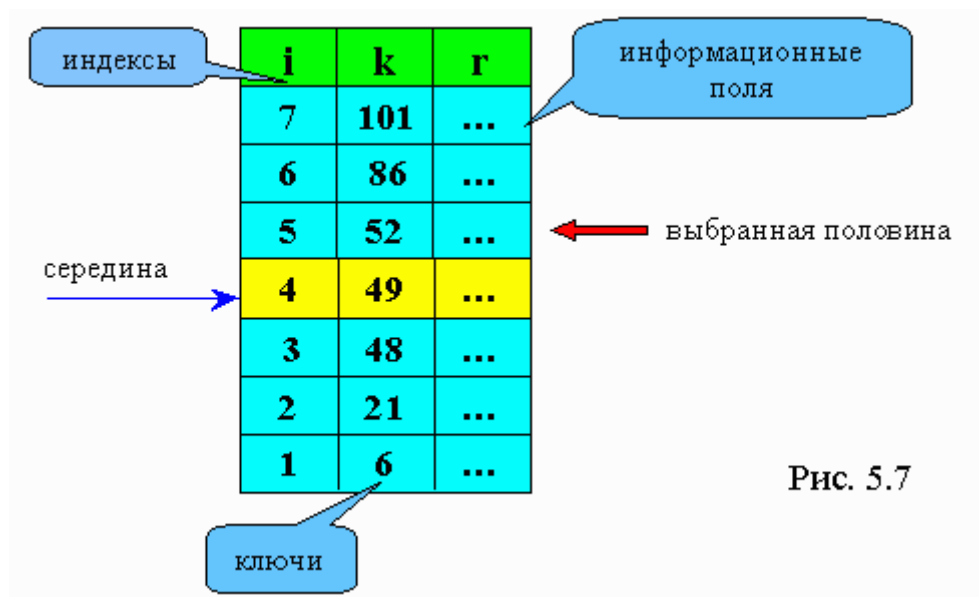


Рис. 5.7

Алгоритм бинарного поиска:

```

low = 1
hi = n
while (low <= hi) do
    mid = (low + hi) div 2
    if key = k(mid) then
        search = mid
        return
    endif
    if key < k(mid) then
        hi = mid - 1
    
```

```

        else
        low = mid + 1
    endif
endwhile
search = 0
return

```

Использование данного алгоритма позволяет найти запись с ключом, равным, например, 101 в таблице поиска рис. 5.7, за три сравнения (в последовательном поиске понадобилось бы семь сравнений).

Если  $C$  - количество сравнений, а  $n$  - число элементов в таблице, то

$C = \log_2 n$ , следовательно, эффективность бинарного поиска имеет порядок

$$O(\log_2 N)$$

Например,  $n = 1024$ .

При последовательном поиске  $C = 512$ , а при бинарном  $C = 10$ .

Можно совместить бинарный и индексно-последовательный поиск (при больших объемах данных), учитывая, что последний также используется при отсортированном массиве.

## 5.7 Поиск по бинарному дереву

Назначение его в том, чтобы по заданному ключу осуществить поиск узла дерева. Длительность операции зависит от структуры дерева. Действительно, дерево может быть вы-

рождено в однонаправленный список, как правое на рис. 5.8 справа.

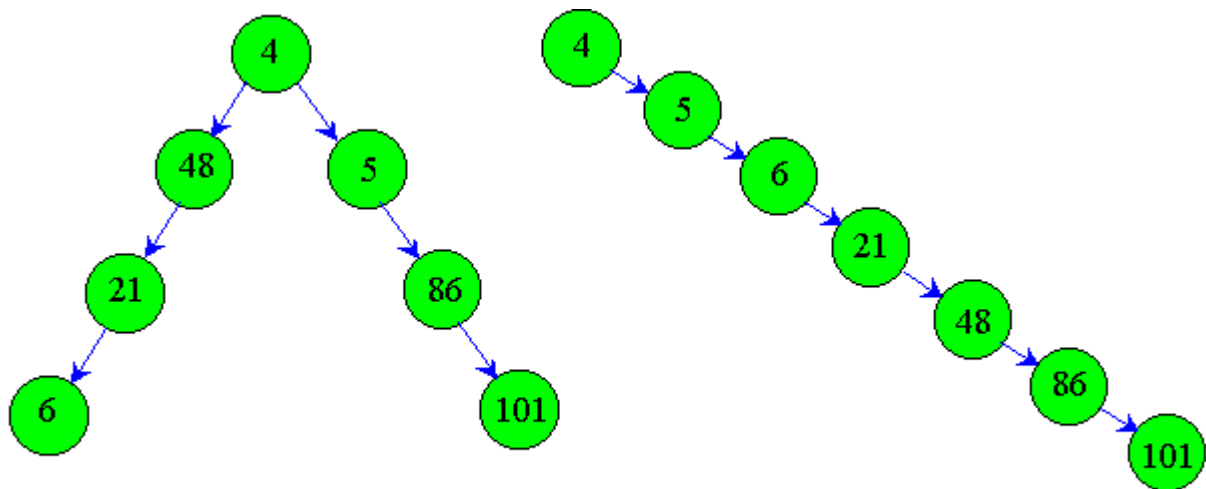


Рис. 5.8. Бинарные деревья.

В этом случае время поиска будет такое же, как и в однонаправленном списке, т.е. придется перебирать  $N/2$  элементов.

Наибольшего эффекта использования дерева достигается в том случае, когда дерево сбалансировано. В этом случае для поиска придется перебрать не больше  $\log_2 N$  элементов. В этом случае эффективность поиска по бинарному дереву имеет порядок  $O(\log_2 N)$ .

Поиск элемента в бинарном дереве называется бинарным поиском по дереву.

Такое дерево называют деревом бинарного поиска.

Суть поиска заключается в следующем. Анализируем вершину очередного поддерева. Если ключ меньше информационного поля вершины, то анализируем левое поддерево, больше - правое.

Алгоритм поиска по бинарному дереву:

```

p = tree
while p <> nil do
    if key = k(p) then
        search = p

```



```

                                return
endif
if key < k(p) then
    p = left(p)
else
    p = right(p)
endif
endwhile
search = nil
return

```

Очень часто следствием поиска являются ситуации:

если узел с заданным ключом не **найден**, то его надо **вставить**

если узел с заданным ключом **найден**, то его надо **удалить**.

## 5.8 Поиск со вставкой (с включением)

Для вставки элемента в дерево, сначала нужно осуществить поиск в дереве по заданному ключу. Если такой ключ имеется, то программа завершается, если нет, то происходит вставка.

Для включения новой записи в дерево, прежде всего, нужно найти тот узел, к которому можно присоединить новый элемент, не нарушив упорядоченности дерева. Алгоритм поиска нужного узла тот же самый, что и при поиске узла с заданным ключом. Однако непосредственно использовать функцию поиска нельзя, потому что по ее окончании она не фиксирует ссылку на узел, после которого поиск прекращается.

Модифицируем функцию поиска так, чтобы фиксировалась ссылка на узел, в котором был найден заданный ключ (в случае успешного поиска), или ссылка на узел, после обработки которого поиск прекращается (в случае неуспешного поиска).

Модифицированный алгоритм следующий:

```
p = tree
q = nil
while p <> nil do
    q = p
    if key = k(p) then
        search = p
        return
    endif
    if key < k(p) then
        p = left(p)
    else
        p = right(p)
    endif
endwhile
v = maketree(key, rec)
if q = nil then
    tree = v
else
    if key < k(q) then
        left(q) = v
    else
        right(q) = v
    endif
endif
search = v
return
```

## 5.9 Поиск с удалением

Удаление узла не должно нарушить упорядоченности дерева. Возможны три варианта:

1) Найденный узел является листом. Тогда он просто удаляется с помощью обычной операции удаления.

2) Найденный узел имеет только одного сына. Тогда сын перемещается на место отца.

3) У удаляемого узла два сына. В этом случае нужно найти подходящее звено поддерева, которое можно было бы вставить на место удаляемого. Такое звено всегда существует. На место отца помещается либо его предшественник при обходе слева направо, либо его преемник при том же виде обхода.

Предшественник - это самый правый элемент левого поддерева (для достижения этого элемента необходимо перейти в следующий узел по левой ветви, а затем двигаться только по правой ветви этого узла до тех пор, пока очередная ссылка не будет равна *nil*).

Преемник - это самый левый элемент правого поддерева (для достижения этого элемента необходимо перейти в следующий узел по правой ветви, а затем двигаться только по левой ветви этого узла до тех пор, пока очередная ссылка не будет равна *nil*).

Предшественником удаляемого узла 12 в дереве на рис. 5.9 будет узел 11 (самый правый узел левого поддерева). Преемником – узел 13 (самый левый узел правого поддерева).

Будем разрабатывать алгоритм для преемника (рис.5.9).

р - рабочий указатель;

q - отстает от р на один шаг;

$v$  - указывает на приемника удаляемого узла;  
 $t$  - отстает от  $v$  на один шаг;  
 $s$  - на один шаг впереди  $v$  (указывает на левого сына или пустое место).

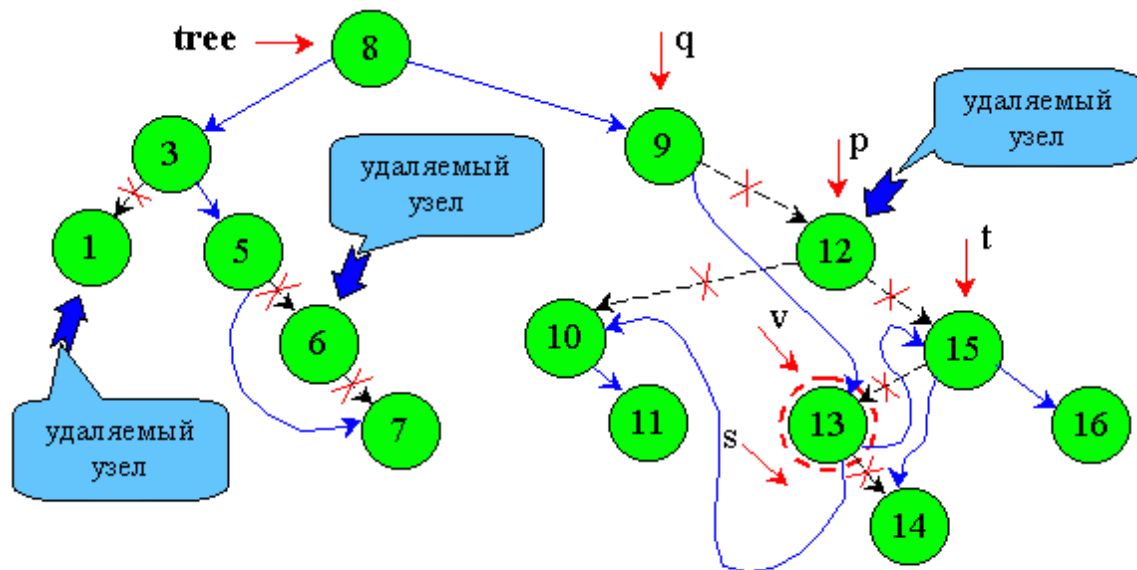


Рис. 5.9. Удаление узлов дерева

На узел 13 в результате поиска приемника должен указывать указатель  $v$ , а указатель  $s$  - на пустое место (как показано на рис. 5.9).

Алгоритм поиска по бинарному дереву с удалением:

```

 $q = nil$ 
 $p = tree$ 
while ( $p \neq nil$ ) and ( $k(p) \neq key$ ) do
   $q = p$ 
  if  $key < k(p)$  then
     $p = left(p)$ 
  else
     $p = right(p)$ 
  endif
endwhile
if  $p = nil$  then 'Ключ не найден'
  return
  
```

```

endif
if left(p) = nil then v = right(p)
    else if right(p) = nil
        then v = left(p)
    else
        'У nod(p) - два сына'
        'Введем два указателя (t отстаёт от v на 1 шаг, s - опере-
жает)'

        t = p
        v = right(p)
        s = left(v)
while s <> nil do
    t = v
    v = s
    s = left(v)
endwhile
if t <> p then
    'v не является сыном p'
    left(t) = right(v)
    right(v) = right(p)
endif
    left(v) = left(p)
endif
endif
if q = nil then 'p - корень'
    tree = v
else if p = left(q)
    then left(q) = v
        else right(q) = v
    endif
endif
endif
freenode(p)
return

```

## Контрольные вопросы

1. В чем состоит назначение поиска?
2. Что такое уникальный ключ?
3. Какая операция производится в случае отсутствия заданного ключа в списке?
4. В чем разница между последовательным и индексно-последовательным поиском?
5. Какой из них более эффективный и почему?
6. Какие способы переупорядочивания таблицы вы знаете?
7. Основные отличия метода перестановки в начало от метода транспозиции?
8. Где они будут работать быстрее, в массиве или списке?
9. В каких списках они работают, упорядоченных или нет?
10. В чем суть бинарного поиска?
11. Как можно обойти бинарное дерево?
12. Можно ли применять бинарный поиск к массивам?
13. Если удалить корень в непустом бинарном дереве, какой элемент станет на его место?

## 6. СОРТИРОВКА

При обработке данных важно знать информационное поле данных и размещение их в машине.

Сортировка - это расположение данных в памяти в регулярном виде по выбранному параметру. Регулярность рассматривают как возрастание (убывание) значения параметра от начала к концу в массиве.

Различают внутреннюю и внешнюю сортировку:

- внутренняя сортировка - сортировка в оперативной памяти;
- внешняя сортировка - сортировка во внешней памяти.

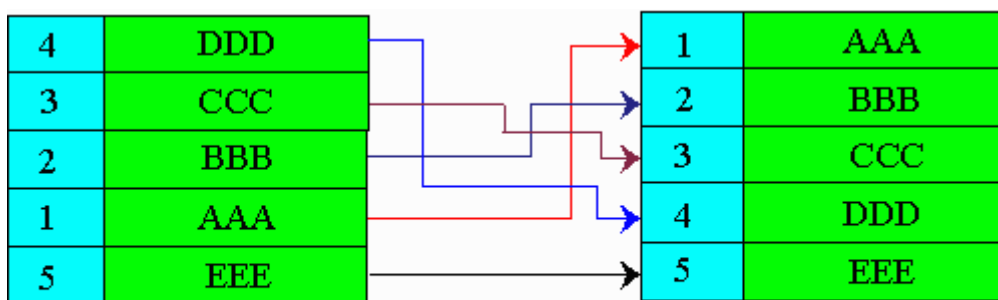


Рис. 6.1. Сортировка.

Если сортируемые записи занимают большой объем памяти, то их перемещение требует больших затрат. Для того, чтобы их уменьшить, сортировку производят в таблице адресов ключей, делают перестановку указателей, т.е. сам массив не перемещается. Это метод сортировки таблицы адресов.

При сортировке могут встретиться одинаковые ключи. В этом случае при сортировке желательно расположить после сортировки одинаковые ключи в том же порядке, что и в исходном файле. Это - устойчивая сортировка.

Эффективность сортировки можно рассматривать по нескольким критериям:

- время, затрачиваемое на сортировку;
- объем оперативной памяти, требуемой для сортировки;

время, затраченное программистом на написание программы.

Выделяем первый критерий, поскольку мы будем рассматривать только методы сортировки «на том же месте», то есть не использующие для процесса сортировки дополнительную оперативную память. Эквивалентом затраченного на сортировку времени можно считать количество сравнений при выполнении сортировки и количество перемещений.

Порядок числа сравнения при сортировке лежит в пределах от  $O(n \log n)$  до  $O(n^2)$ .

$O(n)$  - идеальный и недостижимый случай.

Различают следующие методы сортировки:

строгие (прямые) методы;  
улучшенные методы.

Строгие методы:

- 1) метод прямого включения;
- 2) метод прямого выбора;
- 3) метод прямого обмена.

Эффективность трех строгих методов примерно одинакова.

## **6.1 Сортировка методом прямого включения**

Такой метод широко используется при игре в карты. Элементы мысленно делятся на уже готовую последовательность  $a_1, \dots, a_{i-1}$  и исходную последовательность. При каждом шаге, начиная с  $i = 2$  и увеличивая  $i$  каждый раз на единицу, из исходной последовательности извлекается  $i$ -й элемент и перекладывается в готовую последовательность, при этом он



вставляется на нужное место. На рис. 6.2 показан в качестве примера процесс сортировки с помощью включения шести случайно выбранных чисел. Алгоритм этой сортировки таков:

```

for i = 2 to n
  x = a(i)
  находим место среди a(1)...a(i) для включения x
next i

```



Рис. 6.2.Сортировка методом прямого включения .

Для сортировки методом прямого включения пользуются двумя алгоритмами:

а) Алгоритм сортировки методом прямого включения без барьера:

```

for i = 2 to n
  x = a(i)
  for j = i - 1 downto 1
    if x < a(j)
      then a(j + 1) = a(j)
      else go to L
    endif
  next j
L: a(j + 1) = x

```

```
next i
return
```

Недостатком приведенного алгоритма является нарушение технологии структурного программирования, при которой нежелательно применять безусловные переходы. Если же внутренний цикл организовать как цикл **while**, то необходима постановка «барьера», без которого при отрицательных значениях ключей происходит потеря значимости и «зависание» компьютера.

б) Алгоритм сортировки методом прямого включения с барьером:

```
for i = 2 to n
  x = a(i)
  a(0) = x {a(0) - барьер}
  j = i - 1
  while x < a(j) do
    a(j + 1) = a(j)
    j = j - 1
  endwhile
  a(j + 1) = x
next i
return
```

**Эффективность алгоритма сортировки прямым включением.**

Минимальные оценки числа сравнений  $C_{min}$  и перемещений  $M_{min}$  встречаются в случае уже упорядоченной исходной последовательности элементов, наихудшие же оценки  $C_{max}$  и  $M_{max}$  - когда они первоначально расположены в обратном порядке. В некотором смысле сортировка с помощью включения демонстрирует истинно естественное поведение.

Ясно, что приведенный алгоритм описывает процесс устойчивой сортировки: порядок элементов с равными ключами при нем остается неизменным.

Количество сравнений в худшем случае, когда массив отсортирован противоположным образом,  $C_{max} = n(n - 1)/2$ , то есть порядок  $O(n^2)$ . Количество перестановок  $M_{max} = C_{max} + 3(n-1)$ , то есть порядок  $O(n^2)$ . Если же массив уже отсортирован, то число сравнений и перестановок минимально:  $C_{min} = n-1$ ;  $M_{min} = 3(n-1)$ .

## 6.2 Сортировка методом прямого выбора

Этот прием основан на следующих принципах:

1. Выбирается элемент с наименьшим ключом.
2. Он меняется местами с первым элементом  $a_1$ .
3. Затем этот процесс повторяется с оставшимися  $n-1$  элементами,  $n-2$  элементами и т.д. до тех пор, пока не останется один, самый "большой" элемент.

Алгоритм сортировки прямым выбором:

```
for i = 1 to n - 1
  x = a(i)
  k = i
  for j = i + 1 to n
    if a(j) < x then
      k = j
      x = a(k)
    endif
  next j
  a(k) = a(i)
  a(i) = x
next i
return
```

### **Эффективность алгоритма сортировки прямым выбором.**

Число сравнений ключей  $C$ , очевидно, не зависит от начального порядка ключей. Можно сказать, что в этом смысле поведение этого метода менее естественно, чем поведение прямого включения. Для  $C$  при любом расположении ключей имеем:

$$C = n(n-1)/2$$

Порядок числа сравнений, таким образом,  $O(n^2)$ .

Число перестановок минимально  $M_{min} = 3(n - 1)$  в случае изначально упорядоченных ключей и максимально,  $M_{max} = 3(n - 1) + C$ , т.е. порядок  $O(n^2)$ , если первоначально ключи располагались в обратном порядке.

В худшем случае сортировка прямым выбором дает порядок  $n^2$ , как для числа сравнений, так и для числа перемещений.

### **6.3 Сортировка с помощью прямого обмена (пузырьковая сортировка)**

Классификация методов сортировки редко бывает осмысленной. Оба разбиравшихся до этого метода можно тоже рассматривать как "обменные" сортировки. В данном разделе описан метод, где обмен местами двух элементов представляет собой характернейшую особенность процесса. Изложенный ниже алгоритм прямого обмена основывается на сравнении и смене мест для пары соседних элементов и продолжении этого процесса до тех пор, пока не будут упорядочены все элементы.

Как и в упоминавшемся методе прямого выбора, мы повторяем проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к левому концу массива. Если мы будем рассматривать массивы как вертикальные, а не горизонтальные построения, то элементы мож-

но интерпретировать как пузырьки в чане с водой, причем вес каждого соответствует его ключу. В этом случае при каждом проходе один пузырек как бы поднимается до уровня, соответствующего его весу (рис.6.3).

Такой метод широко известен под именем "пузырьковая сортировка". В своем простейшем виде он представлен ниже.

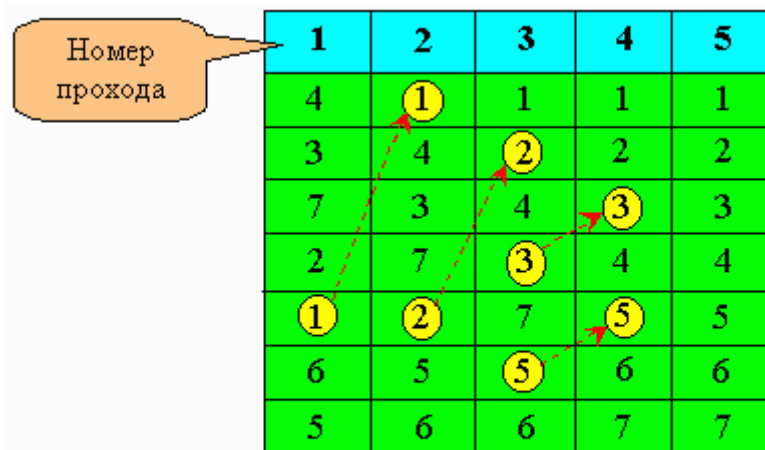


Рис. 6.3.

Алгоритм метода прямого обмена:

```

for i = 2 to n
  for j = n to i step -1
    if a(j) < a(j - 1) then
      x = a(j - 1)
      a(j - 1) = a(j)
      a(j) = x
    endif
  next j
next i
return

```

В нашем случае получился один проход “вхолостую”. Чтобы лишний раз не переставлять элементы, можно ввести флажок *fl*, который остается в значении *false*, если при очередном проходе не будет произведено ни одного обмена. На

нижеприведенном алгоритме добавления отмечены жирным шрифтом.

```
fl = true  
for i = 2 to n  
if fl = false then return  
endif  
fl = false  
for j = n to i step -1  
if a(j) < a(j - 1) then  
    fl = true  
    x = a(j - 1)  
    a(j - 1) = a(j)  
    a(j) = x  
endif  
next j  
next i  
return
```

Улучшением пузырькового метода является **шейкерная сортировка**, где после каждого прохода меняют направление во внутреннем цикле.

**Эффективность алгоритма сортировки прямым обменом.**

Число сравнений  $C_{max} = n(n-1)/2$ , порядок  $O(n^2)$ .

Число перемещений  $M_{max} = 3C_{max} = 3n(n-1)/2$ , порядок  $O(n^2)$ .

Если массив уже отсортирован и применяется алгоритм с флажком, то достаточно всего одного прохода по массиву, и тогда получаем минимальное число сравнений

$C_{min} = n - 1$ , порядок  $O(n)$ , а перемещения вообще отсутствуют.

Сравнительный анализ прямых сортировок показывает, что обменная "сортировка" в классическом виде представляет собой нечто среднее между сортировками с помощью включений и с помощью выбора. Если же в нее внесены приведенные выше усовершенствования, то для достаточно упорядоченных массивов пузырьковая сортировка даже имеет преимущество.

## 6.4 Улучшенные методы сортировки

### 6.4.1 Быстрая сортировка (Quick Sort)

Относится к методам обменной сортировки. В основе лежит методика разделения ключей по отношению к выбранному.

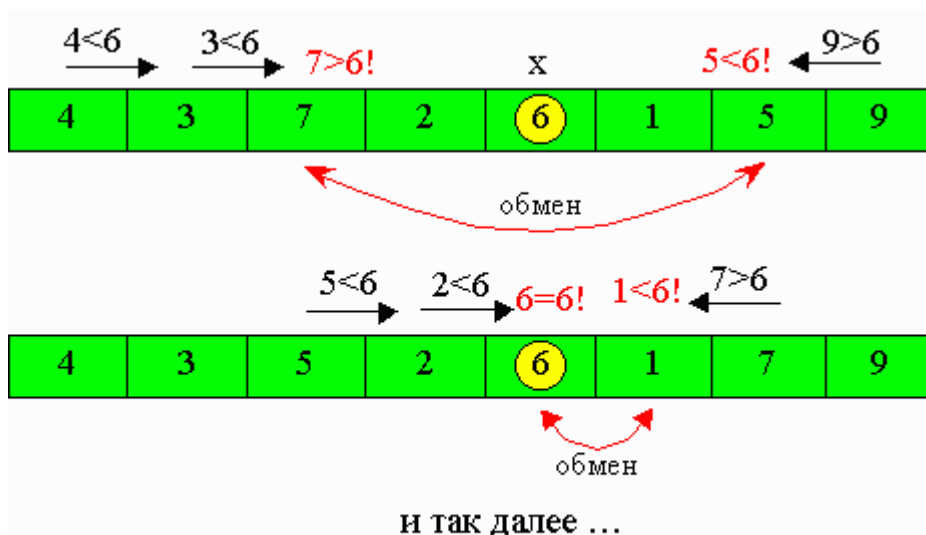


Рис 6.4 . Быстрая сортировка .

Слева от 6 располагают все ключи с меньшими 6, а справа - с большими или равными 6 (рис. 6.4).

Sub Sort (L, R)

```

i = L
j = R
x = a((L + R) div 2)
repeat
  while a(i) < x do
    i = i + 1
  endwhile
  while a(j) > x do
    j = j - 1
  endwhile
  if i ≤ j then
    y = a(i)
    a(i) = a(j)
    a(j) = y
    i = i + 1
    j = j - 1
  endif
until i > j
if L < j then
  sort (L, j)
endif
if i < R then
  sort (i, R)
endif
return

```

```

Sub QuickSort
Sort (l, n)
return

```

**Эффективность алгоритма быстрой сортировки.**

Из всех существующих методов сортировки *Quick Sort* самый эффективный.

Его эффективность имеет порядок  $O(n \log_2 n)$ .



### 6.4.2 Сортировка Шелла (сортировка с уменьшающимся шагом)

В 1959 году Д. Шеллом было предложено усовершенствование сортировки с помощью метода прямого включения. Иллюстрация его сортировки с шагом, равным 4, представлена на рис. 6.5.

сортировка	44	55	12	42	94	18	6	67
после четверной	44	18	6	42	94	55	12	67
двойной	6	18	12	42	44	55	94	67
одинарной	6	12	18	42	44	55	67	94

Рис. 6.5.Сортировка Шелла . .

Сначала отдельно группируются и в группах сортируются элементы, отстоящие друг от друга на расстоянии 4. Такой процесс называется четверной сортировкой. В нашем примере 8 элементов, и каждая группа состоит из двух элементов, то есть 1-й и 5-й элементы, 2-й и 6-й, 3-й и 7-й и, наконец, 4-й и 8-й элементы. После четверной сортировки элементы перегруппировываются - теперь каждый элемент группы отстоит от другого на 2 позиции - и вновь сортируются. Это называется двойной сортировкой. И, наконец, на третьем проходе идет обычная или одинарная сортировка.

На первый взгляд можно засомневаться: если необходимо несколько процессов сортировки, причем в каждый включаются все элементы, то не добавят ли они больше работы, чем сэкономят? Однако, на каждом этапе либо сортируется относительно мало элементов, либо элементы уже довольно хорошо упорядочены и требуют сравнительно немного перестановок.

Ясно, что такой метод в результате дает упорядоченный массив, и, конечно, сразу же видно, что каждый проход от предыдущих только выигрывает. Также очевидно, что расстояния в группах можно уменьшать по-разному, лишь бы последнее было единичным, ведь в самом плохом случае последний проход и сделает всю работу.

Приводимый алгоритм основан на методе прямой вставки без барьера (с условным переходом), и не ориентирован на некую определенную последовательность расстояний, хотя в нем для конкретности заданы шаги 5, 3 и 1.

При использовании метода с барьером каждая из сортировок нуждается в постановке своего собственного барьера, поэтому приходится расширять массив с  $[0..N]$  до  $[-h1..N]$ .

Введем обозначения

$h[1..t]$  - массив размеров шагов

$a[1..n]$  - сортируемый массив

$k$  - шаг сортировки

$x$  - значение вставляемого элемента

Алгоритм сортировки Шелла:

```
const t = 3
      h(1) = 7
      h(2) = 3
      h(3) = 1
for m = 1 to t
  k = h(m)
  for i = 1 + k to n
    x = a(i)
    for j = i - k to 1 step -k
      if x < a(j) then
        a(j+k) = a(j)
      else goto L
    endif
  next j
```

*L: a(j+k) = x*  
*next i*  
*next m*  
*return*

Не доказано, какие расстояния дают наилучший результат, но они не должны быть множителями один другого. Д. Кнут предлагает такую последовательность шагов  $h$  (в обратном порядке): 1, 3, 7, 15, 31, ... . То есть:  $h_m = 2h_{m-1} + 1$ , а количество шагов  $t = (\log_2 n) - 1$ .

При такой организации алгоритма его эффективность имеет порядок  $O(n^{1,2})$ .

### **Контрольные вопросы**

1. Что такое сортировка?
2. Назовите основные методы сортировки.
3. Какие методы сортировки относятся к строгим?
4. Какие методы сортировки относятся к улучшенным?
5. Какая сортировка называется устойчивой?
6. В чем состоит суть метода прямого включения?
7. В чем состоит суть метода прямого выбора?
8. В чем состоит суть метода прямого обмена?
9. Назовите разницу между этими тремя методами.
10. Какой метод сортировки является самым эффективным?
11. К какому из основных методов относится метод Шелла?

## 7. ПРЕОБРАЗОВАНИЕ КЛЮЧЕЙ (РАССТАНОВКА)

Метод расстановок (**хеширования**) направлен на быстрое решение задачи о местоположении элемента в структуре данных. В методе расстановок данные организованы как обычный массив. Поэтому  $H$  - отображение, преобразующее ключи в индексы массива, отсюда и появилось название *«преобразование ключей»*, обычно даваемое этому методу. Следует заметить, что нет никакой нужды обращаться к каким-либо процедурам динамического размещения, ведь массив — это один из основных, статических объектов. Метод преобразования ключей часто используется для таких задач, где можно с успехом воспользоваться и деревьями.

Основная трудность, связанная с преобразованием ключей, заключается в том, что множество возможных значений значительно шире множества допустимых адресов памяти (индексов массива). Возьмем в качестве примера имена, включающие до 16 букв и представляющие собой ключи, идентифицирующие отдельные индивиды из множества в тысячу персон. Следовательно, мы имеем дело с  $26^{16}$  возможными ключами, которые нужно отобразить в  $10^3$  возможных индексов. Поэтому функция  $H$  будет функцией класса «много значений в одно значение». Если дан некоторый ключ  $k$ , то первый шаг операции поиска — вычисление связанного с ним индекса  $h = H(k)$ , а второй (совершенно необходимый) — проверка, действительно ли  $h$  идентифицирует в массиве (таблице)  $T$  элемент с ключом  $k$ .

### 7.1. Выбор функции преобразования

Само собой разумеется, что любая хорошая функция преобразования должна как можно равномернее распределять ключи по всему диапазону значений индекса. Если это требо-

вание удовлетворяется, то других ограничений уже нет, и даже хорошо, если преобразование будет выглядеть как совершенно случайное. Такая особенность объясняет несколько ненаучное название этого метода — «перемалывание» (хеширование), т. е. дробление аргумента, превращение его в какое-то «месиво». Функция же  $H$  будет называться «функцией расстановки». Ясно, что  $H$  должно вычисляться достаточно эффективно, т. е. состоять из очень небольшого числа основных арифметических операций.

Представим себе, что есть функция преобразования  $ORD(k)$ , обозначающая порядковый номер ключа  $k$  в множестве всех возможных ключей. Кроме того, будем считать, что индекс массива  $i$  лежит в диапазоне  $0 \dots N - 1$ , где  $N$  — размер массива. В этом случае ясно, что нужно выбрать следующую функцию:

$$H(k) = ORD(k) \bmod N$$

Для нее характерно равномерное отображение значений ключа на весь диапазон изменения индексов, поэтому ее кладут в основу большинства преобразований ключей. Кроме того, при  $N$ , равном степени двух, эта функция эффективно вычисляется. Однако если ключ представляет собой последовательность букв, то именно от такой функции и следует отказаться. Дело в том, что в этом случае допущение о равновероятности всех ключей ошибочно. В результате слова, отличающиеся только несколькими символами, с большой вероятностью будут отображаться в один и тот же индекс, что приведет к очень неравномерному распределению. Поэтому на практике рекомендуется в качестве  $N$  брать простое число. Следствием такого выбора будет необходимость использования полной операции деления, которую уже нельзя заменить выделением нескольких двоичных цифр.

## 7.2. Алгоритм

Если обнаруживается, что строка таблицы, соответствующая заданному ключу, не содержит желаемого элемента, то, значит, произошел конфликт, т. е. два элемента имеют такие ключи, которые отображаются в один и тот же индекс. В этой ситуации нужна вторая попытка с индексом, вполне определенным образом получаемым из того же заданного ключа. Существует несколько методов формирования вторичного индекса. Очевидный прием — связывать вместе все строки с идентичным первичным индексом  $H(k)$ . Превращая их в связанный список. Такой прием называется *прямым связыванием* (direct chaining). Элементы получающегося списка могут либо помещаться в основную таблицу, либо нет; в этом случае память, где они размещаются, обычно называется *областью переполнения*. Недостаток такого приема в том, что нужно следить за такими вторичными списками и в каждой строке отводить место для ссылки (или индекса) на соответствующий список конфликтующих элементов.

Другой прием разрешения конфликтов состоит в том, что мы совсем отказываемся от ссылок и вместо этого просматриваем другие строки той же таблицы — до тех пор, пока не обнаружим желаемый элемент или же пустую строку. В последнем случае мы считаем, что указанного ключа в таблице нет. Такой прием называется *открытой адресацией*. Естественно, что во второй попытке последовательность индексов должна быть всегда одной и той же для любого заданного ключа. В этом случае алгоритм просмотра строится по такой схеме:

```
h = H(k)
i = 0
repeat
  if T(h) = k
    then элемент найден
```

```

else if T(h) = free
    then элемента в таблице нет
    else {конфликт}
    i := i + 1
    h := H(k) + G(i)
endif
endif
until либо найден, либо нет в таблице (либо она полна)

```

Предлагались самые разные функции  $G(i)$  для разрешения конфликтов. Приведенный в работе Морриса (1968) обзор стимулировал активную деятельность в этом направлении. Самый простой прием — посмотреть следующую строку таблицы (будем считать ее круговой), и так до тех пор, пока либо будет найден элемент с указанным ключом, либо встретится пустая строка. Следовательно, в этом случае  $G(i) = i$ , а индексы  $h_i$ , употребляемые при последующих попытках, таковы:

$$\begin{aligned}
 h_0 &:= H(k) \\
 h_i &:= (h_0 + i) \bmod N, \quad i = 1 \dots N - 1
 \end{aligned}$$

Такой прием называется *линейными пробами*, его недостаток заключается в том, что строки имеют тенденцию группироваться вокруг первичных ключей (т. е. ключей, для которых при включении конфликта не возникало). Конечно, хотелось бы выбрать такую функцию  $G$ , которая вновь равномерно рассеивала бы ключи по оставшимся строкам. Однако на практике это приводит к слишком большим затратам, потому предпочтительнее некоторые компромиссные методы; будучи достаточно простыми с точки зрения вычислений, они все же лучше линейной функции. Один из них — использование квадратичной функции, в этом случае последовательность пробуемых индексов такова:

$$h_0 := H(k)$$

$$h_i := (h_i + i^2) \text{ MOD } N, \quad i > 0$$

Если воспользоваться рекуррентными соотношениями для  $h_i = i^2$  и  $d_i = 2i + 1$  при  $h_0 = 0$  и  $d_0 = 1$ , то при вычислении очередного индекса можно обойтись без операции возведения в квадрат.

$$h_{i+1} = h_i + d_i$$

$$d_{i+1} = d_i + 2 \quad (i > 0)$$

Такой прием называется *квадратичными пробами*, существенно, что он позволяет избежать группирования, присущего линейным пробам, не приводя практически к дополнительным вычислениям. Небольшой же его недостаток заключается в том, что при поиске пробуются не все строки таблицы, т. е. при включении элемента может не найтись свободного места, хотя на самом деле оно есть. Если размер  $N$  — простое число, то при квадратичных пробах просматривается по крайней мере половина таблицы. Такое утверждение можно вывести из следующих рассуждений. Если  $i$ -я и  $j$ -я пробы приводят к одной и той же строке таблицы, то справедливо равенство

$$i^2 \text{ MOD } N = j^2 \text{ MOD } N$$

или

$$(i^2 - j^2) = 0 \text{ (MOD } N \text{)}$$

Разлагая разность на два множителя, получаем

$$(i + j)(i - j) = 0 \text{ (MOD } N \text{)}$$



Поскольку  $i \neq j$ , то либо  $i$ , либо  $j$  должны быть больше  $N/2$ , чтобы было справедливо равенство  $i + j = cN$ , где  $c$  — некоторое целое число. На практике упомянутый недостаток не столь существен, так как  $N/2$  вторичных попыток при разрешении конфликтов встречаются очень редко, главным образом в тех случаях, когда таблица почти заполнена.

### Контрольные вопросы

1. Для чего предназначен метод расстановок ?
2. От чего зависит положение элемента в массива в методе расстановок ?
3. Какую функцию возможно использовать в качестве хэш-функции:
4. Что называется конфликтом ?
5. Какой из методов является методом разрешения конфликтов.
6. Какой из методов разрешения конфликтов позволяет более равномерно распределить элементы по массиву.
7. Почему невозможно применять в качестве  $H(k)$  функцию  $\text{Trunc}(\sqrt{k^5 - i \cdot \tan(k)}) \bmod N$  ?

**ЧАСТЬ 2.**  
**ПРАКТИКУМ ПО СТРУКТУРАМ И**  
**АЛГОРИТМАМ ОБРАБОТКИ ДАН-**  
**НЫХ В ЭВМ**

# МЕТОДИЧЕСКОЕ РУКОВОДСТВО К ЛАБОРАТОРНЫМ РАБОТАМ

## Организационно-методические указания

1. Перед началом лабораторной работы проводится консультация по методике выполнения лабораторных работ по данной дисциплине.
2. Объем каждой лабораторной работы, подготовка и порядок выполнения построены таким образом, чтобы все студенты выполнили работу и сдали отчеты.
3. Студенты готовятся к выполнению очередной работы заблаговременно.
4. Студенты обязаны изучить технику безопасности при работе на лабораторных установках до 1000 В.
5. Готовясь к лабораторному занятию, студент обязан изучить необходимый теоретический материал, пользуясь настоящими указаниями и рекомендованной литературой, произвести необходимые расчеты, заполнить соответствующую часть отчета и дать ответы на контрольные вопросы.
6. Неподготовленные студенты к выполнению лабораторной работы не допускаются.
7. Студенты, не сдавшие отчет во время занятия, сдают его в назначенное преподавателем время.
8. Студент, не выполнивший лабораторную работу, выполняет ее в согласованное с преподавателем время.
9. Каждая лабораторная работа выполняется студентами самостоятельно. Все студенты предъявляют индивидуальные отчеты. Допускается предъявление отчета в виде электронного документа.

10. Проверка знаний студентов производится преподавателем во время лабораторного занятия и при сдаче отчета.
11. При сдаче отчета студент должен показать знание теоретического материала в объеме, определяемом контрольными вопросами, а также пониманием физической сущности выполняемой работы.

## **ПОЛУСТАТИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ**

**Лабораторная работа №1 (4 часа). Полустатические структуры данных**

### **Цель работы:**

- исследовать и изучить полустатические структуры данных (на примере стеков, реализованных с помощью массивов);
- овладеть навыками разработки алгоритмов и написания программ на по исследованию стеков на языке программирования C++;

### **Порядок выполнения работы**

- ознакомиться с краткой теорией и примерами решения задач, относящихся к работе со стеками;
- ответить на контрольные вопросы и по знанию теории;
- получить задание на выполнение конкретного варианта лабораторной работы у преподавателя и выполнить его;
- написать и отладить программу решения задачи на языке C++;
- подготовить отчет по лабораторной работе и защитить его у преподавателя.

### **Содержание отчета по ЛР**

- наименование ЛР и ее цель;
- задание на ЛР согласно варианту;
- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;
- результаты работы программы.

### **Краткая теория**

Понятие очереди всем хорошо известно из повседневной жизни. Элементами очереди в общем случае являются заказы на то или иное обслуживание: выбить чек на нужную сумму в кассе магазина; получить нужную информацию в справочном

бюро, выполнить очередную операцию по обработке детали на данном станке в автоматической линии и т.д.

В программировании имеется структура данных, которая называется очередь. Эта структура данных используется, например, для моделирования реальных очередей с целью определения их характеристик (средняя длина очереди, время пребывания заказа в очереди и т.п.) при данном законе поступления заказов и дисциплине их обслуживания.

По своему существу очередь является полустатической структурой - с течением времени и длина очереди, и набор образующих ее элементов могут изменяться.

Различают два основных вида очередей, отличающихся по дисциплине обслуживания находящихся в них элементов:

1. При первой из дисциплин заказ, поступивший в очередь первым, выбирается первым для обслуживания (и удаляется из очереди). Эту дисциплину обслуживания принято называть **FIFO** (*First input-First output*, т.е. первый пришел - первый ушел). Очередь открыта с обеих сторон.



2. Вторую дисциплину принято называть **LIFO** (*Last input - First output*, т.е. последний пришел - первый ушел), при которой на обслуживание первым выбирается тот элемент очереди, который поступил в нее последним. Очередь такого вида в программировании принято называть **СТЕК**ОМ (магазином) - это одна из наиболее употребительных структур данных, которая оказывается весьма удобной при решении различных задач.

В силу указанной дисциплины обслуживания, в стеке доступна единственная его позиция, которая называется **ВЕРШИНОЙ** стека - эта позиция, в которой находится последний по времени поступления в стек элемент. Когда мы заносим новый элемент в стек, то он помещается поверх вершины и

теперь уже сам находится в вершине стека. Выбрать элемент можно только из вершины стека; при этом выбранный элемент исключается из стека, а в его вершине оказывается элемент, который был занесен в стек перед выбранным из него элементом (структура с ограниченным доступом к данным).



## Алгоритмы

### ОПЕРАЦИИ НАД СТЕКАМИ:

- PUSH (S, x) - занесение элемента в стек, где S - название стека, x - элемент, который заносится в стек;
- POP (S) - выборка элемента из стека. При выборке элемент помещается в рабочую область памяти, где он используется;
- EMPTY (S) - проверка стека на пустоту (true - пуст, false - не пуст);
- STACKTOP (S) - чтение верхнего элемента без его удаления.
- FULL (S) – проверка стека на переполнение (в случае, если стек реализован с помощью массива).

Если  $i$  – указатель вершины стека, то реализация описанных выше операций в псевдокоде будет выглядеть следующим образом:

Push(S,x)

$i = i + 1$

$S(i) = x$

*return*

Pop(S)

$x = S(i)$

$i = i - 1$

*return*

Empty(S)

*if*  $i = 0$

*then* “*нуисто*”

*Stop*

*return*

*endif*

*условие*  $i=0$  *означает, что стек пуст*

Full(S)

*if*  $i = \max S$

*then* “*переполнение*”

*Stop*

*return*

*endif*

StackTop(S)

$x = S(i)$

*return*

Если при выборке проверять стек на пустоту, а при занесении элемента проверять стек на переполнение, то алгоритмы операций считывания и занесения элемента будут следующими:

Pop(S)

*if*  $i = 0$  *then* “*нуисто*”

*Stop*

*return*

*endif*

$x = S(i)$



$i = i - 1$

*return*

Push(S,i)

*if*  $i = \max S$

*then* “переполнение”

*Stop*

*return*

*endif*

$i = i + 1$

$S(i) = x$

*return*

## **Контрольные вопросы по теории**

1. В чём особенности очереди?
  - открыта с обеих сторон;
  - открыта с одной стороны на вставку и удаление;
  - доступен любой элемент.
2. В чём особенности стека?
  - открыт с обеих сторон на вставку и удаление;
  - доступен любой элемент;
  - открыт с одной стороны на вставку и удаление.
3. Какую дисциплину обслуживания принято называть FIFO ?
  - стек;
  - очередь;
  - дек.
4. Какая операция читает верхний элемент стека без удаления ?
  - pop;
  - push;
  - stackpop.
5. Какого правило выборки элемента из стека ?
  - первый элемент;

- последний элемент;
- любой элемент.

### **Примеры алгоритмов конкретных задач (стеки и операции над ними)**

Рассмотрим, как операции над стеком реализуются на языке программирования C++ с помощью функций. Простейший стек будем реализовывать на одномерном массиве (векторе), элементом стека будет символьная переменная. Обычно указатель стека обозначается `sp` (Stack Pointer), в любой момент времени он содержит адрес текущего элемента, являющегося вершиной стека и единственным элементом, доступным в данный момент времени для работы со стеком.

Если исходить из предположения, что вершина стека – последний свободный элемент массива, то операция занесения элемента в стек реализуется присваиванием значения вводимого символа данному элементу. Значение указателя стека при этом должно увеличиваться на единицу, задавая ячейку, как бы находящуюся “над” верхним элементом. При такой реализации представляется вполне возможным заполнение стека с нулевого элемента массива. Если при этом задать начальное значение `sp=1`, легко можно реализовать все операции работы со стеком.

Начальная установка `sp=1`.

Загрузка элемента `x` в стек: `stack[sp]=x; sp=sp+1`.

Извлечение элемента из стека: `sp=sp-1; x=stack[sp]`;

Необходимо учитывать, что массив содержит конечное число элементов, поэтому при занесении элемента в стек необходимо осуществлять проверку на переполнение, поэтому загрузка элемента в стек должна осуществляться с проверкой на переполнение, тогда операция занесения элемента в стек будет выглядеть следующим образом:

```
if (sp<=sd) { stack[sp]=x; sp=sp+1 }
else //стек полон
```

Здесь *sd* – размерность стека (максимальное число элементов массива плюс один, так как в C++ нумерация индексов в массиве начинается с нуля).

При извлечении элемента из стека и при считывании значения верхнего элемента без извлечения необходимо осуществлять проверку стека на пустоту, поэтому операция извлечения реализуется так:

```
if (sp>1) {sp=sp-1; x=stack[sp]}  
    else // стек пуст.
```

Чтение верхнего элемента без извлечения:

```
if (sp>1) {x=stack[sp-1]}  
    else // стек пуст.
```

Поскольку наш стек – последовательность символов, то фрагмент программы с основными функциями работы со стеком будет выглядеть следующим образом:

```
#define MAX_SIZE 20  
char s[MAX_SIZE]; //компоненты стека  
int next=0; // позиция стека  
  
int Empty() // проверка на пустоту  
{ return next==0; }  
  
int Full() // проверка на переполнение  
{ return next==MAX_SIZE; }  
  
void Push() // Занесение элемента в стек  
{  
    if (next==MAX_SIZE)  
    {  
        cout<<"Ошибка: стек полон"<<endl;}  
    else { next++;  
        cout<<"Что поместить в стек?"<<endl;  
        cin >> s[next-1];cout<<"Добавлен"<<endl;  
    }  
}
```

```
}
```

```
void Pop()// Считывание элемента с удалением  
{  
    if (next==0) cout<<"Ошибка: стек пуст"<<endl;  
    else {  
        next--;cout<<"Удален "<<endl;  
    }  
}
```

```
Void Stacktop() // считывание элемента без удаления  
{  
    if (next==0) cout<<"Ошибка: стек пуст"<<endl;  
    else {  
        cout<<s[next-1]<<endl;  
    }  
}
```

*// Данная функция выводит верхний элемент стека на экран.*

Теперь рассмотрим пример конкретной программы, которая позволяет работать с полустатистическим стеком.

```
// Работа со стеком. Проверка стека на пустоту.  
// Добавление элемента в стек. Выборка элемента из  
стека.  
// Проверка стека на переполнение. Печать стека.  
// Просмотр содержимого стека без считывания эле-  
ментов  
  
#include <stdio.h>  
#include <dos.h>  
#include <iostream.h>
```

```

#include <Process.H>
#include <Stdlib.H>
#include <conio.H>
#define MAX_SIZE 200
char s[MAX_SIZE]; //компоненты стека
int next=0; // позиция стека

int Empty()
{ return next==0; }

int Full()
{ return next==MAX_SIZE; }

void Push()
{
    if (next==MAX_SIZE)
    {
        cout<<"Ошибка: стек полон"<<endl;}
        else { next++;
        cout<<"Что поместить в стек?"<<endl;
        cin >> s[next-1];cout<<"Добавлен"<<endl;
    }
}

void Printst()
{
    if (next==0)

```

```

        cout<<"Стек пусм"<<endl;
    else
    do
    {cout<<s[next-1]<<" "<<endl;next--;}
    while(next!=0);
}

void Clear()
{ next=0; }

void Pop()
{
    if (next==0) cout<<"Ошибка: стек пусм"<<endl;
    else {
        next--;cout<<"Удален "<<endl;
    }
}

void Stacktop()
{
    if (next==0) cout<<"Ошибка: стек пусм"<<endl;
    else
    {cout<<s[next-1]<<endl;}
}

void Showst()
{
    int i=0;

```

```

    if (next==0) {
        cout<<"Стек пуст"<<endl;
    }
    else { for(i=0;i<next;i++)
        cout<<s[i]<<" "<<endl;
    }
}

void menu()
{
    cout<<"0: Печать стека"<<endl;
    cout<<"1: Добавление элемента в стек"<<endl;
    cout<<"2: Удаление элемента из стека"<<endl;
    cout<<"3: Считывание элемента из стека без удале-
ния"<<endl;
    cout<<"4: Проверка стека на пустоту"<<endl;
    cout<<"5: Проверка стека на переполнение"<<endl;
    cout<<"6: Очистка стека"<<endl;
    cout<<"7: Просмотр содержимого стека без считы-
вания элементов"<<endl;
    cout<<"8: Выход"<<endl;
}

main()
{
    char c;
    clrscr();
    textcolor(15);
    do {

```

```

    menu();
    cin >> c;
    clrscr();
    switch (c) {
case '0':Printst();getch();break;
case '1':Push();break;
case '2':Pop();getch();break;
case '3':Stacktop();getch();break;
case '4':if (Empty()==1) cout<<"Стек пуст"<<endl;
    else cout<<"Стек не пуст"<<endl;getch();break;
case '5':if (Full()==1)cout<<"стек полон"<<endl;
    else cout<<"стек не полон"<<endl;getch();break;
case '6':Clear();cout<<
    "Стек очищен"<<endl;getch();break;
case '7':Showst();getch();break;
case '8':exit(1);
    }
    delay(200);
}
while (c!=8);
return 0;
}

```

В данной программе функция Printst() выводит содержимое стека на экран в любой момент работы со стеком, при этом стек опустошается. Корректная работа с данной структурой действительно не предусматривает вывода всего содержимого без последовательного считывания с удалени-



ем элементов. На практике может возникнуть необходимость вывода содержимого стека без удаления из него элементов для отладки работы программы. Возможность такого вывода элементов в данной программе предоставляет функция Showst().

Теперь рассмотрим более сложные варианты реализации стеков и работы с ними.

Создадим файл, в котором определены структура дескриптора стека STC и переменная *s1* типа STC, а также включены функции, реализующие рассмотренные выше операции над стеками. Дескриптор построен транслятором, память под элементы стека получена динамически. Элементы стека имеют значения типа EL, максимальное число элементов *m*. В дескрипторе определены указатель начала стека в виде адреса начала динамической памяти и указатели вершины и конца стека в виде целых чисел (индексов элементов стека). Этот файл включается директивой #include в исходный файл с программой для работы со стеком. Предварительно должен быть определен тип элемента EL, например define double EL. Допускаются типы EL, только такие, что переменным этого типа можно присваивать значения оператором «=». Таковыми являются скалярные типы (int, float, double, char) и структурный тип struct.

*/\* Файл включения для работы со стеком.*

*Содержит дескриптор стека и функции для работы*

*со стеком. Включается в головной файл после  
определения элемента стека с именем EL \*/*

*/\* c:\bcpp\bin\incl\_stc.c \*/*

*#define STC struct st*

*STC /\* дескриптор стека \*/*

```

    { EL *un; /* Указатель начала стека */
      int uk; /* Указатель конца стека */
      int uv; /* Указатель вершины стека */
      int m; /* число элементов в стеке */
    } s1; /* s1 -переменная типа STC */
  /*
=====
= */

/*    ДОБАВЛЕНИЕ ЭЛЕМЕНТА В СТЕК    */
int Push_el(STC *s,EL el)
{ if (s->un == NULL) /*стек не создан */
    return -2;
  if (s->uv == s->uk)
    return -1; /* стек полон */
    *(s->un + s->uv+1) = el; ++s->uv;
    return 0;
}
/*
=====
*/

/*    ВЫБОРКА ЭЛЕМЕНТА ИЗ СТЕКА    */
int Pop_el(STC *s,EL *el)
{ if (s->un == NULL)
    return -2; /* стек не создан */
  if (s->uv < 0)
    return -1; /* стек пуст */
  else
    { *el = *(s->un + s->uv);

```

```

        --s->uv;
        return 0;
    }
}
/*
=====
*/

/* ИЗВЛЕЧЕНИЕ ЗНАЧЕНИЯ ЭЛЕМЕНТА ИЗ СТЕКА
БЕЗ УДАЛЕНИЯ ЭЛЕМЕНТА */
int Peek_el(STC *s, EL *el)
{ if (s->un == NULL)
    return -2; /* стек не создан */
  if (s->uv < 0)
    return -1; /* стек пуст */
  else
    { *el = *(s->un + s->uv);
      return 0;
    }
}
/*
=====
*/

/*    ОСВОБОЖДЕНИЕ СТЕКА    */
int Destroy_stc(STC *s)
{ free(s->un);
  s->un = NULL; return 0;
}

```

```

/*
=====
*/

/* СОЗДАНИЕ СТЕКА */
int Crt_stc(STC *s)
{ int nn=12; /* число элементов стека */
  if (s->un != NULL)
    { printf("\n Старый стек уничтожить? (y/n)
");
      fflush();
      if (getchar() != 'y')
        { printf("\n Работаем со старым сте-
ком");
          return -2;
        }
      }
    s->un = (EL*) calloc(nn,sizeof(EL));
    if (s->un == NULL)
      return -1; /* память не выделена */
    else
      { s->uv = -1; s->uk = nn-1;
        s->m = nn; return 0;
      }
  }
/*
*****
*/

/* **** Конец файла включения **** */

```

Теперь рассмотрим пример программы для работы со стеком в векторной памяти. Элементом стека является переменная типа struct, Хотя в структуре содержится единственный элемент - строка. Это вызвано тем ограничением, о котором говорилось выше. Содержанием элемента стека является команда операционной системы либо имя исполняемого файла. Обработка элемента стека сводится к выполнению этой команды или исполняемого файла.

*/\* РАБОТА СО СТЕКОМ В ВЕКТОРНОЙ ПАМЯТИ*

*c:\bcpp\bin\dstackg2.c - головной файл \*/*

```
#include <stdio.h>
#include <alloc.h>
#include <conio.h>
#include <string.h>
#include <process.h>
typedef struct /*Структура элемента стека с именем EL
*/
    { char Name[80];
      } EL;
EL e;
#include "c:\bcpp\bin\incl_stc.c" /*файл включения */
char *menu[7][40];
static int p=1;
int In_el(EL*);
int Show_stc(STC);
void main_menu(void);
```

```

/*      =====ГЛАВНАЯ      ФУНКЦИЯ
===== */

int main()
{
    *menu[0]="1.Создание пустого стека";
    *menu[1]="2.Включение элемента в стек";
    *menu[2]="3.Выборка элемента из стека";
    *menu[3]="4.Освобождение стека";
    *menu[4]="5.Вывод содержимого стека на экран";
    *menu[5]="6.Конец работы";
    *menu[6]="      Введите номер строки";
    clrscr();
    printf("  Работа со стеком в векторной памяти\n");
    while(p)
        {
            main_menu();
            clrscr();
        }
    printf("  Конец работы со стеком\n");
    return 0;
}

/*
=====
= */

/* ВЫВОД ГЛАВНОГО МЕНЮ */
void main_menu(void)
{
    char ns; int pp=1,r=0,i; flushall(); /* чистка буферов */
    while (pp)
        {
            for(i=0;i<7;i++)
                printf("\n %s",*menu[i]);

```

```

printf("\n");
ns=getchar();
if (ns<'1' || ns>'6')
{ clrscr();
    printf("\nОшибка в номере!!Будьте внимательны.");
    continue;
}
else pp=0;
switch(ns)
{ case '1':if ( Crt_stc(&s1) == -1)
    { printf("\n Память под стек не
выделена");

        getch();
    } break;
case '2':if (In_el(&e) == 0)
    { r=Push_el(&s1,e);
        if (r == -2)
        { printf("\nСтек
не создан!!!");

            getch();
        }
    }
else
    if (r == -1)
    { printf("\n Стек
полон!!!");

        getch();
    }
}

```

```

        } break;
case '3': r=Pop_el(&s1,&e);
        if (r == -1)
            printf("\n      Стек
пуст");
        else
            if (r == -2)
                printf("\n Стек не
создан!!");
            else
                { printf("\n Эле-
мент выбран\n");
                /* Обработка
элемента */
                system(e.Name);
                }
            getch(); break;
case '4': Destroy_stc(&s1); break;
case '5': if (Show_stc(s1) == -1)
            { printf("\n Стек не
создан");
            getch();
            } break;
case '6': p=0;
    }
}
}

```



```

/*
=====
= */

    int In_el(EL *el)
    { printf("\n Ввод элемента стека или ** для отказа от
ввода");

        printf("\n Введите команду DOS или имя исполняе-
мого файла\n=>");

        flushall();
        gets(el->Name);
        return 0;
    }
/*
=====
===== */

    int Show_stc(STC s)
    { int i;

        if (s.un == NULL)
            return -1;

        for (i=0; i<=s.uv; i++)
            printf("\n %s",s.un[i].Name);

        getch();
        return 0;
    }
/*
*****
*** */

```

## **Задания**

*Ввести символы, формируя из них стек.*

### **Варианты**

1. Поменять местами первый и последний элементы стека.
2. Развернуть стек, т.е. сделать "дно" стека вершиной, а вершину - "дном"
3. Удалить элемент, находящийся в середине стека, если число элементов нечетное, или 2 средних элемента, если число элементов четное.
4. Удалить каждый второй элемент стека
5. Вставить символ '\*' в середину стека, если число элементов четное, или после среднего элемента, если число элементов нечетное.
6. Найти минимальный элемент и вставить после него 0.
7. Найти максимальный элемент и вставить после него 0
8. Удалить минимальный элемент.
9. Удалить все элементы, равные первому.
10. Удалить все элементы, равные последнему.
11. Удалить максимальный элемент.
12. Найти минимальный элемент и вставить на его место 0.

Вывести полученный стек на экран.

**Составить отчет по лабораторной работе и защитить его у преподавателя**

## **ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ**

### **Лабораторная работа №2 (4 часа). Списковые структуры данных**

#### **Цель работы:**

- исследовать и изучить списковые структуры данных и их основные процедуры;
- овладеть умениями и навыками написания программ по исследованию списковых структур данных и их основных процедур на языке программирования C++;

#### **Порядок выполнения работы**

- ознакомиться с краткой теорией и примерами решения задач, относящихся к работе со списковыми структурами данных;
- ответить на контрольные вопросы и получить оценку по знанию теории;
- получить задание на выполнение конкретного варианта лабораторной работы и выполнить его;
- написать и отладить программу решения задачи на языке C++;
- подготовить отчет по лабораторной работе и защитить его у преподавателя.

#### **Содержание отчета по ЛР**

- наименование ЛР и ее цель;
- задание на ЛР согласно варианту;
- Листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;
- результаты работы программы.

## Краткая теория

На ЛЗ №1 рассматривались только статические программные объекты. Этим термином обозначаются объекты, которые порождаются непосредственно перед выполнением программы, существуют в течение всего времени ее выполнения и размер значений которых не изменяется по ходу выполнения программы.

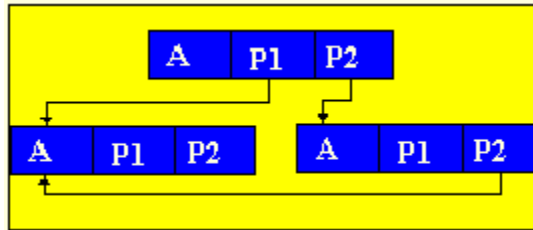
Поскольку статические объекты порождаются до выполнения программы и размер их значений можно выделить еще на этапе трансляции исходного текста программы на языке машины.

Однако использование при программировании только статических объектов может вызвать определенные трудности, особенно с точки зрения получения эффективной машинной программы, а такая эффективность бывает чрезвычайно важной при решении ряда задач. Дело в том, что иногда мы заранее не знаем не только размера значения того или иного программного объекта, но даже и того, будет ли существовать этот объект или нет. Такого рода программные объекты, которые возникают уже в процессе выполнения программы, называют динамическими объектами.

Динамические структуры данных имеют две особенности  
:

1. Заранее не определимо количество элементов в структуре;
2. Элементы динамической структуры не имеют жесткой линейной упорядоченности. Они могут быть разбросаны по памяти.

Чтобы связать элементы динамической структуры между собой, в состав элемента помимо информационного поля входят поля указателей (связок с другими элементами структуры).



$p1$ ,  $p2$  - указатели, содержащие адреса элементов, с которыми данный элемент связан.

Наиболее распространенными динамическими структурами являются связанные списки. С точки зрения логического представления различают *линейные* и *нелинейные* списки. В линейных списках связи строго упорядочены. Указатель предыдущего элемента дает адрес последующего элемента или наоборот.

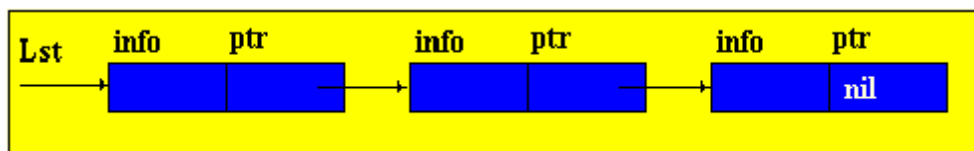
К линейным спискам относятся *односвязные* и *двусвязные* списки.

К нелинейным - многосвязные списки.

Элемент списка в общем случае представляет собой комбинацию поля записи (информационного поля) и одного или нескольких указателей.

## Алгоритмы

### Линейные однонаправленные списки (односвязные списки)



Под односвязными списками понимают упорядоченную последовательность элементов, каждый из которых имеет 2 поля:

- информационное поле *info* и

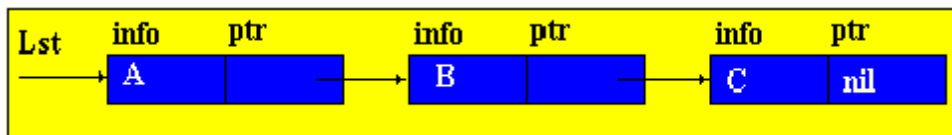
- поле указателя *ptr* .

Особенностью указателя является то, что он дает только адрес последующего элемента списка. У однонаправленных списков поле указателя последнего элемента в списке является пустым *nil*.

*Lst* - указатель начала списка. Он представляет список как единое целое. Иногда список может быть пустым, т.е. в данном списке нет ни одного элемента. В этом случае *lst = nil*.

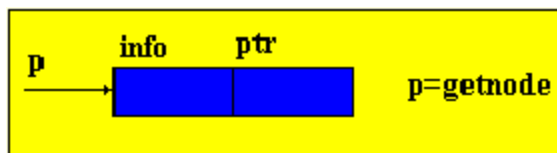
### Операции с односвязными списками.

#### 1. Вставка элемента в начало односвязного списка.

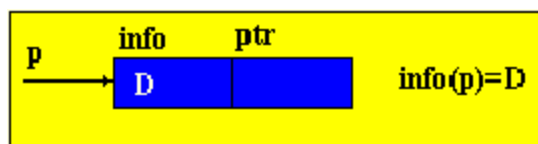


Вставим в начало данного списка элемент, информационное поле которого содержит переменную *D*. Чтобы это осуществить, необходимо произвести следующие действия :

а) Создать пустой элемент, на который указывает указатель *p*.



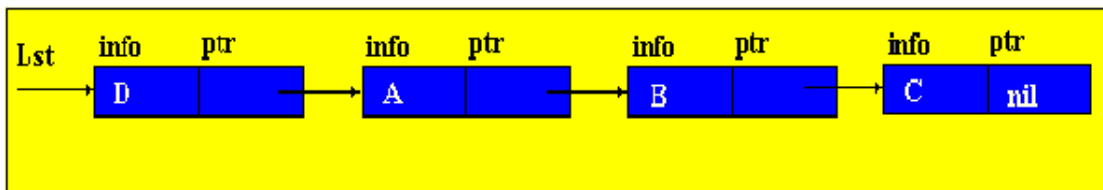
б) Информационному полю созданного элемента присвоить значение *D*.



в) Связать новый элемент со списком.

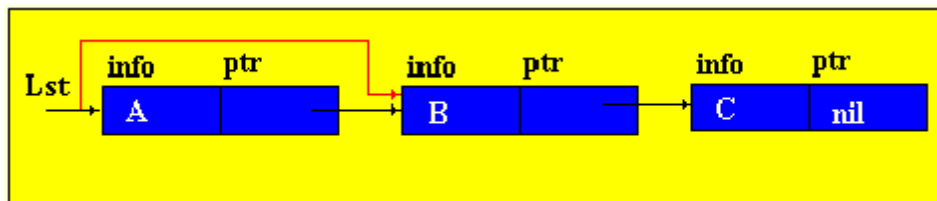
ка)  $\text{Ptr}(p)=\text{lst}$  (*lst* - уже не указывает на начало списка)

д) Перенести указатель *lst* на начало списка.  
Окончательно:



## 2. Удаление элемента из начала односвязного списка

Удалим 1-й элемент списка, но при этом запомним информацию, содержащуюся в поле info этого элемента.



Чтобы это осуществить, необходимо произвести следующие действия :

а) Ввести указатель p, который будет указывать на удаляемый элемент.

$P = \text{lst}$

б) Запомнить поле info элемента, на который ссылается указатель p, в некоторую переменную (x).

$X = \text{info}(P)$

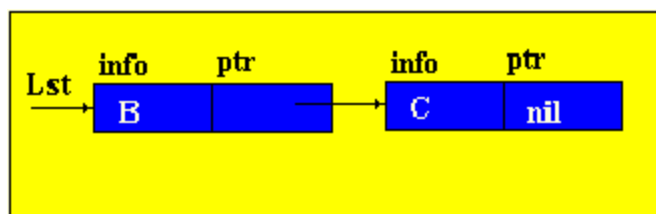
с) Перенести указатель lst на новое начало списка.

$\text{lst} = \text{ptr}(P)$

д) Удалить элемент на который указывает указатель p.

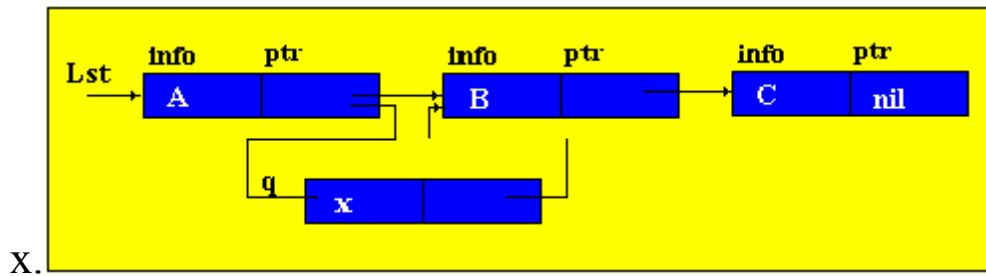
$\text{Freenode}(P)$

Окончательно:



## 3. Вставка элемента в список

Вставим в данный список после элемента на который указывает указатель p, элемент с информационным полем x.



Чтобы это осуществить, необходимо произвести следующие действия :

а) Создать пустой элемент на который указывает указатель q.

$Q = \text{getnode}$

б) Внести x в информационное поле созданного элемента.

$\text{Info}(Q) = x$

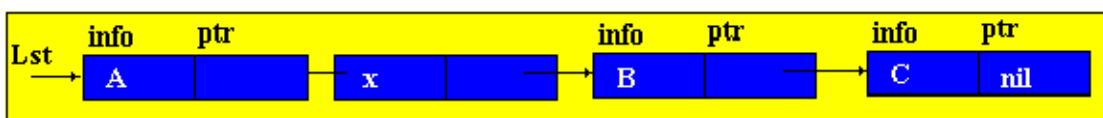
с) Связать элемент x с элементом B.

$\text{Ptr}(Q) = \text{Ptr}(P)$  - это значит, что указателю созданного элемента присваивается значение указателя элемента p.

д) Связать элемент A с элементом x.

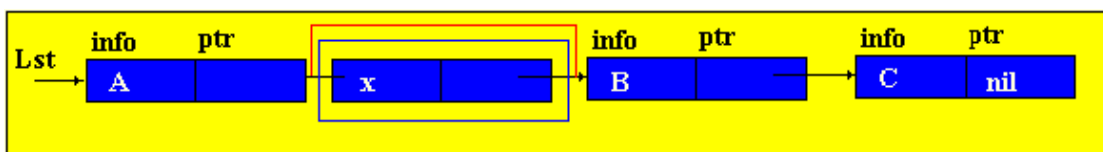
$\text{Ptr}(p) = Q$  - это значит, что следующим за элементом A будет элемент на который указывает указатель Q.

Окончательно:



#### 4. Удаление элемента из односвязного списка

Удалим из списка элемент, который следует за элементом с рабочим указателем p.



Чтобы это осуществить необходимо произвести следующие действия :



a) Ввести указатель Q, который будет указывать на удаляемый элемент.

$Q = \text{ptr}(p)$

b) Поставить за элементом A элемент B.

$\text{Ptr}(p) = \text{Ptr}(Q)$

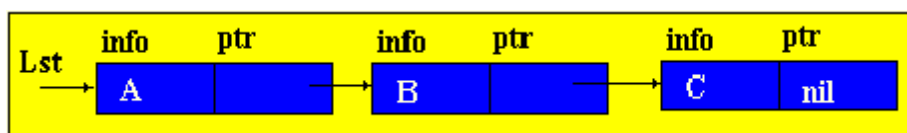
c) Запомнить информацию, которая содержится в поле info удаляемого элемента.

$K = \text{info}(Q)$

d) Удалим элемент с указателем Q.

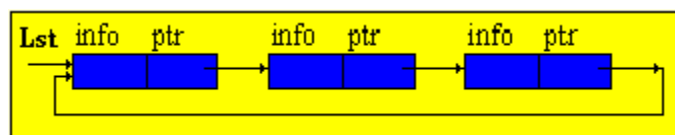
$\text{Freenode}(Q)$

Окончательно:



## Кольцевые списки

Пример кольцевого списка представлен на следующем рисунке.



На этом рисунке, список замыкается в своеобразное "кольцо": двигаясь по ссылкам, можно от последнего элемента списка переходить к заглавному элементу. В связи с этим списки подобного рода называют кольцевыми списками.

Чтобы закольцевать список необходимо присвоить указателю последнего элемента указатель начала списка ( $\text{Ptr}(p) = \text{lst}$ ).

p - указатель последнего элемента;

Lst - указатель начала списка.

## Операции с кольцевыми списками:

## 1. Вставка элемента в кольцевой список

Чтобы осуществить эту операцию необходимо произвести следующие действия:

а) Создать пустой элемент на который указывает указатель q

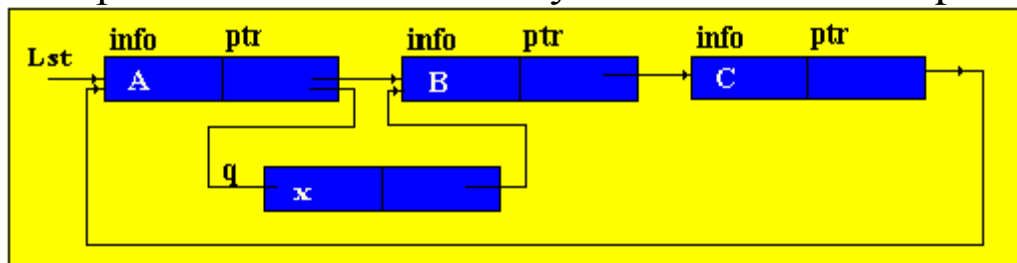
$q = \text{getnode}$

б) Внести x в информационное поле созданного элемента

$\text{info}(q) = x$

в) Связать элемент X с элементом B

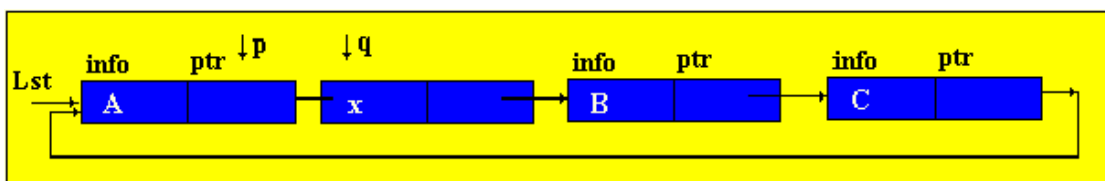
$\text{ptr}(q) = \text{ptr}(p)$  - это означает, что указателю созданного элемента присваивается значение указателя элемента p.



д) Связать элемент A с элементом X

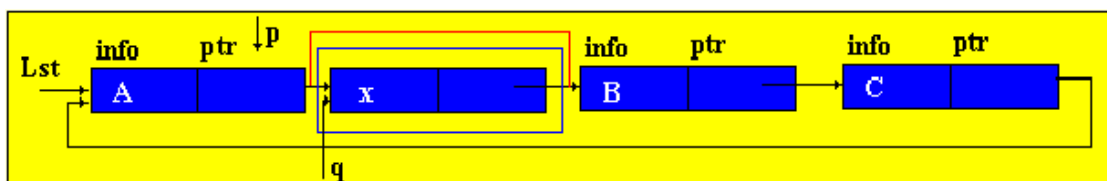
$\text{ptr}(p) = q$  - это означает, что следующим за элементом A будет элемент на который указывает указатель q.

Окончательно:



## 2. Удаление элемента из кольцевого списка

Удалим из списка элемент, который следует за элементом с рабочим указателем p.



Чтобы это осуществить, необходимо произвести следующие действия:

а) Ввести указатель q, который будет указывать на удаляемый элемент.

$q = \text{ptr}(p)$

б) Поставить за элементом A элемент B

$\text{ptr}(p) = \text{ptr}(q)$

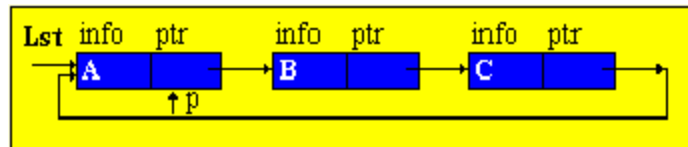
с) Запомнить информацию, которая содержится в поле info удаляемого элемента.

$k = \text{info}(q)$

д) Удалить элемент с указателем q.

$\text{Freenode}(q)$

Окончательно:



## Контрольные вопросы по теории

### Линейные однонаправленные списки (односвязные списки)

1. Как освободить память от удаленного из списка элемента?

- $p = \text{getnode};$
- $\text{ptr}(p) = \text{nil};$
- $\text{freenode}(p);$
- $p = \text{lst}.$

2. Как создать новый элемент списка с информационным полем D?

- $p = \text{getnode};$
- $p = \text{getnode}; \text{info}(p) = D;$
- $p = \text{getnode}; \text{ptr}(D) = \text{lst}.$

3. Как создать пустой элемент с указателем p?

- $p = \text{getnode};$
- $\text{info}(p);$
- $\text{freenode}(p);$
- $\text{ptr}(p) = \text{lst}.$

4. Сколько указателей используется в односвязных списках?

- 1;
- 2;
- сколько угодно.

5. В чём отличительная особенность динамических объектов?

- порождаются непосредственно перед выполнением программы;
- возникают уже в процессе выполнения программы;
- задаются в процессе выполнения программы.

### **Кольцевые структуры данных.**

6. При удалении элемента из кольцевого списка

- список разрывается;
- в списке образуется дыра;
- список становится короче на один элемент.

7. Для чего используется указатель в кольцевых списках?

- для ссылки на следующий элемент;
- для запоминания номера сегмента расположения элемента;
- для ссылки на предыдущий элемент;
- для расположения элемента в списке памяти.

8. Чем отличается кольцевой список от линейного списка?

- в кольцевом списке последний элемент является одновременно и первым;
- в кольцевом списке указатель последнего элемента пустой;
- в кольцевых списках последнего элемента нет;
- в кольцевом списке указатель последнего элемента не пустой.

9. Сколько указателей используется в односвязном кольцевом списке?

- 1;
- 2;

- сколько угодно.

10. В каких направлениях можно перемещаться в кольцевом двунаправленном списке?

- в обоих;
- влево;
- вправо.

### **Примеры алгоритмов конкретных задач**

Рассмотрим, как реализуются возможные с односвязными списками операции на языке программирования C++ с помощью функций.

Элемент списка можно определить как структуру, внутри которой содержится указатель на следующий элемент такой же структуры. В простейшем случае, если информационным полем элемента односвязного списка являются целые числа, элемент списка можно описать так:

```
struct TNode;  
typedef TNode* PNode;  
struct TNode  
{  
    int Data;  
    PNode Next;  
};
```

Здесь с помощью typedef мы определяем PNode как указатель на элемент данной структуры.

Функцию вставки элемента в начало односвязного списка можно описать следующим образом:

```
void InsertFirst(PNode& First, int Data)  
{  
    PNode p = new TNode;
```

```

    p->Data=Data;
    p->Next=First;
    First=p;
}

```

Происходит генерация динамического элемента структуры с рабочим указателем *p*, после чего указателю на следующий элемент списка присваивается значение *First* (*p->Next=First*; *First* уже не указывает на начало списка), затем указателю начала списка присваивается значение рабочего указателя *p* (*First=p*).

Функцию удаления элемента из начала односвязного списка опишем следующим образом:

```

void DeleteFirst(PNode& First)
{
    PNode p=First;
    if(p==0)    cout<<"    Ошибка:    В    списке    нет
элементов"<<endl;
    else
    {First=First->Next;
    Delete p;}
}

```

В случае, когда вставка всегда осуществляется в начало односвязного списка, с его помощью реализуется чисто динамический стек, причем функция *AddFirst(PNode& First, int Data)* реализует операцию занесения в стек нового элемента, а функция *DelFirst(PNode& First)* – считывания элемента из стека. Проверка на пустоту осуществляется по значению рабочего указателя *p* (*if(p==0)* “Список пуст и стек соответственно тоже”). Переполнение стека может возникнуть в данном случае только в случае реального переполнения оперативной памяти машины, поэтому при такой реали-

зации стека нет необходимости вводить в функцию занесения нового элемента условие проверки на переполнение.

Для удаления элемента из начала односвязного списка и из стека, реализованного на односвязном списке, можно использовать следующую функцию.

```
void DeleteFirst(PNode & First)
{
    PNode p=First;
    if (p==NULL)
    {
        cout<<"Ошибка! В списке нет элементов!"<<endl;
        cout<<"Нажмите любую клавишу для продолже-
        ния"<<endl;
        getch();
    }

    else

    {
        First=First->Next;
        delete p;
        cout<<"Первый элемент списка удален"<<endl;
        cout<<"Нажмите любую клавишу для про-
        должения"<<endl;
        getch();
    }
}
```

В данной функции предусмотрена проверка списка (стека) на пустоту.

Структура данных типа односвязный список, помимо указателя начала списка, предусматривает наличие нескольких рабочих указателей, что значительно расширяет ее практическое применение. В частности, можно реализовать

функции его последовательного просмотра, вставки элемента в любое место списка, перестановки элементов путем несложных манипуляций с указателями. Если вставку осуществлять не в начало, а в конец списка, то с помощью односвязного списка реализуется очередь. Функция вставки элемента в очередь в этом случае будет выглядеть следующим образом:

```
void InsertLast(PNode& First, int Data)
{
    PNode p1, p2=First;
    if (First==0)
    {PNode p1=new TNode;
    p1->Data=Data;
    p1->Next=First;
    First=p1;
    cout<<"Список был пустым"<<endl;
    cout<<"Последний элемент списка добавлен и является одновременно первым"<<endl;
    cout<<"Нажмите любую клавишу для продолжения"<<endl;
    getch();
    }
    else
    {
        while (p2->Next!=NULL)
            p2=p2->Next;
        p1=new TNode;
        p1->Data=Data;
        p2->Next=p1;
        p1->Next=NULL;
        cout<<"Последний элемент списка добавлен"<<endl;
        cout<<"Нажмите любую клавишу для продолжения"<<endl;
    }
```



```

    getch();

    }

}

```

Здесь также предусмотрена проверка на пустоту, а также корректное занесение самого первого элемента.

Для удаления элемента из очереди можно использовать ту же функцию, что и для стека.

## **Задания**

### **Списковые структуры данных (односвязные очереди)**

1. Написать программу передвижения элемента на  $n$  позиций.
2. Создать копию списка.
3. Добавить элемент в начало списка.
4. Склеить два списка.
5. Удалить  $n$ -ый элемент из списка.
6. Вставить элемент после  $n$ -го элемента списка.
7. Создать список содержащий элементы общие для двух списков.
8. Упорядочить элементы в списке по возрастанию.
9. Удалить каждый второй элемент списка.
10. Удалить каждый третий элемент списка.
11. Упорядочить элементы списка по убыванию.
12. Очистить список.

### **Кольцевые списки**

13. Дан кольцевой список, содержащий 20 фамилий игроков футбольной команды. Разбить игроков на 2 группы по 10 человек. Во вторую группу попадает каждый 2-й человек.

14. Даны 2 кольцевых списка, содержащие фамилии спортсменов двух фехтовальных команд. Произвести жеребьевку. В первой команде выбирается каждый  $n$ -й игрок, а во второй - каждый  $m$ -й.

15. Задача Джозефуса:  $n$  воинов из одного войска убивают каждого  $m$ -го из другого. Требуется определить номер  $k$  начальной позиции воина, который должен будет остаться последним.

16. Даны 2 кольцевых списка, содержащие фамилии участников лотереи и наименования призов. Выигрывает  $N$  человек (каждый  $K$ -й). Число для пересчета призов -  $t$ . Вывести фамилии выигравших.

17. Даны 2 списка, содержащих фамилии учащихся и номера экзаменационных билетов. Число пересчета для билетов -  $E$ , для учащихся -  $K$ . Определить номера билетов, вытасканных учащимися.

18. Дан список, содержащий перечень товаров. Из элементов 1-го списка (товары изготовленные фирмой SONY) создать новый список.

19. Даны 2 списка, содержащие фамилии студентов 2-х групп. Перевести  $L$  студентов из 1-й группы во вторую. Число пересчета -  $K$ .

20. Даны 2 списка, содержащие перечень товаров, производимых концернами BOSH и FILIPS. Создать список товаров, выпускаемых как одной, так и другой фирмой.

21. Даны 2 списка, содержащие фамилии футболистов основного состава команды и запасного. Произвести  $K$  замен.

22. Даны 2 списка, содержащие фамилии солдат 1-го и 2-го взводов. Во время атаки  $M$  человек из 1-го взвода погибли. Произвести пополнение солдатами 2-го взвода.

23. Даны 2 списка, содержащие перечень товаров и фамилии покупателей. Каждый  $N$ -й покупатель покупает  $M$ -й товар. Вывести список покупок.

24. Даны 2 списка, содержащие наименования товаров, выпускаемых фирмами SONY и SHARP. Создать список конкурирующих между собой товаров.

**Составить отчет по лабораторной работе, и защитить его у преподавателя**



# РЕКУРСИВНЫЕ СТРУКТУРЫ ДАННЫХ

## Лабораторная работа №3 (4 часа). Бинарные деревья (создание и обход)

### Цель работы:

- исследовать и изучить основные процедуры, используемые при работе с бинарными (двоичными) деревьями;
- овладеть умениями и навыками написания программ по исследованию бинарных деревьев на языке программирования C++.

### Порядок выполнения работы

- ознакомиться с краткой теорией и примерами решения задач, относящихся к работе с бинарными деревьями;
- ответить на контрольные вопросы по знанию теории;
- получить задание на выполнение конкретного варианта лабораторной работы у преподавателя и разработать алгоритм решения задачи;
- написать и отладить программу решения задачи на языке C++;
- подготовить отчет по лабораторной работе и защитить его у преподавателя.

### Содержание отчета по ЛР

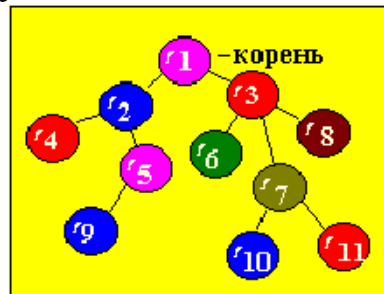
- наименование ЛР и ее цель;
- задание на ЛР согласно варианту;
- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;
- результаты работы программы.

### Краткая теория

Дерево - это нелинейная связанная структура данных, характеризующаяся следующими признаками:

- дерево имеет один элемент, на который нет ссылок от других элементов. Этот элемент, или "узел", называется корнем дерева;
- в дереве можно обратиться к любому элементу путем прохождения конечного числа ссылок (указателей);
- каждый элемент дерева связан только с одним предыдущим элементом.

Каждый узел дерева может быть промежуточным (элементы 2,3,5,7) либо терминальным ("листом" дерева) (элементы 4,9,10,11,8,6). Характерной особенностью терминальных узлов является отсутствие ветвей.



Элемент  $Y$ , находящийся непосредственно ниже элемента  $X$ , называется непосредственным потомком  $X$ , если  $X$  находится на уровне  $i$ , то говорят, что  $Y$  лежит на уровне  $i+1$ . Считается, что корень лежит на уровне 0.

Число непосредственных потомков элемента называется его степенью исхода, в зависимости от степени исхода узлов дерева классифицируют:

А. Если степень исхода узлов -  $M$ , то дерево называется  $M$ -арным ;

В. Если степень исхода узлов -  $M$  или 0, то - полное  $M$ -арное дерево;

С. Если степень исхода узлов дерева равна 2, то дерево называется бинарным ;

Д. Если степень исхода равна 2 или 0, то - полное бинарное дерево.

Особенно важную роль играют бинарные деревья, поэтому далее мы будем рассматривать их более подробно.

## Представление деревьев в памяти ЭВМ

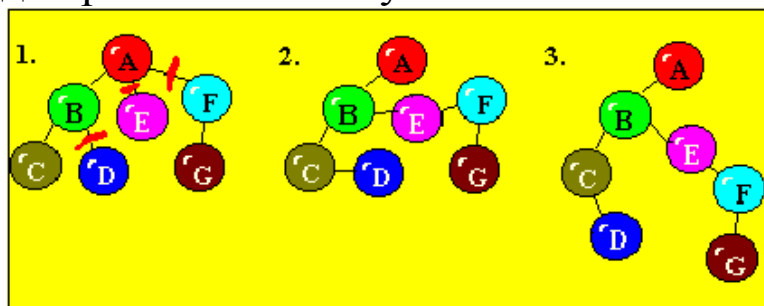
Деревья наиболее удобно представлять в памяти ЭВМ в виде связанных нелинейных списков. Элемент должен содержать *INFO*-поле, где содержится характеристика узла. Следующее поле определяет степень исхода узла и количество полей указателей равное степени исхода. Каждый указатель элемента ориентирует данный элемент-узел с его сыновьями. Узлы, в которые входят стрелки от исходного элемента, называются его сыновьями.

*INFO*-поле содержит два поля : поле записи (*rec*) и поле ключа (*key*). Ключ задается числом, по ключу определяют место элемента в дереве.



## Сведение *m*-арного дерева к бинарному

1. В каждом узле дерева отсекают все ветви, кроме крайних левых, соответствующих старшим сыновьям;
2. Соединяют горизонтальными линиями сыновей одного родителя (узла);
3. Старшим сыном в каждом узле полученной структуры будет узел под обрабатываемым узлом.



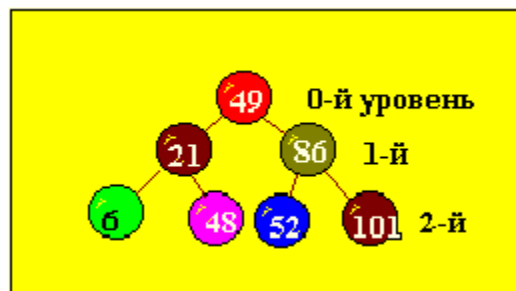
Построение бинарных деревьев. Согласно представлению деревьев в памяти ЭВМ каждый элемент (узел бинарного де-

рева) будет записью, содержащей четыре поля. Значением этих полей будут соответственно

- ключ записи,
- ссылка
  - на элемент влево-вниз,
  - на элемент вправо-вниз и
  - на текст записи.

При построении необходимо помнить, что левый сын имеет ключ меньше чем отец (родитель). Значение ключа правого сына больше значения ключа отца.

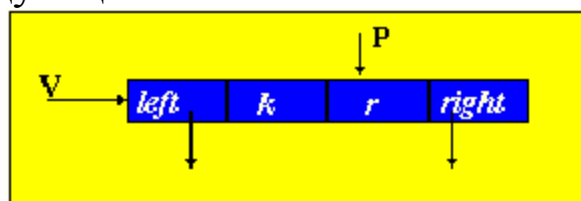
Если узлы дерева имеют значения 6, 21, 48, 49, 52, 86, 101, то бинарное дерево может иметь вид, изображенный на рисунке ниже. Это упорядоченное бинарное дерево с минимальным числом уровней. Идеально сбалансированное дерево - это дерево, в котором левое и правое поддеревья имеют уровни, отличающиеся не больше, чем на единицу.



## Алгоритмы

### Алгоритм создания бинарного дерева (псевдокод)

Для построения дерева необходимо создать в памяти ЭВМ элемент следующего типа:



$P, Q$  - рабочие указатели

$V = \text{maketree}(\text{key}, \text{rec})$  - процедура, которая создает сегмент ключа и записи

$P = \text{getnode}$  - генерация нового элемента

$r = \text{rec}$

$k = \text{key}$

$V = P$

$\text{left} = \text{nil}$

$\text{right} = \text{nil}$

$\text{tree}$  - указатель на корень дерева

Введем сначала первое значение ключа, потом процедурой  $\text{maketree}$  сгенерируем сам элемент узла дерева. Далее идем по циклу до тех пор, пока указатель не передвинется на нулевое значение.

$\text{read}(\text{key}, \text{rec})$

$\text{tree} = \text{maketree}(\text{key}, \text{rec})$

$\text{while not eof do}$

$p = \text{tree}$

$q = \text{tree}$

$\text{read}(\text{key}, \text{rec})$

$v = \text{maketree}(\text{key}, \text{rec})$

$\text{while } p \neq \text{nil do}$

$q = p$

$\text{if } \text{key} < k(p) \text{ then}$

$p = \text{left}(p)$

$\text{else}$

$p = \text{right}(p)$

$\text{endif}$

$\text{endwhile}$

$\text{if } p = \text{nil} \text{ then writeln}(\text{'Это корень'})$

$\text{tree} = v$

$\text{else}$

$\text{if } \text{key} < k(q) \text{ then}$

$\text{left}(q) = v$

$\text{else}$

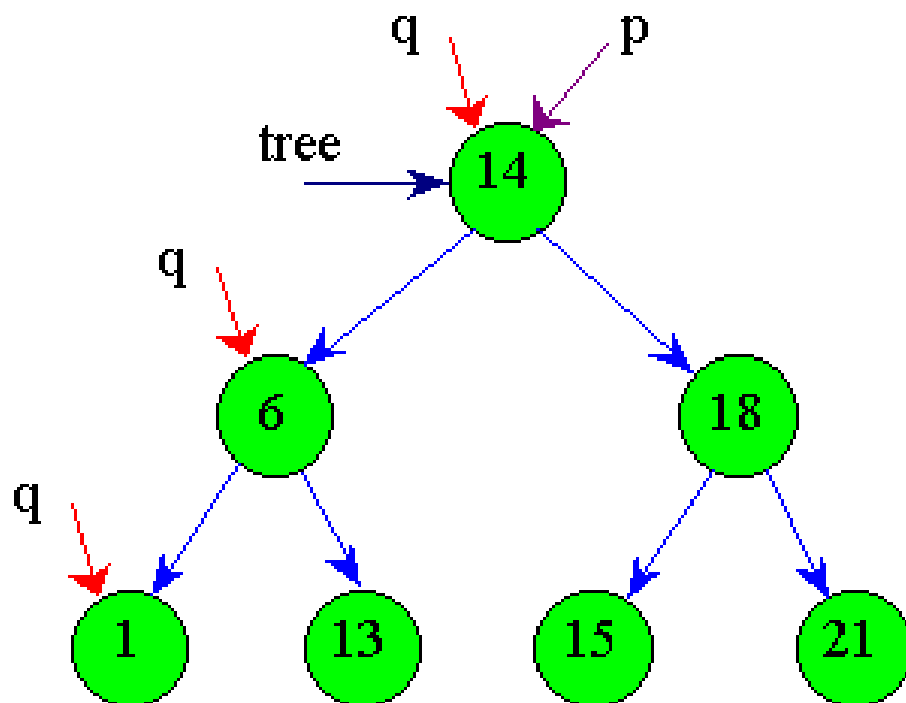


```

        right(q) = v
    endif
    endif
endwhile
return

```

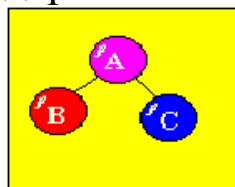
Если по этому алгоритму строить упорядоченное бинарное дерево, то при поступлении ключей в последовательности 14, 18, 6, 21, 1, 13, 15 получим дерево, изображенное на рисунке ниже.



Для того, чтобы просмотреть элементы созданного дерева, необходимо выполнить его обход.

#### Алгоритм "обхода" дерева

Пусть задано бинарное дерево:



Существуют три принципа обхода бинарных деревьев. Рассмотрим их на примере заданного дерева:

1) Обход сверху вниз (корень посещается ранее, чем поддеревья): A, B, C;

2) Слева направо: B, A, C;

3) Снизу вверх (корень посещается после поддеревьев): B, C, A.

Наиболее часто применяется 2-ой способ, узлы посещаются в порядке возрастания значения их ключа.

#### Рекурсивные процедуры обхода дерева:

1) *PROCEDURE pretrave (tree)*

*IF tree<>nil*

*THEN PRINT info (tree)*

*tree=left (pretrave (tree))*

*tree=right (pretrave (tree))*

*END IF*

*RETURN*

2) *PROCEDURE intrave (tree)*

*IF tree<>nil*

*THEN tree=left (intrave (tree))*

*PRINT info (tree)*

*tree=right (intrave (tree))*

*END IF*

*RETURN*

3) *PROCEDURE postrave (tree)*

*IF tree<>nil*

*THEN tree=left (postrave (tree))*

*tree=right (postrave (tree))*

*PRINT info (tree)*

*END IF*

*RETURN*

**Контрольные вопросы по теории (бинарные деревья /создание и обход/)**

1. Для включения новой вершины в дерево нужно найти узел, к которому её можно присоединить. Узел будет найден, если очередной ссылкой, определяющей ветвь дерева, в которой надо продолжать поиск, окажется ссылка:

- $p = \text{right}(p)$ ;
- $p = \text{nil}$ ;
- $p = \text{left}(p)$ .

2. Для написания процедуры над двумя деревьями необходимо описать элемент типа запись, который содержит поля:

- Element=Запись  
Left, Right : Указатели  
Rec : Запись;

- Element=Запись  
Left : Указатель  
Key : Ключ  
Rec : Запись;

- Element=Запись  
Left, Right : Указатели  
Key : Ключ  
Rec : Запись.

3. В памяти ЭВМ бинарное дерево удобно представлять в виде:

- связанных линейных списков;
- массивов;
- связанных нелинейных списков.

4. Элемент  $t$ , на который нет ссылок:

- корнем;
- промежуточным;
- терминальным (лист).

5. Дерево называется полным бинарным, если степень исходов вершин равна:

- 2 или 0;
- 2;

- $m$  или 0;
- $m$ .

## Примеры алгоритмов конкретных задач

Рассмотрим, как операции создания бинарного сбалансированного дерева и его обхода реализуются на языке программирования C++. Для упрощения примера будем использовать дерево, поле записи которого содержит только ключи, являющиеся вещественными числами, тогда элемент дерева описывается следующим образом:

```
struct element
{ double k;
  element* left;
  element* right;
};
```

Далее необходимо ввести указатель вершины дерева и рабочие указатели для реализации функции создания дерева.

```
element *tree=NULL,*P,*Q;
```

Пока дерево не создано, указатель корня нулевой (\*tree=NULL).

Ниже представлена функция создания бинарного дерева

```
void maketree(double a) //создание дерева
{ if(!tree)
  { tree=new element;
    tree->k=a;
    tree->right=tree->left=NULL;
    P=tree;
    Q=NULL;
    return;
  };
  P=Q=tree;
  while (P!=NULL)
```

```

{ Q=P;
  if(a<P->k) P=P->left; else P=P->right;
};
if(a<Q->k) { Q->left=new element;
            Q->left->k=a;
            Q->left->left=Q->left->right=NULL;
          }
else      { Q->right=new element;
            Q->right->k=a;
            Q->right->left=Q->right->right=NULL;
          };
}

```

В теоретической части лабораторной работы были рассмотрены 3 вида обхода бинарного дерева. Алгоритм наиболее часто используемого вида обхода слева-направо на C++ для созданного выше дерева будет иметь следующий вид:

```

void printtree(element* tree) //просмотр и печать дерева
{ if(tree)
  { printtree(tree->left);
    cout<<tree->k<<" ";
    printtree(tree->right);
  };
}

```

Теперь рассмотрим листинг готовой программы, которая позволяет создать описанное выше бинарное и выполнить его обход слева направо.

```

//Листинг программы на C++.
#include<iostream.h>
#include<conio.h>
struct element

```

```

{ double k;
  element* left;
  element* right;
};
element *tree=NULL,*P,*Q;
void maketree(double a) //создание дерева
{ if(!tree)
  { tree=new element;
    tree->k=a;
    tree->right=tree->left=NULL;
    P=tree;
    Q=NULL;
    return;
  };
  P=Q=tree;
  while (P!=NULL)
  { Q=P;
    if(a<P->k) P=P->left; else P=P->right;
  };
  if(a<Q->k) { Q->left=new element;
    Q->left->k=a;
    Q->left->left=Q->left->right=NULL;
  }
  else { Q->right=new element;
    Q->right->k=a;
    Q->right->left=Q->right->right=NULL;
  };
}
void printree(element* tree) //просмотр и печать дерева
{ if(tree)
  { printree(tree->left);
    cout<<tree->k<<" ";
    printree(tree->right);
  };
};

```

```

}
void main()
{ clrscr();
double a,key;
cout<<"Введите элементы дерева , 0 - конец ввода:
";cin>>a;
while(a)
{ maketree(a);
cin>>a;
};
printtree(tree);
getch();
}

```

Данный листинг позволяет создавать бинарное дерево с вещественными ключами до того момента, пока мы не прервем процесс ввода его элементов, после чего выполняет обход дерева слева направо с выдачей ключей на экран.

На практике часто может возникать необходимость добавления и удаления элементов в уже существующее дерево, что не является возможным при использовании вышеописанной программы. Функции поиска, вставки и удаления элементов из бинарного дерева будут рассмотрены в лабораторной работе № 6.

### **Задания**

1. Описать процедуру или функцию, которая:
  - а) присваивает параметру  $E$  запись из самого левого листа непустого дерева  $T$  (лист-вершина, из которого не выходит ни одной ветви);
  - б) определяет число вхождений записи  $E$  в дерево  $T$ .

2. Вершины дерева вещественные числа. Описать процедуру или функцию, которая:

а) вычисляет среднее арифметическое всех вершин дерева;

б) добавляет в дерево вершину со значением, вычисленным в предыдущей процедуре (функции).

3. Записи вершин дерева - вещественные числа. Описать процедуру, которая выбирает все вершины с отрицательными записями и строит из них новое дерево.

4. Записи вершин дерева - вещественные числа. Описать процедуру или функцию, которая:

а) находит максимальное или минимальное значение записей вершин непустого дерева;

б) печатает записи из всех листьев дерева.

5. Описать процедуру или функцию, которая:

а) определяет, входит ли вершина с записью  $E$  в дерево  $T$ ;

б) если такая запись не найдена, то она добавляется.

6. Описать процедуру или функцию, которая:

а) находит в непустом дереве  $T$  длину (число ветвей) пути от корня до ближайшей вершины с записью  $E$ ; если  $E$  не входит в  $T$ , то за ответ принять - 1.

б) определяет максимальную глубину непустого дерева  $T$ , т.е. число ветвей в самом длинном из путей от корня дерева до листьев.

7. Описать процедуру  $COPY(T, T1)$ , которая строит  $T1$  - копию дерева  $T$ .

8. Описать процедуру  $EQUAL(T1, T2)$ , проверяющую на равенство деревья  $T1$  и  $T2$  (деревья равны, если ключи и записи вершин одного дерева равны соответственно ключам и записям другого дерева).

9. Описать процедуру, которая:

а) присваивает переменной  $b$  типа *char* значение:

$K$  - если вершина - корень,

$\Pi$  - если вершина - промежуточная вершина,



*Л* - если вершина - лист;

б) распечатывает атрибуты всех вершин дерева.

10. Описать процедуру, которая:

а) находит максимальное или минимальное значение записей листьев непустого дерева;

б) добавляет элемент с данной записью в дерево.

11. Описать процедуру или функцию, которая:

а) вставляет узел с записью *Е* в дерево, если ранее такой не было;

б) считает и выдает на экран сумму значений всех ключей, если такая запись есть.

12. Описать процедуру или функцию, которая:

а) печатает запись, встречающуюся в дереве один раз;

б) печатает запись, встречающееся в дереве максимальное число раз.

**Составить отчет по лабораторной работе, и защитить его у преподавателя**

## **ПОИСК**

### **Лабораторная работа №4 (4 часа). Исследование методов линейного и бинарного поиска**

#### **Цель работы:**

- изучить методы линейного, бинарного и индексно-последовательного поиска.
- овладеть навыками написания программ для методов линейного, бинарного и индексно-последовательного поиска на языке программирования C++.

#### **Порядок выполнения работы**

- ознакомиться с краткой теорией и примерами решения задач, относящихся к исследованию методов линейного, бинарного и индексно-последовательного поиска;
- ответить на контрольные вопросы и получить оценку по знанию теории;
- получить задание на выполнение конкретного варианта лабораторной работы и выполнить его;
- написать и отладить программу решения задачи на языке C++;
- составить отчет по лабораторной работе и защитить его у преподавателя.

#### **Содержание отчета по ЛР**

- наименование ЛР и ее цель;
- задание на ЛР согласно варианту;
- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;
- результаты работы программы.

#### **Краткая теория**

Поиск – это действие наиболее часто встречающееся в программировании. Он же представляет собой идеальную задачу, на которой можно испытывать различные структуры данных по мере их появления. Существует несколько основных "вариаций этой темы", и для них создано много различных алгоритмов. При дальнейшем рассмотрении мы исходим из такого принципиального допущения: группа данных, в которой необходимо отыскать заданный элемент, фиксирована. Будем считать, что множество из  $N$  элементов задано, скажем, в виде такого массива

a: ARRAY[0.. $N-1$ ] OF item

Обычно тип *item* описывает запись с некоторым полем, выполняющим роль ключа. Задача заключается в поиске элемента, ключ которого равен заданному "аргументу поиска"  $x$ . Полученный в результате индекс  $i$ , удовлетворяющий условию  $a[i].key = x$ , обеспечивает доступ к другим полям обнаруженного элемента. Так как нас интересует в первую очередь сам процесс поиска, а не обнаруженные данные, то мы будем считать, что тип *item* включает только ключ, т.е. он есть ключ (*key*).

## Алгоритмы

### Линейный поиск

Если нет никакой дополнительной информации о разыскиваемых данных, то очевидный подход - простой последовательный просмотр массива с увеличением шаг за шагом той его части, где желаемого элемента не обнаружено. Такой метод называется линейным поиском. Условия окончания поиска таковы:

1. Элемент найден, т.е.  $a[i] = x$ .
2. Весь массив просмотрен и совпадения не обнаружено.

Это дает нам линейный алгоритм:

$i := 0$ ;

```

    WHILE ( $i < N$ ) AND ( $a[i] \neq x$ ) DO
         $i := i + 1$  ;
    END;

```

Обратите внимание, что порядок элементов в логическом выражении имеет существенное значение. Инвариант цикла, т.е. условие, выполняющееся перед каждым увеличением индекса  $i$ , выглядит так:

$$(0 \leq i < N) \text{ AND } (A_k : 0 \leq k < i : a_k \neq x)$$

Он говорит, что для всех значений  $k$ , меньших чем  $i$ , совпадения не было. Отсюда и из того факта, что поиск заканчивается только в случае ложности условия в заголовке цикла, можно вывести окончательное условие его окончания:

$$((i = N) \text{ OR } (a_i = x)) \text{ AND } (A_k : 0 \leq k < i : a_k \neq x)$$

Это условие не только указывает на желаемый результат, но из него же следует, что если элемент найден, то он найден вместе с минимально возможным индексом, т.е. это первый из таких элементов. Равенство  $i = N$  свидетельствует, что совпадения не существует.

Совершенно очевидно, что окончание цикла гарантировано, поскольку на каждом шаге значение  $i$  увеличивается, и, следовательно, оно, конечно же, достигнет за конечное число шагов предела  $N$ ; фактически же, если совпадения не было, это произойдет после  $N$  шагов.

Ясно, что на каждом шаге требуется увеличивать индекс и вычислять логическое выражение. А можно ли эту работу упростить и таким образом убыстрить поиск ?

Единственная возможность - попытаться упростить само логическое выражение, ведь оно состоит из двух членов. Следовательно, единственный шанс на пути к более простому решению - сформулировать простое условие, эквивалентное нашему сложному. Это можно сделать, если мы гарантируем, что совпадение всегда произойдет. Для этого достаточно в конец массива поместить дополнительный элемент со значением  $x$ . Назовем такой вспомогательный элемент "барьером".

ером", ведь он охраняет нас от перехода за пределы массива. Теперь массив будет описан так:

*a: ARRAY[0..N] OF INTEGER*

и алгоритм линейного поиска с барьером выглядит следующим образом:

```
a[N] := x;  
i := 0;  
WHILE a[i] <> x DO  
  i := i+1;  
END;
```

Результирующее условие, выведенное из того же инварианта, что и прежде:

$(a_i = x) \text{ AND } (A_k : 0 \leq k < i : a_k \neq x)$

Ясно, что равенство  $i = N$  свидетельствует о том, что совпадения (если не считать совпадения с барьером) не было.

Поиск делением пополам (двоичный поиск).

Совершенно очевидно, что других способов убыстрения поиска не существует, если, конечно, нет еще какой-либо информации о данных, среди которых идет поиск. Хорошо известно, что поиск можно сделать значительно более эффективным, если данные будут упорядочены. Вообразите себе телефонный справочник, в котором фамилии не будут расположены по порядку. Это нечто совершенно бесполезное! Поэтому мы приводим алгоритм, основанный на знании того, что массив  $a$  упорядочен, т.е. удовлетворяет условию

$A_k : 1 \leq k < N : a_{k-1} \neq a_k$

Основная идея - выбрать случайно некоторый элемент, предположим  $a[m]$ , и сравнить его с аргументом поиска  $x$ . Если он равен  $x$ , то поиск заканчивается, если он меньше  $x$ , то мы заключаем, что все элементы с индексами, меньшими или равными  $m$ , можно исключить из дальнейшего поиска; если же он больше  $x$ , то исключаются индексы больше и равные  $m$ . Это соображение приводит нас к следующему алгоритму (он называется "поиском делением пополам"). Здесь две индексные переменные  $L$  и  $R$  отмечают соответственно

левый и правый конец секции массива  $a$ , где еще может быть обнаружен требуемый элемент.

```
L := 0;  
R := N-1;  
found := FALSE;  
WHILE (L < R) AND NOT found DO  
    m := любое значение между L и R;  
    IF a[m] = x THEN found := TRUE;  
    IF a[m] < x THEN L := m+1  
    ELSE R := m-1;  
    ENDIF;  
ENDWHILE;
```

Инвариант цикла, т.е. условие, выполняющееся перед каждым шагом, таков:

$$(L \leq R) \text{ AND } (A_k : 0 \leq k < L : a_k < x) \text{ AND } (A_k : R < k < N : a_k > x)$$

из чего выводится результат

$$\textit{found} \text{ OR } ((L > R) \text{ AND } (A_k : 0 \leq k < L : a_k < x) \text{ AND } (A_k : R < k < N : a_k > x))$$

откуда следует

$$(a_m = x) \text{ OR } (A_k : 0 \leq k < N : a_k \neq x)$$

Выбор  $m$  совершенно произволен в том смысле, что корректность алгоритма от него не зависит. Однако на его эффективность выбор влияет. Ясно, что наша задача - исключить на каждом шагу из дальнейшего поиска, каким бы ни был результат сравнения, как можно больше элементов. Оптимальным решением будет выбор среднего элемента, так как при этом в любом случае будет исключаться половина массива. В результате максимальное число сравнений равно  $\log N$ , округленному до ближайшего целого. Таким образом, приведенный алгоритм существенно выигрывает по сравнению с линейным поиском, ведь там ожидаемое число сравнений -  $N/2$ .

Эффективность можно несколько улучшить, поменяв местами заголовки условных операторов. Проверку на равен-

ство можно выполнять во вторую очередь, так как она встречается лишь единожды и приводит к окончанию работы. Но более существенно следующее соображение: нельзя ли, как и при линейном поиске, отыскать такое решение, которое опять бы упростило условие окончания. И мы действительно находим такой быстрый алгоритм, как только отказываемся от наивного желания закончить поиск при фиксации совпадения. На первый взгляд это кажется странным, однако при внимательном рассмотрении обнаруживается, что выигрыш в эффективности на каждом шаге превосходит потери от сравнения с несколькими дополнительными элементами. Напомним, что число шагов в худшем случае -  $\log N$ . Быстрый алгоритм основан на следующем инварианте:

$$(A_k : 0 \leq k < L : a_k < x) \text{ AND } (A_k : R \leq k < N : a_k \geq x)$$

причем поиск продолжается до тех пор, пока обе секции не "накроют" массив целиком.

$L := 0;$

$R := N;$

*WHILE*  $L < R$  *DO*

$m := (L+R) \text{ DIV } 2;$

*IF*  $a[m] < x$  *THEN*  $L := m+1$

*ELSE*  $R := m ;$

*END*

*END*

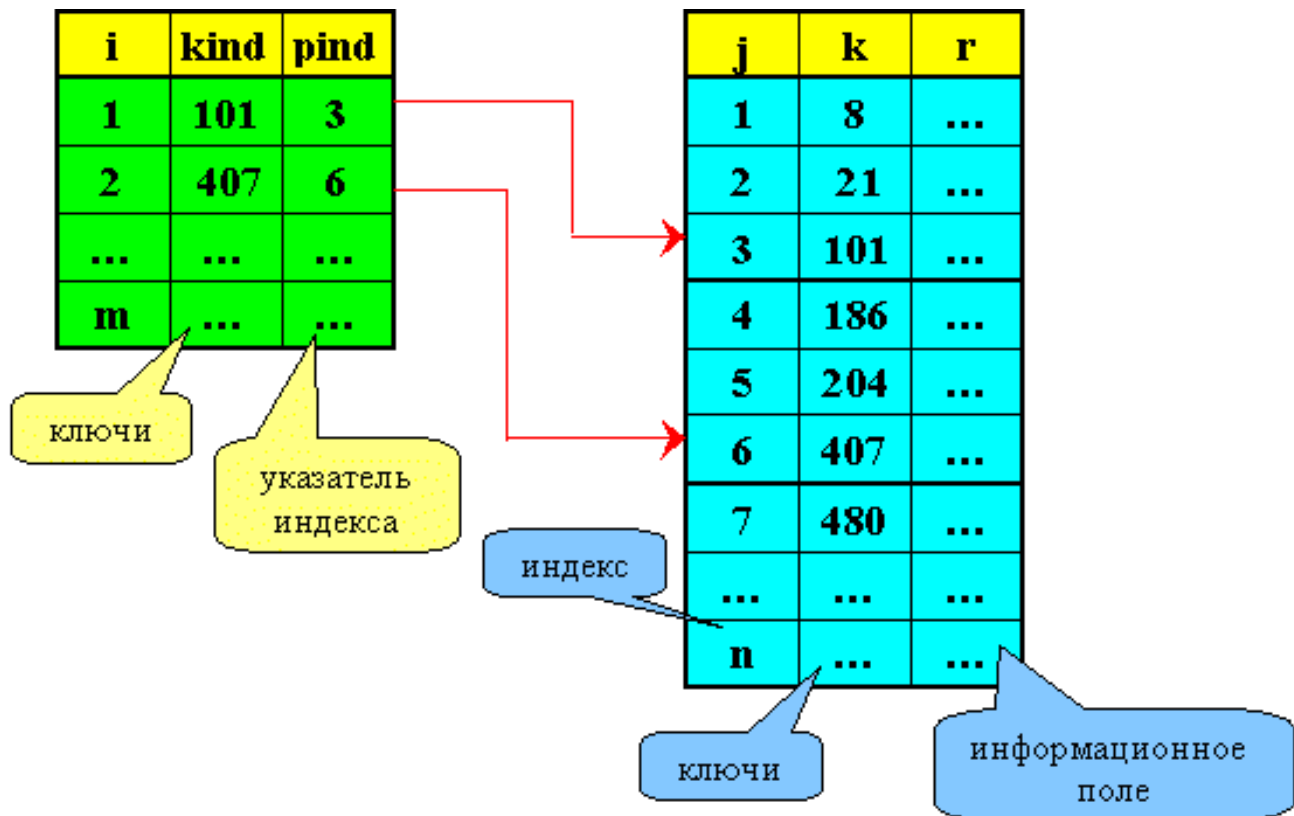
Условие окончания -  $L < R$ , но достижимо ли оно? Для доказательства этого нам необходимо показать, что при всех обстоятельствах разность  $R-L$  на каждом шаге убывает. В начале каждого шага  $L < R$ . Для среднего арифметического  $m$  справедливо условие  $L < m < R$ . Следовательно, разность действительно убывает, ведь либо  $L$  увеличивается при присваивании ему значения  $m+1$ , либо  $R$  уменьшается при присваивании значения  $m$ . При  $L = R$  повторение цикла заканчивается. Однако наш инвариант и условие  $L = R$  еще не свидетельствуют о совпадении. Конечно, при  $R = N$  никаких совпадений нет. В других же случаях мы должны учитывать, что

элемент  $a[R]$  в сравнениях никогда не участвует. Следовательно, необходима дополнительная проверка на равенство  $a[R] = x$ . В отличие от первого нашего решения приведенный алгоритм, как и в случае линейного поиска, находит совпадающий элемент с наименьшим индексом.

Еще одним вариантом убыстрения поиска в случае упорядоченности данных является индексно-последовательный поиск. При таком поиске организуется две таблицы: таблица данных со своими ключами - упорядоченных по возрастанию, и таблица **индексов**, которая тоже состоит из ключей данных, но эти ключи взяты из основной таблицы через определенный интервал. Другими словами, при прохождении упорядоченной таблицы мы сравниваем с ключом элементы не последовательно, а через определенный интервал, то есть задаем некоторый шаг поиска. Когда на очередном шаге поиска предыдущий элемент меньше значения ключа, а следующий элемент больше значения ключа, то устанавливаются соответственно нижняя  $low$  ( $kind < key$ ) и верхняя  $hi$  ( $kind > key$ ) границы в основной таблице. Между этими границами по основной таблице будет соответственно произведен уже обычный последовательный поиск.

Таблицы индексно-последовательного поиска





В псевдокоде алгоритм индексно-последовательного поиска следующий:

```

i = 1
while (i <= m) and (kind(i) <= key) do
    i=i+1
endwhile
if i = 1 then low = 1
    else low = pind(i-1)
endif
if i = m+1 then hi = n
    else hi = pind(i)-1
endif
for j = low to hi
    if key = k(j) then
        search = j
    
```

```

        return
    endif
next j
search = 0
return

```

Эффективность данного вида поиска величина  $O(\sqrt{n})$ . Данный вид поиска, также как и бинарный, может давать значительный эффект при больших размерах таблиц поиска. Рекомендованный шаг для  $n$  элементов таблицы – приблизительно  $\sqrt{n}$ . В принципе, для достижения наибольшей эффективности поиска при решении конкретных задач пользователь может задавать какой угодно шаг.

#### 4.5. Контрольные вопросы по теории (линейного и бинарного поиска)

1. Где эффективен линейный поиск?
  - в списке;
  - в массиве;
  - в массиве и в списке.
2. Какой поиск эффективнее?
  - линейный;
  - бинарный;
  - без разницы.
3. В чём суть бинарного поиска ?
  - нахождение элемента массива  $x$  путём деления массива пополам каждый раз, пока элемент не найден;
  - нахождение элемента  $x$  путём обхода массива;
  - нахождение элемента массива  $x$  путём деления массива.
4. Как расположены элементы в массиве бинарного поиска ?
  - по возрастанию;
  - хаотично;
  - по убыванию.

## 5. В чём суть линейного поиска?

- производится последовательный просмотр от начала до конца и обратно через 2 элемента;
- производится последовательный просмотр элементов от середины таблицы;
- производится последовательный просмотр каждого элемента.

### Примеры алгоритмов конкретных задач

Теперь рассмотрим листинги программ на языке C++, реализующие рассмотренные выше виды поиска.

В примерах таблицы поиска содержат только целочисленный ключ, а поля данных отсутствуют. Таблицы поиска задаются в виде одномерных массивов целых чисел (ключей записей).

Функция поиска для элемента по совпадению в неупорядоченной таблице возвращает индекс элемента либо -1 в случае отсутствия искомого элемента и имеет следующий вид:

```
int poisk1(int A[],int n,int key)
{ int j = 0;
  while (A[j] != key && j < n-1)
    j++;
  return (A[j] == key) ? j : -1;
}
```

По данному алгоритму на каждом выполняется 2 сравнения.

Пример программы с использованием данной функции будет следующий:

```

/* ПОИСК ПО СОВПАДЕНИЮ КЛЮЧА */
#include <stdio.h>
#include <conio.h>
#define m 10
int poisk1(int A[],int n,int key);
/*
=====
*/

int main()
{
int i,key,ind, A[m];
printf(" Поиск по совпадению ключа ");
printf("\n Введите %d целых чисел \n",m);
for (i=0; i<m; i++)
scanf("%d",&A[i]);
printf("\n Введите значение искомого ключа =>");
scanf("%d",&key);

printf("\n Поиск элемента по совпадению ");
printf("\n Исходный массив:\n");
for(i=0; i<m; i++)
printf("%5d",A[i]);
ind= poisk1(A,m,key);
if (ind == -1)
printf("\nЭлемент %d не найден\n",key);
else printf("\n Элемент %d имеет индекс %d\n",key,ind);
getch(); return 0;

```

```

    }
    /*
=====
== */

    /* ФУНКЦИЯ ПОИСКА ЭЛЕМЕНТА ПО СОВПАДЕНИЮ
*/

    int poisk1(int A[],int n,int key)
    { int j = 0;
      while (A[j] != key && j < n-1)
        j++;
      return (A[j] == key) ? j : -1;
    }
    /*
***** */

```

Как было сказано выше, для улучшения алгоритма поиска можно использовать заграждающий элемент или барьер. В этом случае последняя запись таблицы запоминается, а после завершения поиска восстанавливается в таблице. В последний элемент массива заносится ключ поиска, и образуется так называемый заграждающий элемент.

Теперь на каждом шаге поиска осуществляется только одно сравнение, а сам поиск продолжается до нахождения элемента с заданным ключом. Если искомого элемента в исходной таблице не было, поиск закончится на барьере. Таким образом, использование барьера в случае числовых ключей существенно сокращает число сравнений. Если ключом поиска является символьная строка, то использование заграждающего элемента вряд ли оправдано.

Приведем пример программы последовательного поиска на C++ с использованием барьера.

```

/* ****
*/

/* ПОИСК ПО СОВПАДЕНИЮ КЛЮЧА - ЗАГРАЖДАЮ-
ЩИЙ ЭЛЕМЕНТ */

#include <stdio.h>
#include <conio.h>
#define m 10
int poisk2(int A[],int n,int key);
main()
{
    int i,key,ind, A[m];
    printf(" Поиск по совпадению ключа ");
    printf("\n Введите %d целых чисел \n",m);
    for (i=0; i<m; i++)
        scanf("%d",&A[i]);
    printf("\n Введите значение искомого ключа =>");
    scanf("%d",&key);

    printf("\n Поиск элемента по совпадению ");
    printf("\n Исходный массив:\n");
    for(i=0; i<m; i++)
        printf("%5d",A[i]);
    ind= poisk2(A,m,key);
    if (ind == -1)
        printf("\nЭлемент %d не найден\n",key);
    else printf("\n Элемент %d имеет индекс %d\n",key,ind);
    getch(); return 0;
}

```

```

    }
    /*
===== */
    /* ПОИСК ЭЛЕМЕНТА ПО СОВПАДЕНИЮ С ЗАГРАЖ-
ДАЮЩИМ ЭЛЕМЕНТОМ */
    int поиск2(int A[],int n,int k)
    { int i = 0;
      A[n] = k;
      while (A[i] != k )
        i++;
      return (i == n) ? -1 : i;
    }
    /*
*****
*/

```

Как было сказано выше, в случае, если последовательность упорядочена по возрастанию (убыванию), можно использовать более эффективные методы поиска. Приведем листинг примера программы, которая осуществляет бинарный поиск в упорядоченной по возрастанию последовательности чисел.

```

    /*
*****
*/

    /* БИНАРНЫЙ ПОИСК В УПОРЯДОЧЕННОЙ ТАБЛИЦЕ
*/

#include <stdio.h>
#include <conio.h>
#define m 10

```

```

int bisearch(int A[],int n,int key);
int bisearch1(int A[],int n,int key);
/*
=====
*/

/* ГЛАВНАЯ ФУНКЦИЯ */
main()
{ int i,key, A[m];
  printf(" Бинарный поиск в упорядоченной таблице ");
  printf("\n Введите %d целых чисел в возрастающем по-
рядке\n",m);
  for (i=0; i<m; i++)
    scanf("%d",&A[i]);
  printf("\n Введите значение искомого ключа =>");
  scanf("%d",&key);
  i = bisearch1(A,m,key);
  if (i == -1)
    printf(" Ключ %d не найден",key);
  else
    printf(" %d-й элемент имеет ключ = %d \n",i,A[i]);
  getch();
  return 0;
}
/*
=====
= */

/* ФУНКЦИЯ БИНАРНОГО ПОИСКА В УПОРЯДОЧЕН-
НОЙ ТАБЛИЦЕ Вариант 1 */

```



```

int bisearch1(int A[],int n,int key)
{ int li,rj,k;
  li=0; rj=n-1;
while ( li <= rj )
  { k = (li+rj)/2;
    if (key > A[k])
      li = k+1;
    else
      if (key < A[k])
        rj = k-1;
    else return k;
    printf(" li=%d, rj=%d, k=%d ",li,rj,k);/* отладочная
печать*/
  }
  return -1;
}
/*
=====
== */

```

*/\* ФУНКЦИЯ БИНАРНОГО ПОИСКА Вариант 2,*

*оба варианта равноценны \*/*

```

int bisearch(int A[],int n,int key)
{ int li,rj,k;
  li=0; rj=n-1; k = li;
while ( li <= rj && A[k] != key )
  { k = (li+rj)/2;
    if (key > A[k])
      li = k+1;

```

```

        else
            if (key < A[k])
                rj = k-1;
            }
        return (A[k] == key) ? k : -1;
    }
    /*
    *****/

```

Еще одним эффективным методом поиска в упорядоченной таблице данных является индексно-последовательный поиск. Рассмотрим листинг программы, реализующей данный вид поиска в упорядоченной таблице данных.

```

/*ИНДЕКСНО-ПОСЛЕДОВАТЕЛЬНЫЙ ПОИСК*/
#include <iostream.h>
#include <conio.h>
struct index
{
    int kind; int pind;
};
int poisk(int *mas) /*Функция индексно-последовательного
поиска*/
{
    const int n = 10;
    int step, m;
    cout<<"\n Введите шаг поиска \n";
    cin>>step;

```

```

m = n/step;
int key;
cout<<"Введите ключ поиска"<<endl;
cin>>key;
index* masindex = new index [m];
int i =0;
int j =step-1;
int search;
while ((i<m)&&(mas[j]<=key)&&(j<n))
    {
        /*masindex[i].idx=i;*/
        masindex[i].kind=mas[j]; masindex[i].pind=j;
        if (masindex[i].kind == key)
            {
                search = j; delete mas; return search;
            }
        j=j+step; i++;
    }
int hi, low;
if (i==0)
    low = 0;
else
    low = masindex[i-1].pind;
if (i == m)
    hi =n-1;
else
    hi = j - 1;

```

```

    cout<<"Значение LOW = "<<low<<endl;
    cout<<"Значение HIGH = "<<hi<<endl;

    for (int z = low;z<=hi; z++)
    {
        if(key == mas[z])
        {
            search =z;
            delete mas;
            return search;
        }
    }
    search=-1;
    delete mas;
    return search;
}

void main()
{
    clrscr();
    const int n = 10;
    int mas[n];
    cout <<" ----Ввод элементов массива в возрастающем по-
рядке-----"<<endl;
    for(int j = 0; j<n;j++)
        {cout <<" Введите элемент массива с индексом
"<<j<<endl; cin>>mas[j];}

```

```

int result;
result = poisk(mas);
if (result == -1)
    cout<<"\n Таких элементов нет!\n";
else
    cout<<"Найденный элемент имеет индекс =
"<<result<<endl;
    getch();
}

```

Следует обратить внимание, что в данном листинге для обеспечения ввода шага поиска пользователем используется динамический массив.

### Задания

1. Найти наименьший элемент в упорядоченном массиве *A* с помощью линейного поиска. Найти данный элемент с помощью бинарного и индексно-последовательного поиска.

2. Найти элементы в упорядоченном массиве *A*, которые больше 30, с помощью линейного поиска. Найти данные элементы с помощью бинарного и индексно-последовательного поиска.

3. Вывести на экран все числа массива *A* кратные 3 (3,6,9,...) с помощью линейного поиска. Найти данные элементы с помощью бинарного и индексно-последовательного поиска.

4. Найти все элементы, модуль которых больше 20 и меньше 50 в упорядоченной таблице, с помощью с помощью линейного поиска. Найти данные элементы с помощью бинарного и индексно-последовательного поиска.

5. Вывести на экран все числа упорядоченного массива *A* кратные 4 (4,8,...) с помощью помощью линейного поиска.

Найти данные элементы с помощью бинарного и индексно-последовательного поиска.

6. Вывести на экран сообщение, каких чисел больше относительно 50 в упорядоченной таблице с помощью линейного поиска. Найти числа, большие 50 с помощью бинарного и индексно-последовательного поиска.

7. Найти элемент в упорядоченном массиве  $A$  и найти число сравнений помощью линейного, бинарного и индексно-последовательного поиска.

8. Поиск элементов случайным образом помощью линейного, бинарного и индексно-последовательного поиска.

9. Дан список номеров машин (345, 368, 876, 945, 564, 387, 230), найти, на каком месте стоит машина с заданным номером, с помощью линейного поиска. Расположить данные номера в порядке возрастания и решить предыдущую задачу с помощью бинарный и индексно-последовательного поиска.

10. Поиск каждого кратного двум элемента в упорядоченном массиве помощью линейного поиска. Найти данные элементы с помощью бинарного и индексно-последовательного поиска.

11. Найти элемент с ключом, равным сумме индексов упорядоченного массива  $A$  с помощью линейного, бинарного и индексно-последовательного поиска.

**Составить отчет по лабораторной работе и защитить его у преподавателя**

## **Лабораторная работа №5 (4 часа). Исследование методов оптимизации поиска**

### **Цель работы:**

- исследовать и изучить методы поиска с перемещением в начало и транспозицией;
- овладеть умениями и навыками написания на C++ программ по исследованию методов поиска с перемещением в начало и транспозицией.

### **Порядок выполнения работы**

- ознакомиться с краткой теорией и примерами решения задач, относящихся к методам оптимизации поиска;
- ответить на контрольные вопросы и получить оценку по знанию теории;
- получить задание на выполнение конкретного варианта лабораторной работы и выполнить его;
- написать и отладить программу решения задачи на языке C++;
- составить отчет по лабораторной работе и защитить его у преподавателя.

### **Содержание отчета по ЛР**

- наименование ЛР и ее цель;
- задание на ЛР согласно варианту;
- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;
- результаты работы программы.

### **Краткая теория**

Поиск (*search*) является одной из основ обработки данных в ЭВМ. Его назначение состоит в том, чтобы по заданному аргументу найти среди массива данных те данные, которые соответствуют этому аргументу или определить, что этих

данных нет. Если этих данных нет, добавить их в массив данных.

Набор любых данных будем называть таблицей или файлом. Каждое данное или элемент структуры отличается каким-то признаком от других данных. Этот признак называется ключом. Ключ может быть уникальным, т.е. в таблице только одно данное с таким ключом, иначе уникальные ключи называются первичным ключом.

Вторичный ключ в одной таблице может повторяться, но по нему тоже можно организовать поиск. Ключи данных собраны в одном месте (таблице).

Ключи, которые выделены из таблицы данных и организованы в свой файл, называются внешним ключами. Если ключ в записи, то он называется внутренним ключом.

Поиск по заданному аргументу называется **алгоритм**, определяющий соответствие ключа данного с заданным аргументом. Результатом работы алгоритма поиска может быть нахождение этого данного или отсутствие данного в таблице. В случае отсутствия данного возможны две операции:

1. Индикация того, что данного нет.
2. Вставка данного в таблицу.

Пусть  $K$  - массив ключей, тогда для всех  $K(i)$  существует  $R(i)$  - данное.  $KEY$  - аргумент поиска. Ему соответствует информационная запись  $REC$ . В зависимости от того, какова структура данных в таблице, различают несколько видов поиска.

Переупорядочение таблицы поиска.

Всегда можно говорить о некотором значении вероятности нахождения того или иного элемента.

$P(i)$  - вероятность нахождения элемента.

В этом случае таблица поиска представляется как система с дискретными состояниями, а искомый элемент характеризуется  $i$ -ым состоянием системы, вероятность которого  $P(i)$ .

$$P(1) + P(2) + \dots + P(n) = 1$$



Количество сравнений  $Z$  при поиске в таблице, т.е. количество сравнений, необходимых для поиска по заданному аргументу, представляет собой значение дискретной случайной величины, определяемой номерами состояний и вероятностями состояний системы.

$$Z = 1 * P(1) + 2 * P(2) + 3 * P(3) + \dots + n * P(n)$$

Таблица данных должна быть упорядочена таким образом, чтобы в начале таблицы были элементы с большими вероятностями поиска или элементы, к которым чаще всего обращаются в таблице.

$$P(1) \geq P(2) \geq P(3) \geq \dots \geq P(n)$$

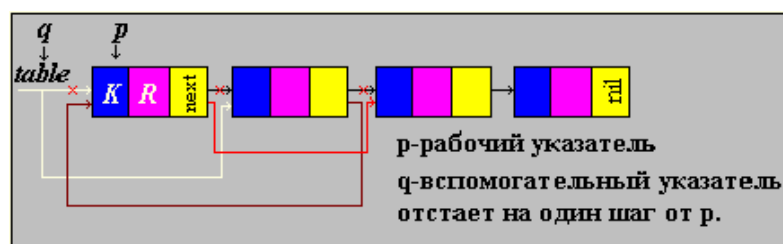
Имеется два основных метода переупорядочивания таблиц поиска:

## Алгоритмы

1) Переупорядочивание путем перестановки найденного элемента в начало списка.

2) Метод транспозиции.

Алгоритм переупорядочивания путем перестановки найденного элемента в начало списка.



Найденный элемент сразу оказывается в голове списка.

На рисунке проиллюстрирован случай, когда найденный элемент второй в списке. Первоначально указатель  $q$  нулевой, указатель  $p$  указывает на начало списка;  $p$  перемещается на второй элемент, а  $q$  на первый. Указатель начала списка ( $table$ ) перемещается на второй элемент, а указатель второго элемента на третий. Таким образом, второй элемент перемещается на первое место.

Нижеприведенный алгоритм предусматривает возможность перестановки в начало списка любого по счету найденного элемента списка, а также случай, когда найденный элемент в списке первый и перестановка не нужна.

Пример программы переупорядочивания (на псевдокоде)

```
q = nil
p = table
while (p <> nil) do
  if key = k(p) then
    search = p
    if q = nil
      then 'перестановка не нужна'
      return
    endif
    nxt(q) = nxt(p)
    nxt(p) = table
    table = p
    return
  endif
  q = p
  p = nxt(p)
endwhile
search = 0
return
```

### Метод транспозиции

В данном методе найденный элемент переставляется на один элемент к голове списка. Если к этому элементу обращаются часто, то он, постепенно перемещаясь к началу списка, скоро окажется в его голове.

Этот метод удобен тем, что он эффективен не только в списковых структурах, но и в неупорядоченных массивах, т.к. меняются местами только два рядом стоящих элемента.

Будем использовать три указателя:

$p$  - рабочий указатель, указывает на элемент.

$q$  - вспомогательный указатель, отстает на один шаг от  $p$ .

$s$  - вспомогательный указатель, отстает на два шага от  $p$ .

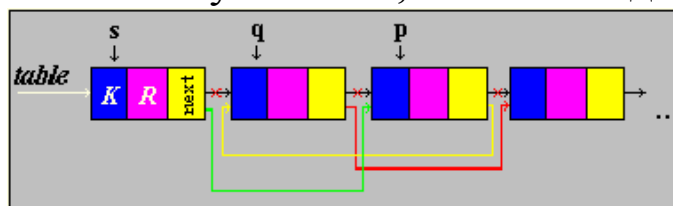


Рисунок иллюстрирует случай, когда найденный элемент третий в списке. Найденный нами третий элемент передвигается на один шаг к голове списка (т.е. становится вторым). Указатель первого элемента перемещается на третий элемент, указатель второго элемента на четвертый, таким образом, третий элемент перемещается на второе место. Если к данному элементу обратиться еще раз, то он окажется в голове списка.

Нижеприведенный алгоритм описывает метод транспозиции для любого по счету найденного элемента списка, а также случай, когда найденный элемент в списке первый и перестановка не нужна.

Пример программной реализации метода транспозиции (на псевдокоде)

$s = nil$

$q = nil$

$p = table$

*while* ( $p \neq nil$ ) *do*

*if*  $key = k(p)$  *then*

    ‘ нашли, транспонируем

*if*  $q = nil$  *then*

```

        ' переставлять не надо
        return
    endif
     $nxt(q) = nxt(p)$ 
     $nxt(p) = q$ 
if  $s = nil$  then
     $table = p$ 
else
     $nxt(s) = p$ 
     $search = p$ 
endif
return

endif
 $s = q$ 
 $q = p$ 
 $p = nxt(p)$ 
endwhile
 $search = nil$ 
return

```

**Контрольные вопросы по теории** (поиск с перемещением в начало и транспозицией)

1. Где наиболее эффективен метод транспозиций?
  - в массивах и в списках;
  - только в массивах;
  - только в списках.
2. В чём суть метода перестановки ?
  - найденный элемент помещается в голову списка;
  - найденный элемент помещается в конец списка;

- найденный элемент меняется местами с последующим.

### 3. В чём суть метода транспозиции ?

- перестановка местами соседних элементов;
- нахождение одинаковых элементов;
- перестановка найденного элемента на одну позицию в сторону начала списка.

### 4. Что такое уникальный ключ ?

- если разность значений двух данных равна ключу;
- если сумма значений двух данных равна ключу;
- если в таблице есть только одно данное с таким ключом.

### 5. В чём состоит назначение поиска ?

среди массива данных найти те данные, которые соответствуют заданному аргументу;  
определить, что данных в массиве нет;  
с помощью данных найти аргумент.

## Алгоритмы на языке C++

(поиск с перемещением и транспозицией)

Описанные выше методы оптимизации поиска для односвязного списка, полем данных которого будут целые числа, на языке C++ можно реализовать следующими функциями:

*/\* Функция поиска элемента в списке с перестановкой найденного элемента в начало списка \*/*

*PNode Find1(PNode& First, int KEY)*

*{*

*PNode search=0;*

*PNode q=NULL;*

*PNode p=First;*

*while (p)*

```

{
    if (KEY==p->Data)
    {
        search=p;
        if (!q)
        {
            return 0;
        }
        q->Next=p->Next;
        p->Next=First;
        First=p;
        return 0;
    }
    q=p;
    p=p->Next;
}
search=0;
return search;
}
/-----
-----/

```

*/\* Функция поиска элемента в списке с транспозицией \*/*

*PNode Find2(PNode& First, int KEY)*

```

{
    PNode search=0;
    PNode S=0;

```

```

PNode q=0;
PNode p=First;
while (p)
{
    if (KEY==p->Data)
    {
        if (!q)
        {
            return 0;
        }
        q->Next=p->Next;
        p->Next=q;
        if (!S) First=p;
        else
        {
            S->Next=p;
            search=p;
        }
        return search;
    }
    S=q;
    q=p;
    p=p->Next;
}
search=0;
return search;
}

```

/-----  
-----/

## Задания

Дан массив целых чисел. Решить заданную согласно варианту задачу и переставить найденный элемент обоими методами оптимизации поиска в начало списка.

1. Найти максимальный элемент массива.
2. Найти минимальный элемент массива.
3. Найти значение  $\lg(x)$  от каждого элемента и переставить на 1 место элемент, значение функции от которого максимально.
4. Найти число, нацело делящееся на 11 (если таких чисел несколько, то найти максимальное; если таких чисел нет - выдать сообщение).
5. Найти элемент, разность соседних элементов которого не меньше 72. Если таких элементов несколько, выбрать максимальный. Если таких элементов нет, выдать сообщение.
6. Найти элемент, частное соседних элементов которого четное число. Если таких элементов несколько, выбрать максимальный или минимальный элемент. Если таких элементов нет, выдать сообщение.
7. Найти элемент, разность соседних элементов которого четное число. Если таких элементов несколько, выбрать максимальный или минимальный элемент. Если такого элемента нет, выдать сообщение.
8. Найти элемент, среднее арифметическое элементов, находящихся до этого элемента равно 12. Если таких элементов нет, выдать сообщение.
9. Найти максимальный элемент, делящийся на 10. Если такого элемента нет, выдать сообщение.



10. Найти элемент, разность соседних элементов которого четное число и делится на 3. Если такого элемента нет, выдать сообщение.

11. Найти элемент, для которого квадратичное элементов, находящихся после этого элемента меньше 10. Если таких элементов несколько, выбрать максимальный элемент. Если таких элементов нет, выдать сообщение.

12. Найти значение  $\lg(x)$  от каждого элемента и переставить на 1 место элемент, значение функции от которого максимально.

**5.8. Составить отчет по лабораторной работе и защитить его у преподавателя**

## **Лабораторная работа №6 (4 часа). Поиск по дереву с включением и исключением**

### **Цель работы:**

- исследовать и изучить методы поиска элемента в дереве по заданному ключу со вставкой (включением) и удалением (исключением);
- овладеть умениями и навыками написания на C++ программ поиска по дереву с включением и исключением.

### **Порядок выполнения работы**

- ознакомиться с краткой теорией и примерами решения задач, относящихся к поиску по дереву с включением и исключением;
- ответить на контрольные вопросы и получить оценку по знанию теории;
- получить задание на выполнение конкретного варианта лабораторной работы и выполнить его;
- написать и отладить программу решения задачи на языке C++;
- составить отчет по лабораторной работе и защитить его у преподавателя.

### **Содержание отчета по ЛР**

- наименование ЛР и ее цель;
- задание на ЛР согласно варианту;
- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;
- результаты работы программы.

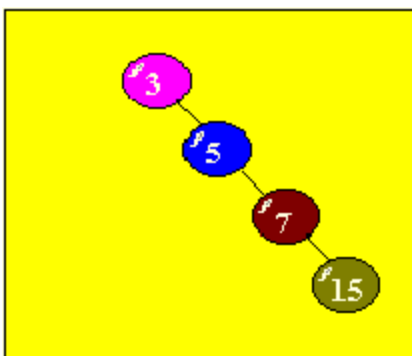
### **Краткая теория**

В лабораторной работе № 3 были рассмотрены основные понятия, касающиеся сбалансированных бинарных деревьев, и представлены алгоритмы создания и возможных обходов бинарных деревьев. В данной работе будут рассмотрены случаи, когда дерево уже создано и необходимо осуществить поиск

элемента по ключу в уже имеющемся бинарном дереве. На практике часто может возникать необходимость вначале осуществить поиск элемента в структуре, а затем, в случае, если элемент найден, либо информировать пользователя об этом, либо удалить его. В случае, если элемент не найден, пользователя необходимо либо об этом проинформировать, либо вставить в структуру элемент с ненайденным ключом.

#### Поиск по бинарному дереву.

Назначение его в том, чтобы по заданному ключу осуществить поиск узла дерева. Также для вставки элемента в дерево сначала нужно осуществить поиск в дереве по заданному ключу. Если такой ключ имеется, то программа завершается, если нет, то происходит вставка. Длительность операции поиска (число узлов, которые надо перебрать для этой цели) зависит от структуры дерева. Действительно, дерево может быть вырождено в однонаправленный список (иметь единственную ветвь) - такое дерево может возникнуть, если элементы поступали в дерево в порядке возрастания (убывания) их ключей, например:



В этом случае время поиска будет такое же, как и однонаправленном списке, т.е. в среднем придется перебрать  $N/2$  элементов.

Наибольший эффект использования дерева достигается в том случае, когда дерево "сбалансировано". В этом случае для поиска придется перебрать не более  $\log_2 N$ .

### **Алгоритмы**

#### Процедура поиска по бинарному дереву

Опишем процедуру поиска в псевдокоде. Переменной *search* будет присваиваться указатель на найденное звено:

```
p = tree
while p <> nil do
  if key = k(p) then
    search = p
    return
  endif
  if key < k(p) then
    p = left(p)
  else
    p = right(p)
  endif
endwhile
search = nil
return
```

### Включение элемента в дерево

Для включения новой записи в дерево, прежде всего, нужно найти тот узел, к которому можно "подвесить" (присоединить) новый элемент. Алгоритм поиска нужного узла тот же самый, что и при поиске узла с заданным ключом. Этот узел будет найден в тот момент, когда в качестве очередной ссылки, определяющий ветвь дерева, в которое надо продолжить поиск, окажется ссылка *NIL*.

Однако непосредственно использовать процедуру поиска нельзя, потому что по окончании вычисления ее значение не фиксирует тот узел, из которого была выбрана ссылка *NIL*. Модифицируем описание процедуры поиска так, чтобы в качестве ее побочного эффекта фиксировалась ссылка на узел, в котором был найден заданный ключ (в случае успешного поиска), или ссылка на узел, после обработки которого, поиск прекращен (в случае неуспешного поиска).

Алгоритм поиска по бинарному дереву с включением  
(вставкой) в псевдокоде.

```
p = tree
q = nil
while p <> nil do
    q = p
    if key = k(p) then
        search = p
        return
    endif
    if key < k(p) then
        p = left(p)
    else
        p = right(p)
    endif
endwhile
v = maketree(key, rec)
if q = nil then
    tree = v
else
    if key < k(q) then
        left(q) = v
    else
        right(q) = v
    endif
endif
search = v
return
```

Вспомогательный указатель *q* здесь отстает на один шаг от рабочего *p*.

### **Исключение элемента из дерева**

Удаление узла дерева не должно нарушать упорядоченность дерева. При удалении возможны 3 расположения узлов:

1) найденный узел является листом - он просто удаляется;

2) найденный узел имеет только сына - в этом случае сын перемещается на место удаленного отца;

3) у удаляемого отца два сына - в этом случае основная трудность состоит в удалении отца, поскольку в удаляемую вершину входит одна стрелка, а выходит две. В этом случае нужно найти подходящее звено поддеревы, которое должно занять место удаляемого без нарушения упорядоченности дерева. Данным звеном может быть либо предшественник удаляемого узла, либо его приемник.

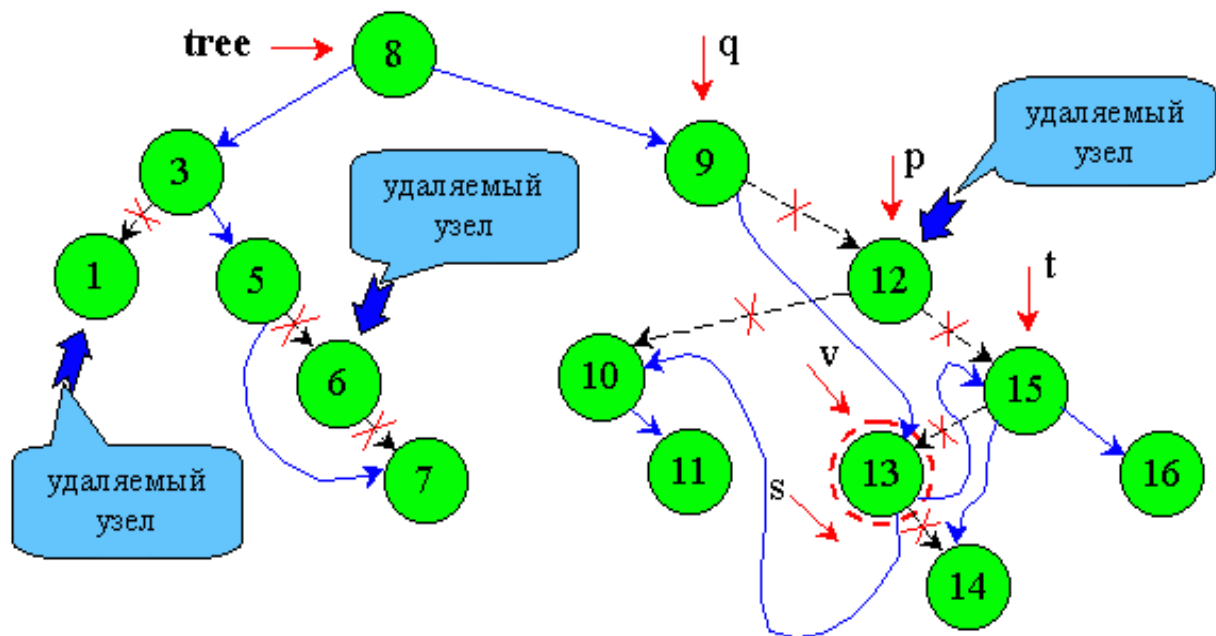
Предшественник - это самый правый элемент левого поддерева (для достижения этого элемента необходимо перейти в следующий узел по левой ветви, а затем двигаться только по правой ветви этого узла до тех пор, пока очередная ссылка не будет равна *nil*).

Приемник - это самый левый элемент правого поддерева (для достижения этого элемента необходимо перейти в следующий узел по правой ветви, а затем двигаться только по левой ветви этого узла до тех пор, пока очередная ссылка не будет равна *nil*).

Очевидно, что такие подходящие звенья могут иметь не более одной ветви.

На изображенном ниже рисунке проиллюстрированы возможные удаляемые из дерева узлы при трех возможных вариантах (лист (ключ 1), один сын (ключ 6), два сына (ключ 12)).

Предшественником удаляемого узла (12) является самый правый узел левого поддерева (11). Приемником узла (12) - самый левый узел правого поддерева (13).



Нижеприведенный в псевдокоде алгоритм разработан для поиска по дереву с удалением для всех трех возможных случаев нахождения узлов, причем в случае наличия у удаляемого узла двух сыновей его место занимает преемник (на рисунке узел 13 занимает место удаляемого узла 12).

Введем указатели:

$p$  - рабочий указатель;

$q$  - отстает от  $p$  на один шаг;

$v$  - указывает на преемника удаляемого узла;

$t$  - отстает от  $v$  на один шаг;

$s$  - на один шаг впереди  $v$  (указывает на левого сына или пустое место).

В результате поиска преемника на узел 13 должен указывать указатель  $v$ , а указатель  $s$  - на пустое место (как показано на рисунке).

$q = nil$

$p = tree$

*while* ( $p \neq nil$ ) and ( $k(p) \neq key$ ) *do*

```

     $q = p$ 
    if  $key < k(p)$  then
         $p = left(p)$ 
    else
         $p = right(p)$ 
    endif
endwhile
if  $p = nil$  then 'Ключ не найден'
    return
endif
if  $left(p) = nil$  then  $v = right(p)$ 
    else if  $right(p) = nil$ 
        then  $v = left(p)$ 
    else
        'У  $nod(p)$  - два сына'
        'Введем два указателя ( $t$  отстает от  $v$  на 1 шаг,  $s$  - опережа-
ет)',
         $t = p$ 
         $v = right(p)$ 
         $s = left(v)$ 
    while  $s \neq nil$  do
         $t = v$ 
         $v = s$ 
         $s = left(v)$ 
    endwhile
    if  $t \neq p$  then
        'v не является сыном p'

```



```

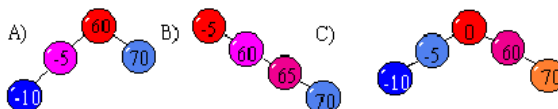
    left(t) = right(v)
    right(v) = right(p)
    endif
    left(v) = left(p)
  endif
endif
if q = nil then 'p - корень'
    tree = v
else if p = left(q)
    then left(q) = v
        else right(q) = v
    endif
endif
freenode(p)
return

```

## Контрольные вопросы по теории (поиск по дереву с включением)

### Включение элемента в дерево

1. В каком дереве при бинарном поиске нужно перебрать в среднем  $N/2$  элементов?



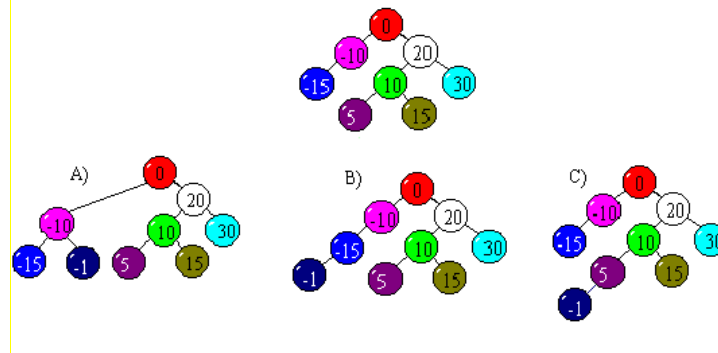
- A;
- B;
- C.

2. Сколько нужно перебрать элементов в сбалансированном дереве при бинарном поиске?

- A -  $N/2$ ;;

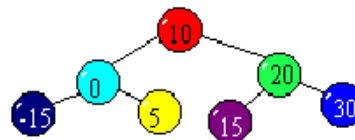
- $B - \ln(N)$ ;;
- $C - \log_2(N)$ ;;
- $D - e^N$ .

3. Выберите вариант дерева, полученного после вставки узла - 1.



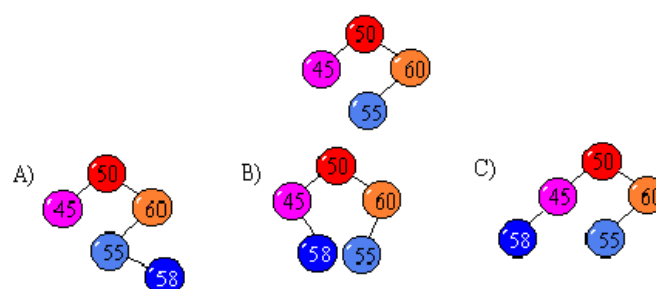
- A;
- B;
- C.

4. К какому элементу присоединить элемент 40 для вставки его в данное дерево?



- к 30-му;
- к 15-му;
- к -15-му;
- к 5-му.

5. Какой вид примет дерево после вставки элемента с ключом 58?

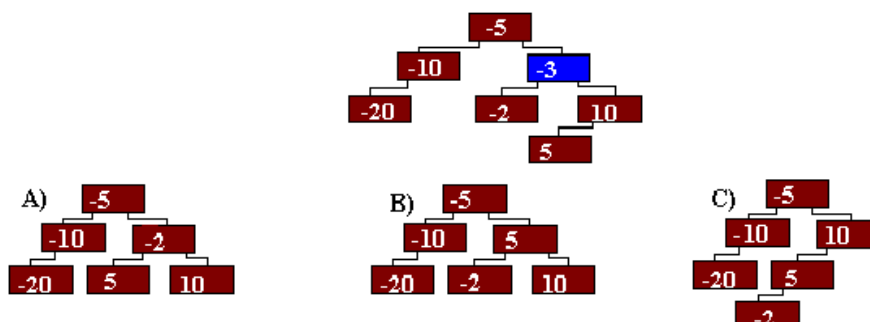


- A;
- B;

- C.

### *В. Исключение элемента из дерева*

1. Выберите вариант дерева, полученного после удаления узла – 3.

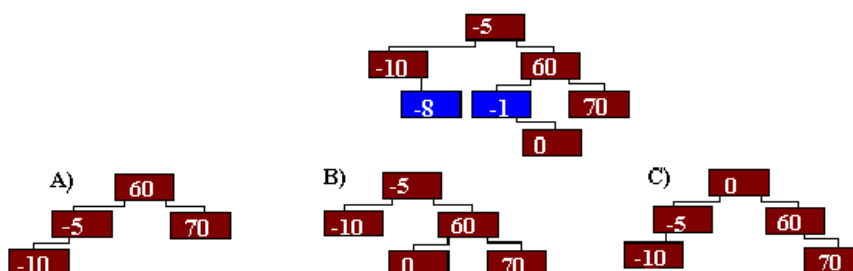


- A;

- B;

- C.

2. Какой вариант дерева получится после удаления элемента – 1, а затем – 8?

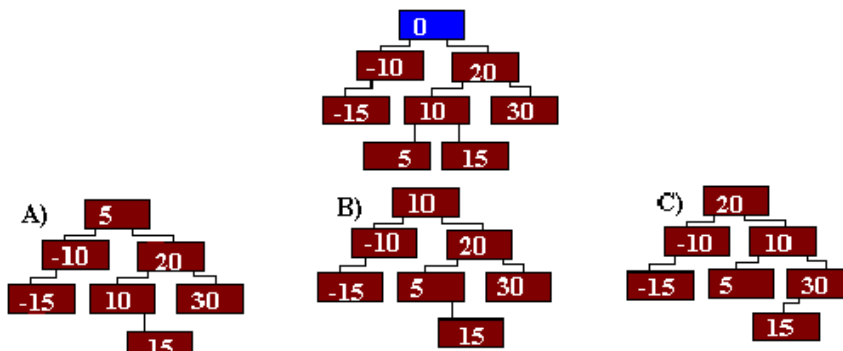


- A;

- B;

- C.

3. Выберите вариант дерева, полученного после удаления узла с индексом 0.

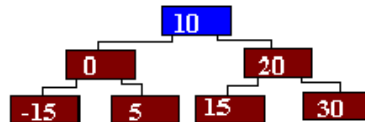


- A;

- B;

- С.

4. Какие из следующих пар чисел могут стать корнями дерева после удаления элемента 10 в соответствии с двумя способами удаления узла, имеющего двух сыновей?



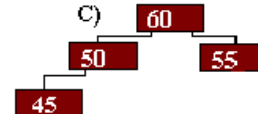
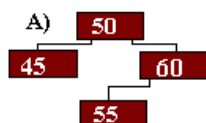
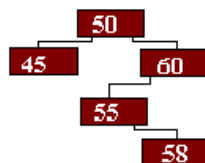
- 0 или 15;

- 0 или 20;

- 5 или 30;

- 5 или 15.

5. Какой вид примет дерево после удаления элемента с ключом 58?



- A;

- B;

- C.

### Примеры алгоритмов конкретных задач (поиск по дереву с включением, исключением)

Рассмотрим теперь, как реализуются алгоритмы поиска по дереву с включением и исключением на языке программирования C++.

Для простоты будем рассматривать бинарное упорядоченной дерево, в полях элементов которого ключи – вещественные числа, а поля записей отсутствуют. Описание самого элемента, функции создания и обхода подобного дерева

подробно описаны в лабораторной работе № 3, поэтому здесь мы на них останавливаться не будем. Функция поиска со вставкой при уже созданном дереве будет иметь следующий вид:

```
void Vstavka(double key) //функция, вставляющая элемент в дерево  
{Q=NULL;  
P=tree;  
while(P)  
{ Q=P;  
  if(key==P->k)  
    {cout<<"\n В дереве уже есть такой элемент, он найден, вставлять не надо\n";  
      return;  
    /*если такой элемент в дереве уже есть, то функция завершает свою работу*/  
    }  
    if(key<P->k) P=P->left;  
    else P=P->right;  
};  
V=new element;  
V->k=key;  
V->left=V->right=NULL;  
if(key<Q->k) Q->left=V;  
else Q->right=V;  
}
```

Функция поиска с удалением при уже созданном дереве будет иметь следующий вид:

*void Poisk\_Udalenie (double key) //функция поиска с удалением*

*{Q=NULL;*

*P=tree;*

*{ while(P&&P->k!=key)*

*{ Q=P;*

*if(key<P->k) P=P->left;*

*else P=P->right;*

*};*

*if(!P) { cout<<"Ключ не найден!"; return;};*

*if(P->left==NULL) V=P->right;*

*else*

*{if(P->right==NULL) V=P->left;*

*else*

*{ T=P;*

*V=P->right;*

*S=V->left;*

*while(S)*

*{ T=V;*

*V=S;*

*S=V->left;*

*};*

*if(T!=P) { T->left=V->right;*

*V->right=P->right;*

```

        }
        V->left=P->left;
    }
}

```

```

if(Q==NULL) tree=V;
    else
        if(P==Q->left) Q->left=V;
            else Q->right=V;

```

```

delete P;
}
}

```

Теперь пример программы, в которой создается выше-описанное дерево и осуществляется вставка элемента в него

```

/*Поиск по дереву со вставкой, обходы слева направо */
#include<iostream.h>
#include<conio.h>

```

```

struct element
{ double k;
  element* left;
  element* right;
};

```

```

element *tree=NULL,*P,*Q, *V;

```

```

void maketree(double a)
{ if(!tree)
    { tree=new element;
      tree->k=a;
      tree->right=tree->left=NULL;
      P=tree;
      Q=NULL;
      return;
    };
  P=Q=tree;
  while (P!=NULL)
  { Q=P;
    if(a<P->k) P=P->left; else P=P->right;
  };
  if(a<Q->k) { Q->left=new element;
              Q->left->k=a;
              Q->left->left=Q->left->right=NULL;
            }
  else      { Q->right=new element;
              Q->right->k=a;
              Q->right->left=Q->right->right=NULL;
            };
  }

void Vstavka(double key) //функция вставляющая эле-
мент в дерево
{
  P=tree;

```



```

Q=NULL;
while(P)
{ Q=P;
  if(key==P->k)
    {cout<<"\n В дереве уже есть такой элемент, он
найден, вставлять не надо\n";
    return;
    /*если такой элемент в дереве уже есть, то функция
завершает свою работу*/
  }
  if(key<P->k) P=P->left;
  else P=P->right;
};
V=new element;
V->k=key;
V->left=V->right=NULL;
if(key<Q->k) Q->left=V;
else Q->right=V;
}
void printtree(element* tree)
{ if(tree)
  { printtree(tree->left);
    cout<<tree->k<<" ";
    printtree(tree->right);
  };
}
void main()

```

```

    { clrscr();
    double a,key;
    cout<<"\n Введите элементы дерева, 0 - конец ввода:
\n";
    cin>>a;
    while(a)
    { maketree(a);
      cin>>a;
    };
    printtree(tree);
    cout<<endl<<"Введите ключ вставляемого элемента:
";
    cin>>key;
    Vstavka(key);
    printtree(tree);
    getch();
}

/*-----
----- */

```

Следующий пример – программа создания бинарного дерева с последующим поиском и удалением найденного элемента

```

/*Поиск по дереву со вставкой, обходы слева направо */
#include<iostream.h>
#include<conio.h>
struct element

```

```

{ double k;
  element* left;
  element* right;
};

element *tree=NULL, *P, *Q, *V, *T, *S;

void maketree(double a) //функция, создающая (добавля-
ющая) элемент в) дерево
{ if(!tree)
  { tree=new element;
    tree->k=a;
    tree->right=tree->left=NULL;
    P=tree;
    Q=NULL;
    return;
  };
  P=Q=tree;
  while (P!=NULL)
  { Q=P;
    if(a<P->k) P=P->left; else P=P->right;
  };
  if(a<Q->k) { Q->left=new element;
              Q->left->k=a;
              Q->left->left=Q->left->right=NULL;
            }
  else      { Q->right=new element;
              Q->right->k=a;
              Q->right->left=Q->right->right=NULL;
            }
  }

```

```

        };
    }
    void Poisk_Udalenie (double key) //функция поиска с уда-
    лением
    {
        Q=NULL;
        P=tree;
        { while(P&&P->k!=key)
            { Q=P;
              if(key<P->k) P=P->left;
              else P=P->right;
            };
        if(!P) { cout<<"Ключ не найден!"; return;};
        if(P->left==NULL) V=P->right;
        else
            {if(P->right==NULL) V=P->left;
              else
                { T=P;
                  V=P->right;
                  S=V->left;
                  while(S)
                  { T=V;
                     V=S;
                     S=V->left;
                  };
                  if(T!=P) { T->left=V->right;
                           V->right=P->right;

```

```

        }
        V->left=P->left;
    }
}

```

```

if(Q==NULL) tree=V;
    else
        if(P==Q->left) Q->left=V;
            else Q->right=V;

```

```

delete P;
}
}

```

*void printtree(element\* tree)//функция вывода дерева на экран*

```

{ if(tree)
{ printtree(tree->left);
    cout<<tree->k<<" ";
    printtree(tree->right);
};}

```

*void main()*

```

{ clrscr();

```

```

double a,key;

```

*cout<<"\n Введите элементы дерева, 0 - конец ввода:  
\n";*

```

cin>>a;

```

```

while(a)

```

```

    { maketree(a);
      cin>>a;
    };
    printtree(tree);//печать дерева до удаления из него эле-
мента
    Q=NULL;
    P=tree;
    cout<<endl<<"Введите ключ удаляемого элемента: ";
    cin>>key;
    Poisk_Udalenie(key);

    printtree(tree);
    getch();
}
/*-----
----- */

```

## Задания

Используя генератор случайных чисел сформировать бинарное дерево, состоящее из 15 элементов (предусмотреть ручной ввод элементов). Причем числа должны лежать в диапазоне от -99 до 99. Произвести поиск со вставкой и удалением элементов в соответствии со следующими вариантами заданий:

1. Числа кратные  $N$ .
2. Нечетные числа.
3. Числа  $> N$ .
4. Числа  $< N$ .
5. Числа по выбору.

6. Простые числа.
  7. Составные числа.
  8. Числа в интервале от  $X$  до  $Y$ .
  9. Числа, сумма цифр (по модулю) которого  $> N$ .
  10. Числа, сумма цифр (по модулю) которого  $< N$ .
  11. Числа, сумма цифр (по модулю) которого лежит в интервале от  $X$  до  $Y$ .
  12. Числа, взятые по модулю, квадратный корень которых целое число.
- где:  $N$ ,  $X$ ,  $Y$  - задается преподавателем.

**Составить отчет по лабораторной работе и защитить его у преподавателя**

## **СОРТИРОВКА**

### **Лабораторная работа №7 (4 часа). Сортировки методами прямого включения и выбора**

#### **Цель работы:**

- исследовать и изучить методы сортировки включением и выбором;
- овладеть умениями и навыками написания на языке программирования C++ программ с использованием сортировки включением и выбором.

#### **Порядок выполнения работы**

- ознакомиться с краткой теорией и примерами решения задач, относящихся к сортировке структур данных включением и выбором;
- ответить на контрольные вопросы и получить оценку по знанию теории;
- получить задание на выполнение конкретного варианта лабораторной работы и выполнить его;
- написать и отладить программу решения задачи на языке C++;
- составить отчет по лабораторной работе и защитить его у преподавателя.

#### **Содержание отчета по ЛР**

- наименование ЛР и ее цель;
- задание на ЛР согласно варианту;
- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;
- результаты работы программы.



## Краткая теория

При обработке данных в ЭВМ важно знать и информационное поле элемента, и его размещение в памяти машины. Для этих целей используется сортировка. Итак, сортировка - это расположение данных в памяти в регулярном виде по их ключам. Регулярность рассматривают как возрастание значения ключа от начала к концу в массиве.

Различают следующие типы сортировок:

- внутренняя сортировка - это сортировка, происходящая в оперативной памяти машины

- внешняя сортировка - сортировка во внешней памяти.

Если сортируемые записи занимают большой объем памяти, то их перемещение требует больших затрат. Для того, чтобы их уменьшить, сортировку производят в таблице адресов ключей, делают перестановку указателей, т.е. сам массив не перемещается. Это метод сортировки таблицы адресов.

При сортировке могут встретиться одинаковые ключи. В этом случае при сортировке желательно расположить после сортировки одинаковые ключи в том же расположении, что и в исходном файле. Это устойчивая сортировка.

Эффективность сортировки можно рассмотреть с нескольких критериев:

- время, затрачиваемое на сортировку
- объем оперативной памяти, требуемой для сортировки
- время, затраченное программистом на написание программы

Выделяем первый критерий, поскольку будем рассматривать только методы сортировки «на том же месте», то есть не резервируя для процесса сортировки дополнительную память. Эквивалентом затраченного на сортировку времени можно считать количество сравнений при выполнении сортировки и количество перемещений.

Различают следующие методы сортировки:

- строгие (прямые) методы
- улучшенные методы

Рассмотрим преимущества прямых методов:

1. Программы этих методов легко понимать, и они короткие. Напомним, что сами программы также занимают память.

2. Прямые методы особенно удобны для объяснения характерных черт основных принципов большинства сортировок.

3. Усложненные методы требуют небольшого числа операций, но эти операции обычно сами более сложны, и поэтому для достаточно малых количествах элементов прямые методы оказываются быстрее, хотя при больших количествах элементов их использовать, конечно, не следует.

Методы сортировки "на том же месте" можно разбить в соответствии с определяющими их принципами на 3 категории:

1. Сортировки с помощью включения (by insertion)
2. Сортировки с помощью выбора (by selection)
3. Сортировки с помощью обменов (by exchange)

### **Сортировка методом прямого включения**

Такой метод широко используется при игре в карты. Элементы (карты) мысленно делятся на уже "готовую" последовательность  $a(1), \dots, a(i-1)$  и исходную последовательность. При каждом шаге, начиная с  $i=2$  и увеличивая  $i$  каждый раз на единицу, из исходной последовательности извлекается  $i$ -й элемент и перекладывается в готовую последовательность, при этом он вставляется на нужное место.

Алгоритм сортировки прямым включением таков:

```
for x=2 to n do
    x=a[i]
```

включение  $x$  на соответствующее место среди  $a[1]...a[i]$   
*end*

В реальном процессе поиска подходящего места удобно, чередуя сравнения сравнивать текущий элемент с очередным элементом  $a(j)$ , а затем

- либо  $x$  вставляется на свободное место,
- либо  $a(j)$  сдвигается (передается) вправо и процесс "уходит" влево.

Обратите внимание, что процесс просеивания может закончиться при выполнении одного из двух следующих различных условий:

1. Найден элемент  $a(j)$  с ключом, меньшим, чем ключ  $x$ .
2. Достигнут левый конец готовой последовательности.

### **Алгоритм сортировки прямым включением**

В псевдокоде алгоритм сортировки методом прямого включения следующий:

```
for  $i = 2$  to  $n$   
   $x = a(i)$   
   $a(0) = x$  { $a(0)$  - барьер}  
   $j = i - 1$   
  while  $x < a(j)$  do  
     $a(j + 1) = a(j)$   
     $j = j - 1$   
  endwhile  
   $a(j + 1) = x$   
next i  
return
```

Для внутреннего цикла, организованного как цикл *while*, необходима постановка «барьера», без которого при отрица-

тельных значениях ключей происходит потеря значимости и «зависание» компьютера.

Эффективность алгоритма:

Наилучшие оценки числа сравнений  $C_{min}$  и перемещений  $M_{min}$  встречаются в случае уже упорядоченной исходной последовательности элементов, наихудшие же оценки  $C_{max}$  и  $M_{max}$  - когда они первоначально расположены в обратном порядке.

Количество сравнений в худшем случае, когда массив отсортирован противоположным образом,  $C_{max} = n(n-1)/2$ , то есть порядок  $O(n^2)$ . Количество перестановок  $M_{max} = C_{max} + 3(n-1)$ , то есть порядок  $O(n^2)$ .

## **Сортировка методом прямого выбора**

В общем сортировка - это процесс перегруппировки заданного множества объектов в некотором определенном порядке. Цель сортировки облегчить последующий поиск элементов в таком отсортированном множестве. Это почти универсальная, фундаментальная деятельность. Мы встречаемся с отсортированными объектами в телефонных книгах, в списках подходящих налогов, в оглавлениях книг, в библиотеках, в словарях, на складах почти везде, где нужно искать хранимые объекты. Даже малышей учат держать свои вещи "в порядке", и они уже сталкиваются с некоторыми видами сортировок задолго до того, как познакомятся с азами арифметики.

Таким образом, разговор о сортировке вполне уместен и важен, если речь идет об обработке данных. Что может легче сортироваться, чем данные? Тем не менее наш первоначальный интерес к сортировке основывается на том, что при построении алгоритмов мы сталкиваемся со многими весьма фундаментальными приемами. Почти не существует методов, с которыми не приходится встречаться при обсуждении этой

задачи. В частности, сортировка это идеальный объект для демонстрации огромного разнообразия алгоритмов, все они изобретены для одной и той же задачи, многие в некотором смысле оптимальны, большинство имеет свои достоинства. Поэтому это еще и идеальный объект, демонстрирующий необходимость анализа производительности алгоритмов. К тому же на примерах сортировок можно показать, как путем усложнения алгоритма, хотя под рукой и есть уже очевидные методы, можно добиться значительного выигрыша в эффективности.

Выбор алгоритма зависит от структуры обрабатываемых данных это почти закон, но в случае сортировки такая зависимость столь глубока, что соответствующие методы были даже разбиты на два класса сортировку массивов и сортировку файлов последовательностей. Иногда их называют внутренней и внешней сортировкой, поскольку массивы хранятся в быстрой оперативной, внутренней памяти машины со случайным доступом, а файлы обычно размещаются в более медленной, но и более емкой внешней памяти, на устройствах, основанных на механических перемещениях дисков или лент. На примере сортировки пронумерованных карточек становится очевидным существенное различие в этих подходах. Если карты "выстроены" в виде массива, то они как бы лежат перед сортирующим, он видит каждую из них и имеет к ней доступ. Если же карты образуют файл, то это предполагает, что видна только верхняя карта в каждой из стопок. Такое ограничение, конечно же, серьезно повлияет на метод сортировки, но ничего не поделаешь: ведь карточек может быть так много, что все они на столе не поместятся.

Прежде чем идти дальше, введем некоторые понятия и обозначения. Ими мы будем пользоваться далее. Если у нас есть элементы  $a_1, a_2, \dots, a_n$ , то сортировка есть перестановка этих элементов массив  $a_{k1}, a_{k2}, \dots, a_{kn}$ , где при некоторой упорядочивающей функции  $f$  выполняются отношения  $f a_{k1} \leq f a_{k2} \leq \dots \leq f a_{kn}$ .

Обычно упорядочивающая функция не вычисляется по какому-либо правилу, а хранится как явная компонента поля каждого элемента. Ее значение называется ключом *key* элемента. Поэтому для представления элементов хорошо подходят такие образования, как запись, а графически это представляется так - а:



Отсортированный массив - b:



Массив, отсортированный другим методом - c:



Говоря об алгоритмах сортировки, мы будем обращать внимание лишь на компоненту - ключ, другие же компоненты можно даже и не определять (b). Чтобы уменьшить эти затраты, сортировку производят в таблице адресов ключей. После сортировки переставляют указатели. Это метод сортировки таблицы адресов (c). Метод сортировки называется устойчивым, если в процессе сортировки относительное расположение элементов с равными ключами не изменяется. Устойчивость сортировки часто бывает желательной, если речь идет об элементах, уже упорядоченных (отсортированных) по некоторым вторичным ключам (т.е. свойствам), не влияющим на основной ключ.

Основное условие: выбранный метод сортировки массивов должен экономно использовать доступную память. Это предполагает, что перестановки, приводящие элементы в порядок, должны выполняться на том же месте, т.е. методы, в которых элементы из массива  $A$  передаются в результирующий массив  $H$ , представляют существенно меньший интерес. Ограничив критерием экономии памяти наш выбор нужного метода среди многих возможных, мы будем сначала классифицировать методы по их экономичности, т.е. по времени их работы. Хорошей мерой эффективности может быть  $C$  - число необходимых сравнений ключей и  $M$  - число пересылок (перестановок) элементов.

Эти числа - функции от  $n$  - числа сортируемых элементов. Сортировка методом прямого выбора требует порядка  $n^2$  сравнений ключей.

Рассматриваем весь ряд массива и выбираем элемент меньший или больший элемента  $a(i)$ , определяем его место в массиве -  $k$ , и затем меняем местами элемент  $a(i)$  и элемент  $a(k)$ .

### Алгоритм сортировки прямым выбором

```
for  $i = 1$  to  $n - 1$ 
   $x = a(i)$ 
   $k = i$ 
  for  $j = i + 1$  to  $n$ 
    if  $a(j) < x$  then
       $k = j$ 
       $x = a(k)$ 
    endif
  next  $j$ 
   $a(k) = a(i)$ 
   $a(i) = x$ 
next  $i$ 
return
```

Эффективность алгоритма:

- Количество сравнений

$$M = \frac{N}{2} \cdot (N-1) = \frac{N^2 - N}{2}$$

- Количество перемещений, когда массив упорядочен

$$C_{\min} = 3 \cdot (N-1)$$

- Количество перемещений когда массив обрат-но отсортирован

$$C_{\max} = C_{\min} \cdot \frac{N}{2} = 3 \cdot (N-1) \cdot \frac{N}{2}$$

В худшем случае сортировка прямым выбором дает порядок  $n^2$ , как и для числа сравнений, так и для числа перемещений.

## Контрольные вопросы по теории

### *Сортировка методом включения*

1. Даны три условия окончания просеивания при сортировке прямым включением. Найдите среди них лишнее.

- найден элемент  $a(i)$  с ключом, меньшим чем ключ  $u$ ;
- найден элемент  $a(i)$  с ключом, большим чем ключ  $u$ ;
- достигнут левый конец готовой последовательности.

2. Какой из критериев эффективности сортировки определяется формулой  $M = 0,01 * n * n + 10 * n$ ?

- число сравнений;
- время, затраченное на написание программы;
- количество перемещений;
- время, затраченное на сортировку.

3. Как называется сортировка, происходящая в оперативной памяти?

- сортировка таблицы адресов;
- полная сортировка;
- сортировка прямым включением;



- внутренняя сортировка;
- внешняя сортировка.

4. Как можно сократить затраты машинного времени при сортировке большого объема данных?

- производить сортировку в таблице адресов ключей;
- производить сортировку на более мощном компьютере;
- разбить данные на более мелкие порции и сортировать их.

5. Существуют следующие методы сортировки. Найдите ошибку.

- строгие;
- улучшенные;
- динамические.

### *Сортировка методом выбора.*

1. Метод сортировки называется устойчивым, если в процессе сортировки...

- относительное расположение элементов безразлично;
- относительное расположение элементов с равными ключами не меняется;
- относительное расположение элементов с равными ключами изменяется;
- относительное расположение элементов не определено.

2. Улучшенные методы имеют значительное преимущество:

- при большом количестве сортируемых элементов;
- когда массив обратно упорядочен;
- при малых количествах сортируемых элементов;
- во всех случаях.

3. Что из перечисленных ниже понятий является одним из типов сортировки?

- внутренняя сортировка;
- сортировка по убыванию;
- сортировка данных;

- сортировка по возрастанию.

4. Сколько сравнений требует улучшенный алгоритм сортировки?

-  $n \cdot \log(n)$ ;

-  $e^n$ ;

-  $n \cdot n/4$ .

5. К какому методу относится сортировка, требующая  $n \cdot n$  сравнений ключей?

- прямому;

- бинарному;

- простейшему;

- обратному.

### Примеры алгоритмов и приложений

Рассмотрим теперь реализацию функций вышеописанных сортировок на C++ и примеры сортировки конкретных числовых массивов.

```
/* ****  
*/  
/* СОРТИРОВКА МЕТОДОМ ПРЯМОГО ВКЛЮЧЕНИЯ  
*/  
#include <stdio.h>  
#include <conio.h>  
#define n 8  
int A[n]={27,412,71,81,59,14,273,87};  
/*  
=====  
= */  
main()  
{ void Sis(int A[],int nn);  
  int j;  
  printf("\n Сортировка методом прямого включения");  
  printf("\n Исходный массив: \n\t");  
  for (j=0; j<n; j++)
```

```

        printf("%d\t",A[j]);
        printf("\n");
    Sis(A,n);
    printf("\n Отсортированный массив :\n\t");
    for (j=0; j<n; j++)
        printf("%d\t",A[j]);
    printf("\n");
    getch(); return 0;
}
/*
=====
*/
/* ФУНКЦИЯ СОРТИРОВКИ ПРЯМЫМ ВКЛЮЧЕНИЕМ
*/
void Sis(int A[],int nn)
{ int i,j,k;
  printf("\n Отладочная печать по шагам сортировки");
  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;
    /* Отладочная печать */
    printf("\n j = %d",j);
    for (i=0; i<nn; i++)
        printf("\t%d",A[i]);

  }
}
/* *****
*/
*/

```

Алгоритм с прямыми включениями можно легко улучшить с учетом того, что готовая последовательность, куда надо вставить новый элемент, уже упорядочена. Тогда ме-

сто включения ищется методом двоичного (бинарного) поиска. Такой улучшенный алгоритм сортировки называется методом с двоичным включением (методом прямого включения с делением пополам). Однако применение этого алгоритма оправдано только тогда, когда число сортируемых элементов достаточно велико.

```

/*
***** */
/* СОРТИРОВКА МЕТОДОМ ПРЯМОГО ВКЛЮЧЕНИЯ
С ДЕЛЕНИЕМ ПОПОЛАМ */
#include <stdio.h>
#include <conio.h>
#define n 8
int A[n] = {27,412,71,81,59,14,273,87};
/*
=====
== */
main()
{ void BinIns(int A[],int nn);
  int j;
    printf("\n Сортировка методом прямого включения с
деления пополам");
    printf("\n Исходный массив \n");
    for (j=0; j<n; j++)
        printf("\t%d",A[j]);
    printf("\n");
    BinIns(A,n);
    printf("\n Отсортированный массив \n");
    for (j=0; j<n; j++)
        printf("\t%d",A[j]);
    printf("\n");
    getch(); return 0;
}

```

```

/*
=====
*/
/* ФУНКЦИЯ СОРТИРОВКИ ПРЯМЫМ ВЫБОРОМ С
ДЕЛЕНИЕМ ПОПОЛАМ */
void BinIns(int A[],int nn)
{ int i,j,x,m,L,R;
  printf("\n Отладочная печать по шагам сортировки ");
  for ( i=1; i<nn; i++ )
    { x = A[i]; L = 0; R = i;
      while (L < R)
        { m = (L+R)/2;
          if (A[m] <= x)
            L = m+1;
          else
            R = m;
        }
      for (j=i; j>=R; j--)
        A[j] = A[j-1];
      A[R] = x;
      /* отладочная печать */
      printf("\ni=%d",i);
      for (j=0; j<nn;j++)
        printf("\t %d",A[j]);
    }
}
/* ***** */

```

Последний рассматриваемый в данной работе алгоритм – сортировка методом прямого выбора.

```

/*
***** */
/* СОРТИРОВКА ПОСРЕДСТВОМ ПРЯМОГО ВЫБОРА
*/
#include <stdio.h>

```

```

#include <conio.h>
#define n 8
int A[n] = {27,412,71,81,59,14,273,87};
/*
=====
*/

int main()
{ void StrSel(int A[],int nn);
  int j;
    printf("\n Сортировка методом прямого выбора");
    printf("\n Исходный массив \n");
    for (j=0; j<n; j++)
        printf("\t%d",A[j]);
        printf("\n");
    StrSel(A,n);
    printf("\n Отсортированный массив \n");
    for (j=0; j<n; j++)
        printf("\t%d",A[j]);
        printf("\n"); getch();
  }
  /*
=====
===== */

/* ФУНКЦИЯ СОРТИРОВКИ ПРЯМЫМ ВЫБОРОМ */
void StrSel(int A[],int nn)
{ int i,j,x,k;
  printf("\n Отладочная печать по шагам сортировки");
  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;

      printf("\n i=%d",i);
      for (j=0; j<nn;j++)
        printf("\t %d",A[j]);

    }
  }
  /* *****
*/
*/

```

## Задания

### *Сортировка методом включения*

В ремонтной мастерской находятся несколько ( $N$ ) машин. О них имеются следующие сведения:

- номер,
- марка,
- имя владельца,
- дата последнего ремонта (число, месяц, год),
- день, к которому машина должна быть отремонтирована (число, месяц, год).

Требуется (согласно варианту):

1. Расположить по алфавиту имена владельцев и, соответственно, вывести информацию об их машинах.
2. Исходя из того, что машина, дата окончания ремонта которой раньше, должна ремонтироваться в первую очередь, вывести порядок ремонта автомобилей.
3. Вывести по убыванию количество всех предыдущих ремонтов машин марки "Жигули".

4. Вывести по убыванию номера тех машин, количество предыдущих ремонтов которых равно 2.

5. Вывести по возрастанию даты конца ремонта всех машин, которые ранее не ремонтировались.

6. Вывести по алфавиту в обратном порядке владельцев автомобилей марки "Мерседес"

7. Вывести по алфавиту марки машин, которые должны быть отремонтированы раньше всех (дата конца ремонта меньше 01.08.96).

8. Вывести по возрастанию номера машин марки "Жигули".

9. Вывести по алфавиту имена владельцев, чьи машины не ремонтировались с прошлого года.

10. Вывести машины, которые надо отремонтировать к следующему месяцу по возрастанию даты их последнего ремонта.

11. Вывести по алфавиту в обратном порядке имена владельцев, количество предыдущих ремонтов машин которых больше трех.

12. Вывести по убыванию номера машин марки "Мерседес".

### *Сортировка методом выбора*

Создать группу из  $N$  студентов.

Ввести их:

- фамилии, имена, годы рождения,
- оценки по предметам:

структуры и алгоритмы данных, высшая математика, физика, программирование,

- общий балл сдачи сессии.

Разработать программу, которая осуществляет сортировку (согласно варианту) :

1. Фамилий студентов по алфавиту.
2. Фамилий студентов по алфавиту в обратном порядке.
3. Студентов по старшинству (начиная со старшего).



4. Студентов по старшинству (начиная с младшего).
5. Студентов по общему баллу (по возрастанию).
6. Студентов по общему баллу (по убыванию).
7. Студентов по результатам 1-го экзамена (по возрастанию).
8. Студентов по результатам 2-го экзамена (по убыванию).
9. Студентов по результатам 3-го экзамена (по возрастанию).
10. Студентов по результатам 4-го экзамена (по убыванию).
11. Имен в алфавитном порядке.
12. Имен в обратном алфавитном порядке.

**7.8. Составить отчет по лабораторной работе и защитить его у преподавателя**

## **Лабораторная работа №8 (4 часа). Сортировки методом прямого обмена и с помощью дерева.**

### **Цель работы:**

- исследовать и изучить методы сортировки с помощью прямого обмена и с помощью дерева;
- овладеть умениями и навыками написания на языке программирования C++ программ сортировки с использованием прямого обмена.

### **Порядок выполнения работы**

- ознакомиться с краткой теорией и примерами решения задач, относящихся к сортировкам методом прямого обмена и с помощью дерева;
- ответить на контрольные вопросы и получить оценку по знанию теории;
- получить задание на выполнение конкретного варианта лабораторной работы и выполнить его;
- написать и отладить программу решения задачи на языке C++;
- составить отчет по лабораторной работе и защитить его у преподавателя.

### **Содержание отчета по ЛР**

- наименование ЛР и ее цель;
- задание на ЛР согласно варианту;
- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;
- результаты работы программы.

### **Краткая теория**

В предыдущей лабораторной работе оба разбиравшихся метода сортировки рассматривать как "обменные" сортировки. В данном разделе описан метод, где обмен местами двух элементов представляет собой характернейшую особенность процесса. Изложенный ниже алгоритм прямого обмена осно-

ываается на сравнении и смене мест для пары соседних элементов и продолжении этого процесса до тех пор, пока не будут упорядочены все элементы.

### Сортировка методом прямого обмена (Пузырьковая сортировка).

Как в методе прямого выбора, мы повторяем проходы по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к левому концу массива. Будем рассматривать массив из  $n$  элементов, причем не горизонтальный, а вертикальный, тогда элементы можно интерпритировать как пузырьки в чане с водой, причем вес каждого соответствует его ключу. Массив проходится  $(n-1)$  раз сверху вниз, при этом элементы попарно сравниваются, если нижний элемент меньше верхнего, то элементы переставляются.

## Алгоритмы

Алгоритм метода прямого обмена (пузырьковой сортировки) в псевдокоде следующий:

```
for  $i = 2$  to  $n$ 
  for  $j = n$  to  $i$  step  $-1$ 
    if  $a(j) < a(j - 1)$  then
       $x = a(j - 1)$ 
       $a(j - 1) = a(j)$ 
       $a(j) = x$ 
    endif
  next  $j$ 
next  $i$ 
return
```

Рассмотрим пример.

Пусть дан массив из следующих элементов: 4, 3, 7, 2, 1, 6.

На нижеприведенном рисунке изображена иллюстрация алгоритма пузырьковой сортировки для рассматриваемого массива.

Номер прохода	1	2	3	4	5
4	1	1	1	1	1
3	4	2	2	2	2
7	3	4	3	3	3
2	7	3	4	4	4
1	2	7	5	5	5
6	5	5	6	6	6
5	6	6	7	7	7

В данном случае получился один проход “вхолостую”. Чтобы лишний раз не просматривать элементы, а, значит, не проводить сравнения, затрачивая на это время, можно ввести флажок *fl*, который остается в значении *false*, если при очередном проходе не будет произведено ни одного обмена. Ниже приведен усовершенствованный алгоритм.

```

fl = true
for i = 2 to n
  if fl = false then return
  endif
  fl = false
  for j = n to i step -1
    if  $a(j) < a(j - 1)$  then
      fl = true
       $x = a(j - 1)$ 
       $a(j - 1) = a(j)$ 
       $a(j) = x$ 
    endif
  next j
next i
return
  
```

Еще одним усовершенствованием алгоритма является так называемая шейкерная сортировка, где после каждого прохода меняется направление во внутреннем цикле.

В случае, если реализовывать алгоритм пузырьковой сортировки без усовершенствований, то

Количество сравнений:  $M_{\max} = \frac{n}{2} \cdot \frac{(n-1)}{2}$

Количество перемещений:  $C_{\max} = 3n \cdot \frac{n-1}{2}$

Из приведенных расчетов следует, что эффективность сортировки данным методом также имеет порядок  $n^2$ .

В классическом виде сортировка методом прямого обмена (пузырьковая) представляет собой нечто среднее между сортировками с помощью включений и с помощью выбора. Если же в нее внесены приведенные выше усовершенствования, то для достаточно упорядоченных массивов пузырьковая сортировка даже имеет преимущество. К преимуществам пузырьковой сортировки можно также отнести простоту алгоритма ее реализации.

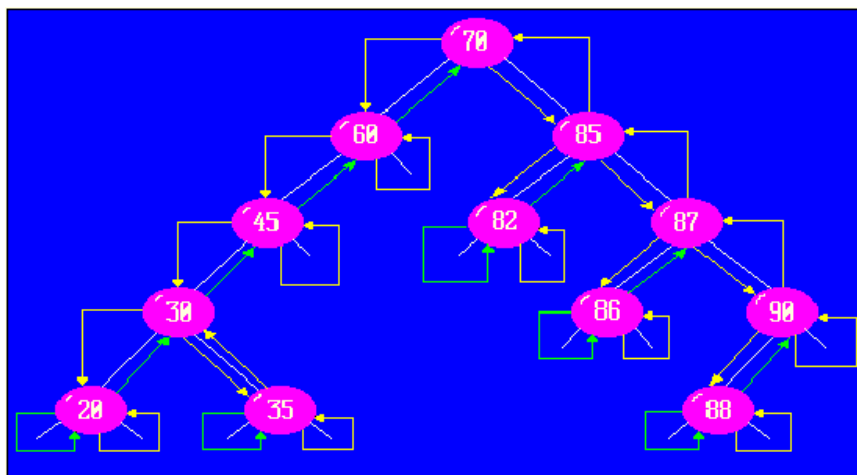
Прежде чем переходить к методам реализации алгоритмов пузырьковой сортировки на языке C++, рассмотрим еще один метод сортировки. Все вышерассмотренные сортировки производились на статистических структурах без динамического выделения памяти. Однако в некоторых случаях сортировка может быть более эффективной, если для ее реализации использовать динамические структуры данных. Естественно, в этом случае усложняется сам алгоритм сортировки. Одним из таких методов является сортировка с помощью бинарного дерева.

В лабораторной работе № 3 были подробно рассмотрены алгоритмы создания и обхода сбалансированного бинарного дерева. Если сортируемые элементы являются це-

лыми числами, и при занесении их в память машины они будут являться ключами создаваемого сбалансированного бинарного дерева, то при обходе полученного дерева слева направо получим отсортированный по возрастанию массив.

Пусть в первоначально пустой массив заносятся последовательно поступающие элементы: 70, 60, 85, 87, 90, 45, 30, 88, 35, 20, 86.

При обходе дерева слева - направо получаем отсортированный массив 20, 30, 35, 45, 60, 70, 82, 85, 86, 87, 88, 90. Элемент дерева заносится в массив при втором заходе в него (на рисунке вторые заходы показаны зелеными стрелками).



Алгоритмы создания и обхода слева направо в псевдокоде и на языке C++ упорядоченного бинарного дерева здесь не представлены, поскольку они подробно описаны в лабораторной работе № 3.

Теперь вернемся к сортировке методом прямого обмена (пузырьковой) и рассмотрим листинги реализации программ

**Контрольные вопросы по теории** (сортировка с помощью обмена).

1. Сколько сравнений и перестановок элементов требуется в пузырьковой сортировке?

- $n \cdot \log(n)$ ;
- $(n \cdot n)/4$ ;
- $(n \cdot n - n)/2$ .

2. Сколько дополнительных переменных нужно в пузырьковой сортировке помимо массива, содержащего элементы?

- 0 (не нужно);
- всего 1 элемент;
- $n$  переменных (ровно столько, сколько элементов в массиве).

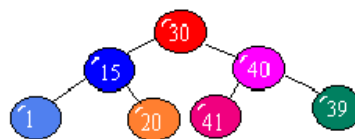
3. Как рассортировать массив быстрее, пользуясь пузырьковым методом?

- одинаково;
- по возрастанию элементов;
- по убыванию элементов.

4. Массив сортируется “пузырьковым” методом. За сколько проходов по массиву самый “лёгкий” элемент в массиве окажется вверху?

- за 1 проход;
- за  $n-1$  проходов;
- за  $n$  проходов, где  $n$  – число элементов массива.

5. При обходе дерева

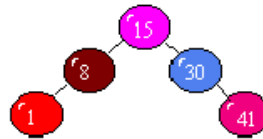


- слева направо получаем последовательность...
- отсортированную по убыванию;
- не отсортированную;
- отсортированную по возрастанию.

6. При обходе дерева слева направо его элемент заносится в массив...

- при втором заходе в элемент;
- при первом заходе в элемент;
- при третьем заходе в элемент.

7. Элемент массива с ключом  $k=20$  необходимо вставить в изображённое дерево так, чтобы дерево осталось отсортированным. Куда его нужно вставить?



- левым сыном элемента 30;
- левым сыном элемента 41;
- левым сыном элемента 8.

8. При обходе какого дерева слева направо получается отсортированный по возрастанию массив?



- A;
- B;
- C.

### Примеры алгоритмов конкретных задач

Рассмотрим теперь реализацию функций сортировки методом пузырька и его усовершенствований на C++ и примеры сортировки конкретных числовых массивов.

```
/*
*****
*/

/* СОРТИРОВКА МЕТОДОМ ПУЗЫРЬКА */
#include <stdio.h>
#include <conio.h>
#define n 8
int A[n] = {27,412,71,81,59,14,273,87};
```



```

/*
=====
*/

main()
{ void BblSort(int A[],int nn);
  int j;
    printf("\n Сортировка методом пузырька");
    printf("\n Исходный массив \n");
    for (j=0; j<n; j++)
        printf("\t%d",A[j]);
    printf("\n");
    BblSort(A,n);
    printf("\n Отсортированный массив \n");
    for (j=0; j<n; j++)
        printf("\t%d",A[j]);
    printf("\n");
    getch(); return 0;
}

/*
=====
===== */

/* ФУНКЦИЯ СОРТИРОВКИ МЕТОДОМ ПУЗЫРЬКА
*/

void BblSort(int A[],int nn)
{ int i,j,k,p;
    printf("\n Отладочная печать по шагам сортиров-
ки");
    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 вы-
полнена */

        if ( p == 0)
            break;

        printf(" \ni = %d",i);
        for (j=0; j<nn;j++)
            printf("\t %d",A[j]);

        }

    }

/*
*****
* */

```

В данном примере функция сортировки методом пузырька BblSort описана таким образом, чтобы “холостых” проходов не было.

В заключительном листинге представлена программе сортировки того же массива с помощью функции Шейкерной сортировки, которая является усовершенствованием пузырьковой.

```

/*
***** */

/*ШЕЙКЕРНАЯ СОРТИРОВКА. */

#include <stdio.h>

```

```

#include <conio.h>
#define n 8
int A[n] = {27,412,71,81,59,14,273,87};
/*
===== */
main()
{ void ShkrSort(int A[],int nn);
  int j;
  printf("\n Шейкерная сортировка");
  printf("\n Исходный массив: \n");
  for (j=0; j<n; j++)
    printf("%d\t",A[j]);
  printf("\n");
  ShkrSort(A,n);
  printf("\n Отсортированный массив :\n");
  for (j=0; j<n; j++)
    printf("%d\t",A[j]);
  printf("\n");
  getch(); return 0;
}
/*
===== */

/* ФУНКЦИЯ ШЕЙКЕРНОЙ СОРТИРОВКИ */
void ShkrSort(int A[],int nn)
{ int i,j,k,x,L,R;
  L = 1; R = nn-1; k = nn-1;
  printf("\n Отладочная печать");

```

```

do
{
for ( j=R; j>=L; j-- )
if ( A[j-1] > A[j] )
{ x = A[j-1]; A[j-1] = A[j]; A[j]=x; k=j; }
L = k + 1;

/* Отладочная печать */
printf(" \nL = %d",L);
for (i=0; i<nn; i++)
printf("\t%d",A[i]);
for (j=L; j<=R; j++)
if ( A[j-1] > A[j] )
{ x = A[j-1]; A[j-1] = A[j]; A[j] = x; k = j; }
R = k -1;

/* Отладочная печать */
printf(" \nR = %d",R);
for (i=0; i<nn; i++)
printf("\t%d",A[i]);
}
while ( L < R );
}
/* **** */

```

## Задания

1. На заводе выпустили детали со следующими серийными номерами: 45, 56, 13, 75, 14, 18, 43, 11, 52, 12, 10, 36, 47, 9. Детали с четными номерами поступают на склад №1, а

с нечетными на склад №2. Требуется отсортировать детали на складе №1.

2. Угнали автомобиль. Свидетель запомнил, что первой цифрой номера была 4. В базе угнанных автомобилей в этот день были следующие номера: 456, 124, 786, 435, 788, 444, 565, 127, 458, 322, 411, 531, 400, 546, 410. Нужно составить список номеров начинающихся на 4 и упорядочить его по возрастанию.

3. За неделю езды в транспорте накопились билеты с номерами 124512, 342351, 765891, 453122, 431350, 876432, 734626, 238651, 455734, 234987. Нужно отобрать "счастливые" билеты и расположить их по возрастанию.

4. Дан список людей с указанием их возраста. Для составления графика ухода сотрудников на пенсию требуется составить новый список в том порядке, в каком они будут уходить на пенсию.

5. Студенты сдали пять экзаменов. Нужно отсортировать список студентов по возрастанию общего балла по результатам сданных экзаменов.

6. В городе был один автобусный парк, куда приезжали автобусы с номерами: 11, 32, 23, 12, 6, 52, 47, 63, 69, 50, 43, 28, 35, 33, 42, 56, 55, 101. После строительства второго автопарка решили перевести туда автобусы с нечетными номерами. Для того чтобы составить расписание их движения нужно организовать список номеров автобусов второго парка, упорядочив их по убыванию.

7. Была составлена ведомость по зарплате, представленная в виде: Иванов - 166000, Сидоров - 180000, ... Требуется упорядочить этот список таким образом, чтобы размер зарплаты уменьшался.

8. На стоянке стоят автомобили со следующими номерами: 1212, 3451, 7694, 4512, 4352, 8732, 7326, 2350, 4536, 2387, 5746, 6776, 4316, 1324. Для статистики необходимо составить список автомобилей с такими номерами, сумма первых двух цифр которых равна сумме двух последних цифр,

так чтобы каждый следующий номер был меньше предыдущего.

9. Выпустили лотерейные билеты с четырехзначными номерами. Выигрышными считаются те билеты, сумма цифр которых делится на 4. Составить список выигрышных билетов, упорядоченных по убыванию.

10. Молодой человек взял номер телефона у своей знакомой, но забыл его. Он смог вспомнить только первые три цифры: 469\*\*\*. В его записной книжке были следующие номера телефонов: 456765, 469465, 469321, 616312, 576567, 469563, 567564, 469129, 675665, 469873, 569090, 469999, 564321, 469010. Составить список номеров начинающихся с цифр 469 и упорядочить их по убыванию.

11. Студенты сдали пять экзаменов. Нужно отсортировать список студентов по убыванию общего балла по результатам сданных экзаменов.

12. Выпустили лотерейные билеты с четырехзначными номерами. Выигрышными считаются те билеты, сумма первых трех цифр которых равна 8. Составить список выигрышных билетов, упорядоченных по возрастанию.

**Составить отчет по лабораторной работе и защитить его у преподавателя**

## **Лабораторная работа №9 (4 часа). Улучшенные методы сортировки**

### **Цель работы:**

- исследовать и изучить улучшенные методы сортировки на примерах метода Шелла и быстрой сортировки;
- овладеть навыками написания программ с использованием улучшенных методов сортировки на языке программирования C++.

### **Порядок выполнения работы**

- ознакомиться с краткой теорией и примерами решения задач, относящихся к сортировкам методом Шелла и быстрой;
- ответить на контрольные вопросы и получить оценку по знанию теории;
- получить задание на выполнение конкретного варианта ЛР и выполнить его;
- написать и отладить программу решения задачи на языке C++;
- составить отчет по лабораторной работе и защитить его у преподавателя.

### **Содержание отчета по ЛР**

- наименование ЛР и ее цель;
- задание на ЛР согласно варианту;
- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;
- результаты работы программы.

### **Краткая теория**

Все прямые методы сортировки фактически передвигают каждый элемент на всяком элементарном шаге на одну позицию. По этой причине они требуют порядка  $O(n^2)$  таких ша-

гов. Отсюда следует, что в основу любых улучшений должен быть положен принцип перемещения элементов на каждом шаге на возможно большие расстояния.

### Метод Шелла.

Метод Шелла является улучшением метода сортировки с помощью прямого включения и основан на сортировке посредством включением с уменьшающимися расстояниями. Сначала отдельно группируются и сортируются методом прямых включений элементы, отстоящие друг от друга на некотором расстоянии  $h_1$ , затем на расстоянии  $h_2 < h_1$  и так далее, последнее расстояние должно быть равно единице. Таким образом, если для сортировки будет использовано  $t$  расстояний, то  $h_t = 1$ ,  $h_{i+1} < h_i$ . Желательно, чтобы расстояния обеспечивали бы взаимодействие различных цепочек как можно чаще.

Рассмотрим процесс сортировки последовательности из 8 целых чисел с начальным шагом  $h_1 = 4$ , вторым шагом  $h_2 = 2$  и последним шагом  $h_3 = 1$ .

Последовательность чисел: 44, 55, 12, 42, 94, 18, 6, 67.

На рисунке ниже представлена иллюстрация сортировки методом Шелла данной последовательности.

сортировка	44	55	12	42	94	18	6	67
Четверная	44	18	6	42	94	55	12	67
Двойная	6	18	12	42	44	55	94	67
Одинарная	6	12	18	42	44	55	67	94

Сначала отдельно группируются и в группах сортируются элементы, отстоящие друг от друга на расстоянии 4. Такой процесс называется четверной сортировкой. В нашем примере 8 элементов, и каждая группа состоит из двух элементов,



то есть 1-й и 5-й элементы, 2-й и 6-й, 3-й и 7-й и, наконец, 4-й и 8-й элементы. После четверной сортировки элементы перегруппировываются - теперь каждый элемент группы отстоит от другого на 2 позиции - и вновь сортируются. Это называется двойной сортировкой. И наконец, на третьем проходе идет обычная или одинарная сортировка.

На первый взгляд можно засомневаться: если необходимо несколько процессов сортировки, причем в каждый включаются все элементы, то не добавят ли они больше работы, чем сэкономят? Однако, на каждом этапе либо сортируется относительно мало элементов, либо элементы уже довольно хорошо упорядочены и требуют сравнительно немного перестановок. Ясно, что такой метод в результате дает упорядоченный массив, и, конечно, сразу же видно, что каждый проход от предыдущих только выигрывает; также очевидно, что расстояния в группах можно уменьшать по разному, лишь бы последнее было единичным, ведь в самом плохом случае последний проход и сделает всю работу.

Приводимый ниже в псевдокоде алгоритм не ориентирован на некую определенную последовательность расстояний и использует метод прямой вставки с условным переходом. Недостатком приведенного алгоритма является нарушение технологии структурного программирования, при которой нежелательно применять безусловные переходы. Если же внутренний цикл организовать как цикл *while*, то необходима постанова «барьера», без которого при отрицательных значениях ключей происходит потеря значимости и «зависание» компьютера. При использовании метода барьера каждая из сортировок нуждается в постановке своего собственного барьера, поэтому приходится расширять массив с  $[0..N]$  до  $[-h1..N]$ , что усложнит написание программы.

Не доказано, какие расстояния дают наилучший результат, но они не должны быть множителями один другого. Д. Кнут предлагает такую последовательность шагов  $h$  (в обратном

порядке): 1, 3, 7, 15, 31, ... то есть:  $h_m = 2h_{m-1} + 1$ , а количество шагов

$$t = (\log_2 n) - 1.$$

При такой организации алгоритма его эффективность имеет порядок  $O(n^{1,2})$

## Алгоритмы

Алгоритм сортировки Шелла (псевдокод):

Обозначим

$h[1..t]$  - массив размеров шагов

$a[1..n]$  - сортируемый массив

$k$  - шаг сортировки

$x$  - значение вставляемого элемента

*const*  $t = 3$

$h(1) = 7$

$h(2) = 3$

$h(3) = 1$

*for*  $m = 1$  *to*  $t$

$k = h(m)$

*for*  $i = 1 + k$  *to*  $n$

$x = a(i)$

*for*  $j = i - k$  *to*  $1$  *step*  $-k$

*if*  $x < a(j)$  *then*

$a(j+k) = a(j)$

*else goto*  $L$  *endif*

```

    next j
    L: a(j+k) = x
    next i
    next m
    return

```

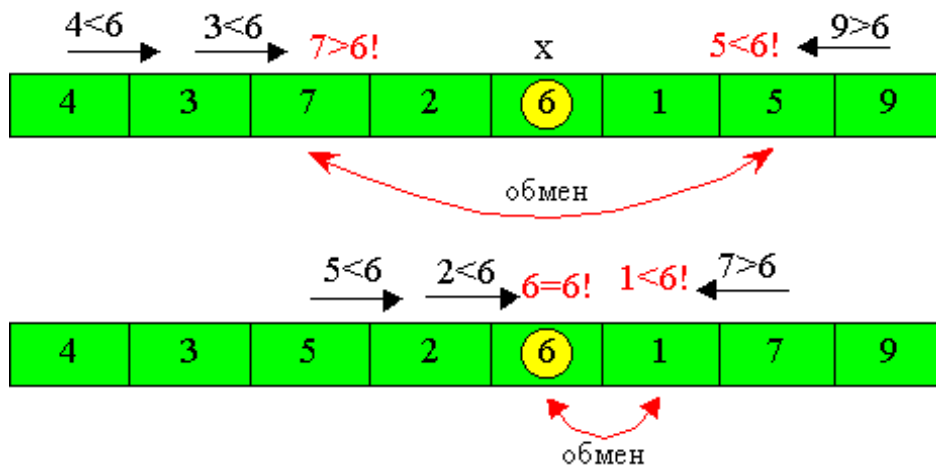
Другим улучшенным методом сортировки является так называемая быстрая сортировка (QuickSort), которая считается сортировкой разделением и носит также название быстрая сортировка Хоара.

### Quicksort - метод быстрой сортировки

Быстрая сортировка является усовершенствованием метода, основанного на обмене. В основу данной сортировки положен метод деления ключей по отношению к выбранному. Он заключается в следующем. Выбираем наугад какой-либо элемент  $x$  исходного массива. Будем просматривать массив слева до тех пор, пока не встретим  $a_i > x$ , затем будем просматривать массив справа, пока не встретим  $a_j < x$ . Поменяем местами эти два элемента и продолжим процесс просмотра до тех пор, пока оба просмотра не встретятся. В результате исходный массив окажется разбитым на две части, левая часть будет содержать элементы меньшие или равные  $x$ , а правая часть – элементы большие  $x$ . Применяв эту процедуру деления к левой и правой частям массива от точки встречи, получим четыре части и так далее, пока в каждой части окажется только один элемент. Остается решить вопрос, каким образом выбирать элемент  $x$  для деления. Если при равновероятном распределении элементов массива в качестве деления каждый раз выбирать медиану, то общее число сравнений будет  $n \cdot \log_2 n$ , а общее число обменов –  $(n \cdot \log_2 n)/6$ . При случайном выборе границы средние затраты увеличатся всего лишь в  $n \cdot \ln 2$  раза. В крайне неблагоприят-

ном случае, когда каждый раз для разделения выбирается наибольший обрабатываемый элемент массива, производительность процедуры будет наихудшая. Обычно в процедурах быстрой сортировки в качестве границы выбирают средний элемент, при этом получаются хорошие показатели производительности.

Нижеприведенный рисунок иллюстрирует процесс обменной сортировки



Слева от 6 располагают все ключи, которые меньше 6 , а справа - которые больше или равны 6.

В псевдокоде алгоритм быстрой сортировки следующий:

Sub Sort ( $L, R$ )

$i = L$

$j = R$

$x = a((L + R) \text{ div } 2)$

repeat

  while  $a(i) < x$  do

$i = i + 1$

  endwhile

  while  $a(j) > x$  do

```

     $j = j - 1$ 
endwhile
if  $i \leq j$  then
     $y = a(i)$ 
     $a(i) = a(j)$ 
     $a(j) = y$ 
     $i = i + 1$ 
     $j = j - 1$ 
endif
until  $i > j$ 
if  $L < j$  then
    sort ( $L, j$ )
endif
if  $i < R$  then
    sort ( $i, R$ )
endif
return

```

### Sub QuickSort

```

Sort ( $l, n$ )
return

```

Из всех существующих методов сортировки *QuickSort* самый эффективный. Его эффективность имеет порядок  $O(n \log_2 n)$ .

**Контрольные вопросы по теории (сортировка с помощью дерева)**

1. В чём заключается идея метода *QuickSort*?
  - выбор  $1, 2, \dots, n$  – го элемента для сравнения с остальными;
  - разделение ключей по отношению к выбранному;
  - обмен местами между соседними элементами.

2. В чем заключается суть метода Шелла?

- последовательное сравнение ключей;
- обмен местами между соседними элементами;
- сначала отдельно группируются и сортируются методом прямых включений элементы, отстоящие друг от друга на некотором расстоянии  $h_1$ , затем на расстоянии  $h_2 < h_1$  и так далее, последнее расстояние должно быть равно единице.

3. Усовершенствованием какого метода сортировки является быстрая сортировка?

- сортировки Шелла;
- сортировки методом прямого обмена;
- сортировки методом прямого выбора.

4. Какой метод сортировки является наиболее эффективным?

- быстрая сортировка;
- сортировка Шелла;
- пузырьковая сортировка.

## Примеры алгоритмов конкретных задач

Рассмотрим теперь реализацию функций сортировки методом Шелла и быстрой сортировки на C++ и примеры сортировки конкретных числовых массивов.

```
/* *****  
*/  
  
/* СОРТИРОВКА МЕТОДОМ ШЕЛЛА */  
#include <stdio.h>  
#include <conio.h>  
#define n 9  
int A[n] = {5,6,2,4,9,8,3,1,7};  
/*  
=====   
===== */  
main()
```

```

{ void Shell(int A[],int nn);
  int j;
  printf("\n Сортировка методом Шелла");
  printf("\n Исходный массив: \n");
  printf("\t");
  for (j=0; j<n; j++)
    printf("%d  ",A[j]);
  printf("\n");
  Shell(A,n);
  printf("\n Отсортированный массив :\n");
  for (j=0; j<n; j++)
    printf("%d  ",A[j]);
  printf("\n");
  getch(); return 0;
}
/*

```

==  
 \*/

```

/* ФУНКЦИЯ СОРТИРОВКИ МЕТОДОМ ШЕЛЛА */
void Shell(int A[],int nn)
{ int i,j,k,x,ii;
  k =( nn+1)/2;
  while ( k >= 1 )
  {
    for ( i=k; i<nn; i++ )
      { if ( A[i-k] > A[i] )
        { x = A[i]; j = i-k;
          M: A[j+k] = A[j];
            if ( j>k )
              { if (A[j-k] > x )
                { j = j-k;
                  goto M;
                }
              }
            A[j] = x;

```

```

/*          printf("\nk = %d x=%d: ",k,x);
for (ii=0; ii<nn; ii++)
printf(" %d ",A[ii]);*/

    }

}

/* Отладочная печать */
printf("\nk = %d ",k);
for (ii=0; ii<nn; ii++)
printf(" %d ",A[ii]);

if ( k>2 )
    k = (k+1)/2;
else
    k = k/2;
}
}
/*
***** */

/*
***** */

/*БЫСТРАЯ СОРТИРОВКА ХОАРА*/
#include <stdio.h>
#include <conio.h>
#define n 15
int A[n] = {12,22,13,4,15,6,9,11,13,7,15,10,11,8,14,};
/*
=====
== */

main()
{ void QuickSort(int A[],int L,int R);
  int j;
  printf("\n Быстрая сортировка, рекурсивная функция");

```



```

printf("\n Исходный массив: \n\t");
for (j=0; j<n; j++)
    printf("%d ",A[j]);
printf("\n Нажмите любую клавишу для продолжения");
getch();
printf("\n Отладочная печать");
QuickSort(A,0,n-1);
printf("\n Отсортированный массив :\n");
for (j=0; j<n; j++)
    printf("%d ",A[j]);
printf("\n");
getch(); return 0;
}
/*
=====
===== */
/* ФУНКЦИЯ БЫСТРОЙ СОРТИРОВКИ QuickSort */
void QuickSort(int A[],int L,int R)
{ int i,j,k,x,m;
  i = L; j = R;
  x =A[(L+R)/2];
  do
  {
    while ( A[i] < x )
      i++;
    while ( x < A[j] )
      j--;
    if (i <= j)
      { k = A[i]; A[i] = A[j]; A[j] = k;
        i++; j--;
        /* Отладочная печать */
        printf("\n i=%d j=%d x=%d: ",i-1,j+1,x);
        for (m=0; m<n; m++)
          printf(" %d",A[m]);
      }
  }
}

```

```

        }
    while (i < j);
    if (L < j)
        { printf("\t L=%d j=%d",L,j);
          QuickSort(A,L,j);
        }
    if (i < R)
        { printf("\t i=%d R=%d",i,R);
          QuickSort(A,i,R);
        }
    }
/*
***** */

```

## Задания

С использованием любого из улучшенных методов сортировки решить задачу согласно своему варианту.

1. Составить программу вывода на экран самого большого (самого малого) элемента массива  $A$ .
2. Составить программу сортировки массива  $A$  по убыванию величин его элементов.
3. В массиве  $A$  находятся элементы. Составить программу, которая сформирует массив  $B$ , в котором расположить элементы массива  $A$  в порядке убывания.
4. Дан упорядоченный массив  $A$  - числа, расположенные в порядке возрастания, и число  $a$ , которое необходимо вставить в массив  $A$ , так, чтобы упорядоченность массива сохранилась.
5. Составить программу для быстрой перестройки данного массива  $A$ , в котором элементы расположены в порядке возрастания, так, чтобы после перестройки эти же элементы оказались расположенными в порядке убывания.

6. Дан массив  $A$ , содержащий как отрицательные, так и положительные числа. Составить программу исключения из него всех отрицательных чисел, а оставшиеся положительные расположить в порядке их возрастания.

7. Составить программу, которая будет из массива  $A$  брать одно число за другим и формировать из них массив  $B$ , располагая числа в порядке возрастания.

8. Дан список авторов в форме массива  $A$ . Составить программу формирования указателя авторов в алфавитном порядке и вывести его на экран.

9. Имеется  $n$  абонентов телефонной станции. Составить программу, в которой формируется список по форме: номер телефона, фамилия (номера идут в порядке возрастания).

10. Имеется  $n$  слов различной длины, все они занесены в массив  $A$ . Составить программу упорядочения слов по возрастанию их длин.

11. Составить программу для сортировки массива  $A$  по возрастанию и убыванию модулей его элементов.

12. В отсортированный массив  $A$ . между каждой соседней парой элементов вставить число больше левого и меньше правого элемента в массиве и вывести полученный массив на экран.

**Составить отчет по лабораторной работе и защитить его у преподавателя**

## ТЕСТЫ К ЛАБОРАТОРНЫМ РАБОТАМ

Лабораторная работа 1. Полустатические структуры данных (стеки).

1. В чём особенности очереди ?
  - открыта с обеих сторон ;
  - открыта с одной стороны на вставку и удаление;
  - доступен любой элемент.
2. В чём особенности стека ?
  - открыт с обеих сторон на вставку и удаление;
  - доступен любой элемент;
  - открыт с одной стороны на вставку и удаление .
3. Какую дисциплину обслуживания принято называть FIFO?
  - стек;
  - очередь ;
  - дек.
4. Какая операция читает верхний элемент стека без удаления?
  - pop;
  - push;
  - stackpop .
5. Какого правила выборки элемента из стека ?
  - первый элемент;
  - последний элемент ;
  - любой элемент.

Лабораторная работа 2.Списковые структуры данных (одно-связные очереди).

1. Как освободить память от удаленного из списка элемента ?
  - p=getnode;
  - ptr(p)=nil;

- freenode(p) ;
  - p=lst.
2. Как создать новый элемент списка с информационным полем D ?
    - p=getnode;
    - p=getnode; info(p)=D ;
    - p=getnode; ptr(D)=lst.
  3. Как создать пустой элемент с указателем p ?
    - p=getnode ;
    - info(p);
    - freenode(p);
    - ptr(p)=lst.
  4. Сколько указателей используется в односвязных списках ?
    - 1 ;
    - 2;
    - сколько угодно.
  5. В чём отличительная особенность динамических объектов?
    - порождаются непосредственно перед выполнением программы;
    - возникают уже в процессе выполнения программы ;
    - задаются в процессе выполнения программы.
  6. При удалении элемента из кольцевого списка...
    - список разрывается;
    - в списке образуется дыра;
    - список становится короче на один элемент .
  7. Для чего используется указатель в кольцевых списках ?
    - для ссылки на следующий элемент;
    - для запоминания номера сегмента расположения элемента;
    - для ссылки на предыдущий элемент ;
    - для расположения элемента в списке памяти.
  8. Чем отличается кольцевой список от линейного ?
    - в кольцевом списке последний элемент является одновременно и первым;

- в кольцевом списке указатель последнего элемента пустой;
  - в кольцевых списках последнего элемента нет ;
  - в кольцевом списке указатель последнего элемента не пустой.
9. Сколько указателей используется в односвязном кольцевом списке ?
- 1;
  - 2;
  - сколько угодно.
10. В каких направлениях можно перемещаться в кольцевом двунаправленном списке ?
- в обоих ;
  - влево;
  - вправо.

Лабораторная работа 3. Бинарные деревья (основные процедуры).

1. Для включения новой вершины в дерево нужно найти узел, к которому её можно присоединить. Узел будет найден, если очередной ссылкой, определяющей ветвь дерева, в которой надо продолжать поиск, окажется ссылка:
  - $p = \text{right}(p)$ ;
  - $p = \text{nil}$  ;
  - $p = \text{left}(p)$ .
2. Для написания процедуры над двумя деревьями необходимо описать элемент типа запись, который содержит поля:
  - $\text{Element} = \text{Запись}$   
      $\text{Left}, \text{Right}$  : Указатели  
      $\text{Rec}$  : Запись;
  - $\text{Element} = \text{Запись}$   
      $\text{Left}$  : Указатель  
      $\text{Key}$  : Ключ

Рес : Запись;

- Element=Запись
- Left, Right : Указатели
- Key : Ключ
- Рес : Запись.

3. В памяти ЭВМ бинарное дерево удобно представлять в виде:
  - связанных линейных списков;
  - массивов;
  - связанных нелинейных списков .
4. Элемент t, на который нет ссылок:
  - корнем ;
  - промежуточным;
  - терминальным (лист).
5. Дерево называется полным бинарным, если степень исходных вершин равна:
  - 2 или 0 ;
  - 2;
  - М или 0;
  - М.

#### Лабораторная работа 4. Исследование методов линейного и бинарного поиска.

1. Где эффективен линейный поиск ?
  - в списке;
  - в массиве;
  - в массиве и в списке .
2. Какой поиск эффективнее ?
  - линейный;
  - бинарный ;
  - без разницы.
3. В чём суть бинарного поиска ?

- нахождение элемента массива  $x$  путём деления массива пополам каждый раз, пока элемент не найден ;
  - нахождение элемента  $x$  путём обхода массива;
  - нахождение элемента массива  $x$  путём деления массива.
4. Как расположены элементы в массиве бинарного поиска ?
- по возрастанию ;
  - хаотично;
  - по убыванию.
5. В чём суть линейного поиска ?
- производится последовательный просмотр от начала до конца и обратно через 2 элемента;
  - производится последовательный просмотр элементов от середины таблицы;
  - производится последовательный просмотр каждого элемента .

Лабораторная работа 5. Исследование методов поиска с перемещением в начало и транспозицией.

1. Где наиболее эффективен метод транспозиций ?
- в массивах и в списках ;
  - только в массивах;
  - только в списках.
2. В чём суть метода перестановки ?
- найденный элемент помещается в голову списка ;
  - найденный элемент помещается в конец списка;
  - найденный элемент меняется местами с последующим.
3. В чём суть метода транспозиции ?
- перестановка местами соседних элементов;
  - нахождение одинаковых элементов;
  - перестановка найденного элемента на одну позицию в сторону начала списка .
4. Что такое уникальный ключ ?
- если разность значений двух данных равна ключу;
  - если сумма значений двух данных равна ключу;



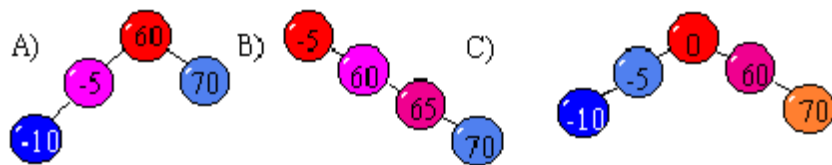
- если в таблице есть только одно данное с таким ключом

5. В чём состоит назначение поиска ?

- среди массива данных найти те данные, которые соответствуют заданному аргументу ;
- определить, что данных в массиве нет;
- с помощью данных найти аргумент.

Лабораторная работа 6. Поиск по дереву с включением и исключением.

1. В каком дереве при бинарном поиске нужно перебрать в среднем  $N/2$  элементов ?



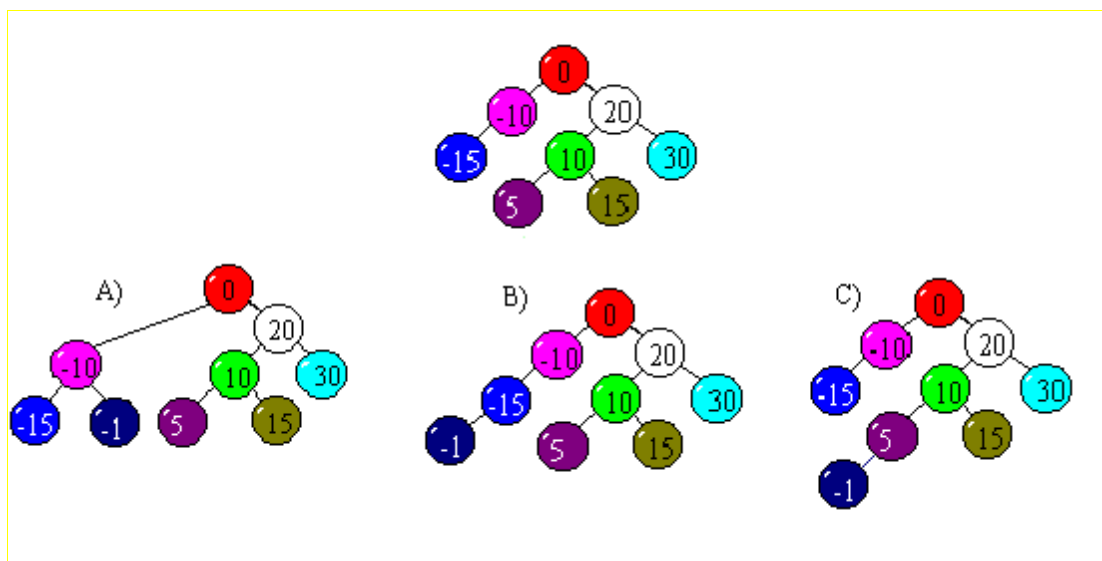
- A;
- B;
- C.

2. Сколько нужно перебрать элементов в сбалансированном дереве ?

- A)  $N/2$ ;
- B)  $\ln(N)$ ;
- C)  $\log_2(N)$ ;
- D)  $e^N$ .

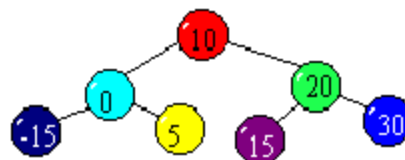
- A;
- B;
- C ;
- D.

3. Выберите вариант дерева, полученного после вставки узла -1.



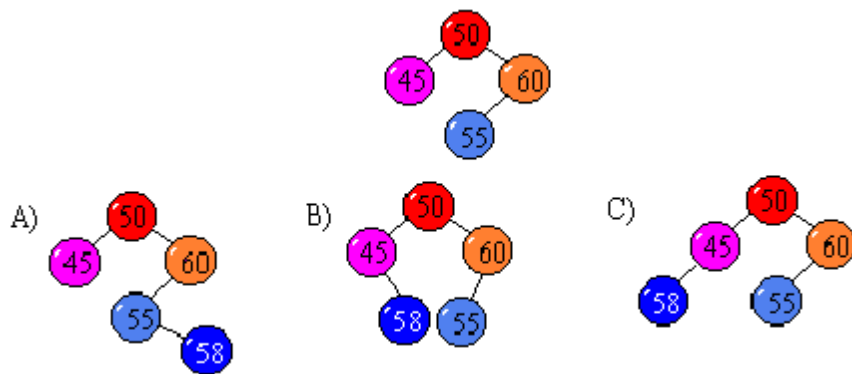
- A ;
- B;
- C.
- 

4. К какому элементу присоединить элемент 40 для вставки его в данное дерево ?



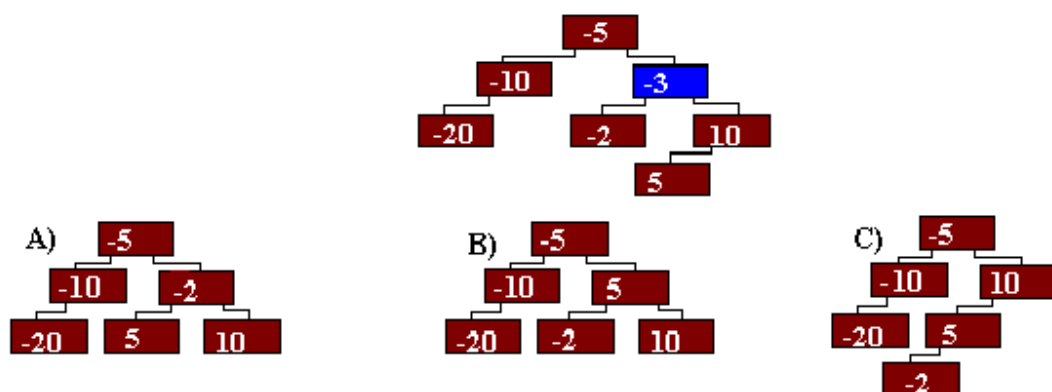
- к 30-му ;
- к 15-му;
- к -15-му;
- к 5-му.

5. Какой вид примет дерево после вставки элемента с ключом 58 ?



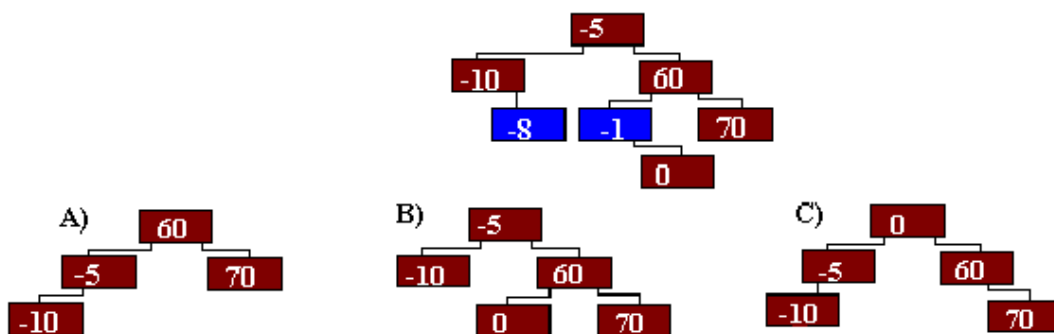
- A ;
- B;
- C.

6. Выберите вариант дерева, полученного после удаления узла  $-3$ .



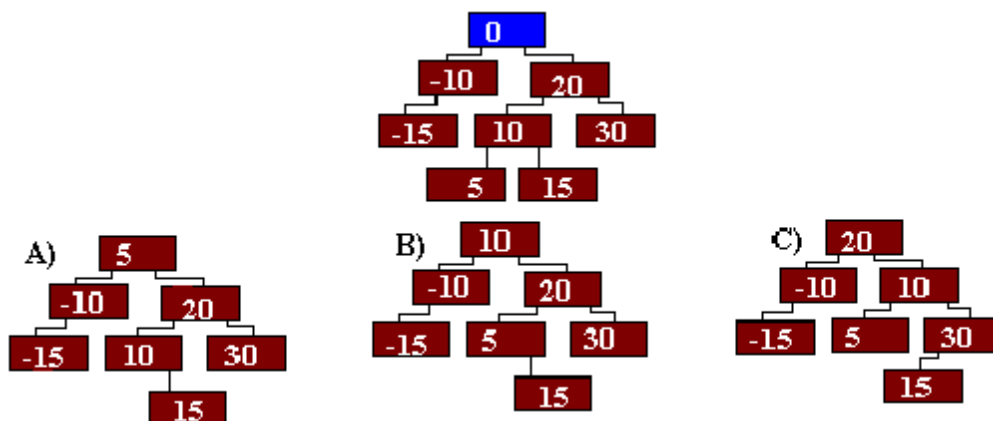
- A;
- B ;
- C.

7. Какой вариант дерева получится после удаления элемента  $-1$ , а затем  $-8$ ?



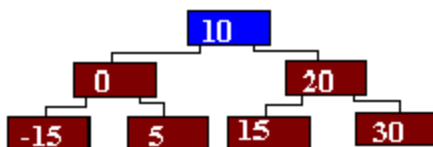
- A;
- B;
- C.

8. Выберите вариант дерева, полученного после удаления узла с индексом 0.



- A ;
- B;
- C.

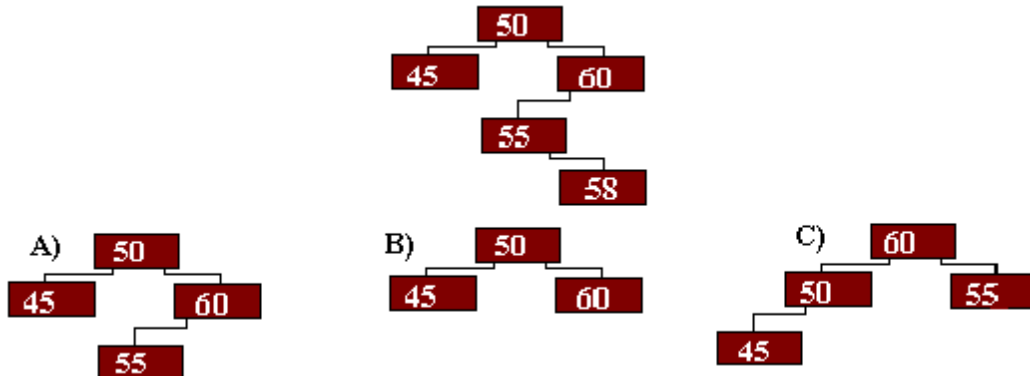
9. Какие из следующих пар чисел могут стать корнями дерева после удаления элемента 10 в соответствии с двумя способами удаления узла, имеющего двух сыновей ?



- 0 или 15;
- 0 или 20;
- 5 или 30;

- 5 или 15 .

10. Какой вид примет дерево после удаления элемента с ключом 58 ?



- A ;
- B;
- C.

Лабораторная работа 7. Сортировки методами прямого включения и выбора.

- Даны три условия окончания просеивания при сортировке прямым включением. Найдите среди них лишнее.
  - найден элемент  $a(i)$  с ключом, меньшим чем ключ  $u$ ;
  - найден элемент  $a(i)$  с ключом, большим чем ключ  $u$  ;
  - достигнут левый конец готовой последовательности.
- Какой из критериев эффективности сортировки определяется формулой  $M=0,01*n*n+10*n$  ?
  - число сравнений ;
  - время, затраченное на написание программы;
  - количество перемещений;
  - время, затраченное на сортировку.
- Как называется сортировка, происходящая в оперативной памяти ?
  - сортировка таблицы адресов;
  - полная сортировка;

- сортировка прямым включением;
  - внутренняя сортировка ;
  - внешняя сортировка.
4. Как можно сократить затраты машинного времени при сортировке большого объёма данных ?
- производить сортировку в таблице адресов ключей ;
  - производить сортировку на более мощном компьютере;
  - разбить данные на более мелкие порции и сортировать их.
5. Существуют следующие методы сортировки. Найдите ошибку.
- строгие;
  - улудшенные;
  - динамические .
6. Метод сортировки называется устойчивым, если в процессе сортировки...
- относительное расположение элементов безразлично;
  - относительное расположение элементов с равными ключами не меняется ;
  - относительное расположение элементов с равными ключами изменяется;
  - относительное расположение элементов не определено.
7. Улучшенные методы имеют значительное преимущество:
- при большом количестве сортируемых элементов ;
  - когда массив обратно упорядочен;
  - при малых количествах сортируемых элементов;
  - во всех случаях.
8. Что из перечисленных ниже понятий является одним из типов сортировки ?
- внутренняя сортировка ;
  - сортировка по убыванию;
  - сортировка данных;
  - сортировка по возрастанию.
9. Сколько сравнений требует улучшенный алгоритм сортировки ?

- $n \cdot \log(n)$  ;
  - $e^n$ ;
  - $n \cdot n/4$ .
10. К какому методу относится сортировка, требующая  $n \cdot n$  сравнений ключей ?
- прямому ;
  - бинарному;
  - простейшему;
  - обратному.

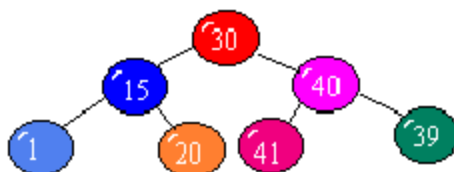
Лабораторная работа 8. Сортировки методами прямого обмена и с помощью дерева.

1. Сколько сравнений и перестановок элементов требуется в пузырьковой сортировке ?
  - $n \cdot \log(n)$ ;
  - $(n \cdot n)/4$  ;
  - $(n \cdot n - n)/2$ .
2. Сколько дополнительных переменных нужно в пузырьковой сортировке помимо массива, содержащего элементы ?
  - 0 (не нужно);
  - всего 1 элемент ;
  - n переменных (ровно столько, сколько элементов в массиве).
3. Как рассортировать массив быстрее, пользуясь пузырьковым методом ?
  - одинаково ;
  - по возрастанию элементов;
  - по убыванию элементов.
4. В чём заключается идея метода QuickSort ?
  - выбор 1,2,...n – го элемента для сравнения с остальными;
  - разделение ключей по отношению к выбранному ;
  - обмен местами между соседними элементами.

5. Массив сортируется “пузырьковым” методом. За сколько проходов по массиву самый “лёгкий” элемент в массиве окажется вверху ?

- за 1 проход ;
- за  $n-1$  проходов;
- за  $n$  проходов, где  $n$  – число элементов массива.

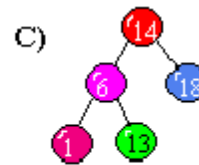
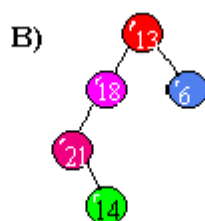
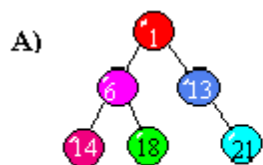
6. При обходе дерева



слева направо получаем последовательность...

- отсортированную по убыванию;
- неотсортированную ;
- отсортированную по возрастанию.

7. Какое из трёх деревьев не является строго сбалансированным ?



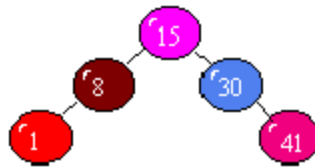
- A;
- B;
- C.

8. При обходе дерева слева направо его элемент заносится в массив...

- при втором заходе в элемент ;
- при первом заходе в элемент;
- при третьем заходе в элемент.

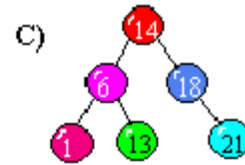
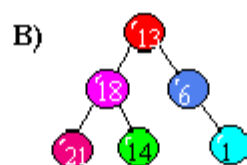
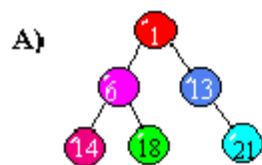
9. Элемент массива с ключом  $k=20$  необходимо вставить в изображённое дерево так, чтобы дерево осталось отсортированным. Куда его нужно вставить ?





- левым сыном элемента 30 ;
- левым сыном элемента 41;
- левым сыном элемента 8.
- 

10. При обходе какого дерева слева направо получается отсортированный по возрастанию массив ?



- A;
- B;
- C .

# **МЕТОДИЧЕСКОЕ РУКОВОДСТВО К КУРСОВОЙ РАБОТЕ**

## **Введение**

Целью курсовой работы является закрепление основ и углубление знаний в области структур и алгоритмов обработки данных в ЭВМ.

Тематика заданий на курсовую работу, приведенных в данных методических указаниях, может быть дополнена, расширена, увязана с решением актуальных научно-исследовательских задач, выполняемых на кафедре.

## **1 Требования к курсовой работе**

**1.1** Тема курсовой работы выдается каждому студенту индивидуально. В коллективных работах, в которых принимают участие два и более студентов, четко определяются объем и характер работы каждого студента. В задании формулируется задача, метод её решения.

**1.2** Курсовая работа состоит из пояснительной записки, к которой прилагается диск с отлаженными программами (пояснительная записка может быть выполнена в виде текстового файла в формате Microsoft Word).

**1.3** В пояснительную записку должны входить:

- титульный лист (приложение Б);
- задание на курсовое проектирование (приложение А);

- реферат (ПЗ, количество таблиц, рисунков, схем, программ приложений, краткая характеристика и результаты работы);
- содержание:
  - а) постановка задачи исследования;
  - б) краткая теория по теме курсовой работы;
  - в) программная реализация исследуемых алгоритмов;
  - г) программа, с помощью которой проводилось исследование;
  - д) результаты проведенного исследования;
  - е) выводы;
- список использованной литературы;
- подпись, дата.

**1.4** Пояснительная записка должна быть оформлена на листах формата А4, имеющих поля. Все листы следует сброшюровать и пронумеровать.

**1.5** Исследование алгоритмов операций над структурами данных и методов сортировок и поиска проводить при следующих фиксированных количествах элементов в структурах: 10, 100, 1000, 10000.

**1.6** Дополнительные условия выполнения курсовой работы выдаются руководителем работы.

## **2. Примерный перечень курсовых работ**

- 1) Исследование стеков.
- 2) Исследование очередей.
- 3) Исследование кольцевых структур.
- 4) Исследование полустатических структур.
- 5) Исследование линейных одно- и двусвязных списков.

- 6) Исследование деревьев бинарного поиска.
- 7) Исследование методов сортировки включением.
- 8) Исследование методов сортировки выбором.
- 9) Исследование методов сортировки обменом.
- 10) Исследование методов сортировки с помощью деревьев.
- 11) Исследование улучшенных методов сортировки.
- 12) Исследование линейного, индексного и бинарного поисков.
- 13) Исследование методов оптимизации поиска.
- 14) Исследование задач поиска по дереву.

### **3. Пример выполнения курсовой работы**

#### **3.1 Постановка задачи**

Осуществить исследование прямых методов сортировки:

- метод прямого выбора;
- метод прямой вставки;
- метод прямого обмена.

Исследование осуществить, используя массивы упорядоченных и неупорядоченных чисел по 10,100,1000 и 10000 элементов.

#### **3.2 Краткая теория**

При обработке данных важно знать и информационное поле данных, и размещение их в машине.

Различают внутреннюю и внешнюю сортировки:

– внутренняя сортировка - сортировка в оперативной памяти;

– внешняя сортировка - сортировка во внешней памяти.

Сортировка - это расположение данных в памяти в регулярном виде по их ключам. Регулярность рассматривают как

возрастание (убывание) значения ключа от начала к концу в массиве.

Если сортируемые записи занимают большой объем памяти, то их перемещение требует больших затрат. Для того, чтобы их уменьшить, сортировку производят в *таблице адресов ключей*, делают перестановку указателей, т.е. сам массив не перемещается. Это *метод сортировки таблицы адресов*.

При сортировке могут встретиться одинаковые ключи. В этом случае при сортировке желательно расположить после сортировки одинаковые ключи в том же порядке, что и в исходном файле. Это *устойчивая сортировка*.

Эффективность сортировки можно рассматривать с нескольких критериев:

- время, затрачиваемое на сортировку;
- объем оперативной памяти, требуемой для сортировки;
- время, затраченное программистом на написание программы.

Выделяем первый критерий. Можно подсчитать количество сравнений при выполнении сортировки или количество перемещений.

Пусть  $N = 0,01n^2 + 10n$  - число сравнений. Если  $n < 1000$ , то второе слагаемое больше, если  $n > 1000$ , то больше первое слагаемое.

Т. е. при малых  $n$  порядок сравнения будет равен  $n^2$ , при больших  $n$  порядок сравнения -  $n$ .

Порядок сравнения при сортировке лежит в пределах от  $0 (n \log n)$  до  $0 (n^2)$ ;  $0 (n)$  - идеальный случай.

Различают следующие методы сортировки:

- строгие (прямые) методы;
- улучшенные методы.

Строгие методы:

- 1) метод прямого включения;
- 2) метод прямого выбора;

3) метод прямого обмена.

Количество перемещений в этих трех методах примерно одинаково.

### 3.2.1 Сортировка методом прямого включения

Неформальный алгоритм

for i = 2 to n

  x = a(i)

  находим место среди a(1)...a(i) для включения x

next i

Программа на C++:

```
void Sortirovka (int n, int *a)
{
    int i,j,x,k;
    for (i=1; i<n; i++)
    {
        x=a[i]; j=i;
        while ((j>0)&&(x<a[j-1]))
        {
            a[j]=a[j-1];
            j--;
        }
        a[j]=x;
        cout<<"\n"<<i<<endl;
```

```

        for (k=0; k<n; k++)
        {
            cout<<a[k]<<" ";
        }
    }
}

```

### ***Эффективность алгоритма:***

Число сравнений  $C_{max}=n(n-1)/2$ , то есть порядок  $O(n^2)$ .  
 Количество перестановок  $M_{max}=C_{max}+3(n-1)$ , то есть порядок  $O(n^2)$ .

### **3.2.2 Сортировка методом прямого выбора**

Рассматриваем весь ряд массива и выбираем элемент, меньший или больший элемента  $a(i)$ , определяем его место в массиве -  $k$ , и затем меняем местами элемент  $a(i)$  и элемент  $a(k)$ .

#### **Программа на C++:**

```

void Sortirovka (int n, int *a)
{
    int i,j,x,k,m;
    for (i=0; i<n-1; i++)
    {

```

```

x=a[i]; k=i;
for (j=i+1; j<n; j++)
    if (a[j]<x) {k=j; x=a[k];}
a[k]=a[i]; a[i]=x;
cout<<"\n"<<i+1<<endl;
for (m=0; m<n; m++)
{
    cout<<a[m]<<" ";
}
}

```

### ***Эффективность алгоритма:***

Число сравнений  $C_{\max} = \frac{n}{2}(n-1) = \frac{n^2 - n}{2}$ .

Число перемещений  $M_{\min} = 3(n-1)$ ,  $M_{\max} = 3(n-1) + C$  (порядок  $n^2$ ).

В худшем случае сортировка прямым выбором дает порядок  $n^2$ , как и для числа сравнений, так и для числа перемещений.

### **3.2.3 Сортировка с помощью прямого обмена (пузырьковая)**

Идея:  $n-1$  раз проходят массив снизу вверх. При этом ключи попарно сравниваются. Если нижний ключ в паре меньше верхнего, то их меняют местами.

Программа на C++:

```

void Sortirovka(int n, int *a)
{

```



```

int i,j,x,k;
for (i=0; i<n; i++)
{
    for (j=n-1; j>i; j--)
        if (a[j-1]>a[j])
        { x=a[j-1];
          a[j-1]=a[j];
          a[j]=x;
        }
    cout<<"\n"<<i<<endl;
    for (k=0; k<n; k++)
    {
        cout<<a[k]<<" ";
    }
}
}

```

В нашем случае получился один проход “вхолостую”. Чтобы лишний раз не переставлять элементы, можно ввести флажок.

Улучшением пузырькового метода является шейкерная сортировка, где после каждого прохода меняют направление внутри цикла.

**Эффективность алгоритма:**

число сравнений  $C_{\max} = \frac{n}{2} \cdot \frac{(n-1)}{2},$

число перемещений  $M_{mzx} = 3n \cdot \frac{n-1}{2}$

### 3.3 Метод исследования

Данная курсовая работа заключается в исследовании прямых методов сортировки:

- метода прямого выбора;
- метода прямого включения;
- метода прямого обмена.

Исследование заключается в следующем:

Берут три массива с одинаковым количеством элементов, но один из них упорядоченный по возрастанию, второй - по убыванию, а третий - случайный. Осуществляется сортировка данных массивов и сравнивается количество перемещений элементов при сортировке первого, второго и третьего массивов, а также сравнивается количество сравнений при сортировке.

Вышеописанный способ применяется для массивов, состоящих из 10, 100, 1000 и 10000 упорядоченных и неупорядоченных элементов для всех методов сортировки.

### 3.4 Результаты исследования

#### *Сортировка 10 элементов:*

- упорядоченных по возрастанию

метод	прямого выбора	прямой вставки	прямого обмена
сравнений	45	45	45
перемещений	11	33	33

- неупорядоченных (случайных)

метод	прямого выбора	прямой вставки	прямого обмена
сравнений	45	45	45
перемещений	27	27	27

- упорядоченных по убыванию

метод	прямого выбора	прямой вставки	прямого обмена
сравнений	45	45	45
перемещений	0	0	0

### ***Сортировка 100 элементов:***

- упорядоченных по возрастанию

метод	прямого выбора	прямой вставки	прямого обмена
сравнений	4950	4950	4950
перемещений	2643	4862	4862

- неупорядоченных (случайных)

метод	прямого выбора	прямой вставки	прямого обмена
сравнений	4950	4950	4950
перемещений	2558	2558	2558

- упорядоченных по убыванию

метод	прямого выбора	прямой вставки	прямого обмена
сравнений	4950	4950	4950
перемещений	0	0	0

### ***Сортировка 1000 элементов:***

- упорядоченных по возрастанию

метод	прямого выбора	прямой вставки	прямого обмена
сравнений	499500	499500	499500
перемещений	241901	498442	498442

- неупорядоченных (случайных)

метод	прямого выбора	прямой вставки	прямого обмена
сравнений	499500	499500	499500
перемещений	244009	250366	250366

- упорядоченных по убыванию

метод	прямого выбора	прямой вставки	прямого обмена
сравнений	499500	499500	499500
перемещений	0	0	0

### ***Сортировка 10000 элементов:***

- упорядоченных по возрастанию

метод	прямого выбора	прямой вставки	прямого обмена
сравнений	49995000	49995000	49995000
перемещений		299	

перемещений	25003189	49984768	49984768
- неупорядоченных (случайных)			
метод	прямого выбора	прямой вставки	прямого обмена
сравнений	49995000	49995000	49995000
перемещений	18392665	24986578	24986578
- упорядоченных по убыванию			
метод	прямого выбора	прямой ставки	прямого обмена
сравнений	49995000	49995000	49995000
перемещений	0	0	0

### 3.5 Контрольный пример

#### ***Задание:***

Дан список, содержащий имена студентов и соответствующие им баллы рейтинга. Необходимо отсортировать данный список по убыванию баллов рейтинга.

Сортировка методом прямого включения:

**До сортировки**

Ф.И.О.	Бал рейтинга
Довженко Александр	15
Мамутов Алим	20
Дидимова Валерия	19
Березовский Владислав	16
Жданов Андрей	21
Тормозова Анастасия	17
Педан Николай	16
Тютюрев Роман	20
Гущева Елена	17
Чирков Александр	18

## После сортировки

Ф.И.О.	Бал рейтинга
Жданов Андрей	21
Мамутов Алим	20
Тютюрев Роман	20
Дидимова Валерия	19
Чирков Александр	18
Тормозова Анастасия	17
Гущева Елена	17
Березовский Владислав	16
Педан Николай	16
Довженко Александр	15

### 3.6 Выводы

По результатам исследования можно утверждать, что лучшим из прямых методов сортировки для проводимых испытаний является метод прямого выбора, так как он дает наименьшее количество сравнений и перемещений независимо от количества сортируемых элементов и их взаимного расположения в массиве.

### 3.7 Приложение

Описание функций, используемых в программе:

<b>Generate</b>	Функция, генерирующая матрицу случайных чисел
<b>obj</b>	Функция, формирующая внешний вид таблиц DataGridView с отчетами о количестве перемещений и сравнений
<b>SortPrVkl</b>	Функция, осуществляющая сортировку методом прямого включения
<b>SortPrVib</b>	Функция, осуществляющая сортировку методом прямого выбора
<b>SortObmen</b>	Функция, осуществляющая сортировку методом прямого обмена

## *Листинг основной программы*

```
#pragma once
namespace КурсоваяАиСДС {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

    int n;
    int*mas;
    /// <summary>
    /// Сводка для Form1
    /// </summary>
    public ref class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
        {
            InitializeComponent();
            //
            //TODO: добавьте код конструктора
            //
        }

    protected:
        /// <summary>
        /// Освободить все используемые ресурсы.
        /// </summary>
        ~Form1()
        {
            if (components)
            {
                delete components;
            }
        }

    private: System::Windows::Forms::Label^ label1;
    protected:
    private: System::Windows::Forms::ComboBox^ comboBox1;
    private: System::Windows::Forms::Button^ button1;
```

private: System::Windows::Forms::DataGridView^	data-
GridView1;	
private:	Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column1;	
private: System::Windows::Forms::Button^ button2;	
private: System::Windows::Forms::Label^ label2;	
private: System::Windows::Forms::Button^ button3;	
private: System::Windows::Forms::Button^ button4;	
private: System::Windows::Forms::DataGridView^	data-
GridView2;	
private:	Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column2;	
private: System::Windows::Forms::DataGridView^	data-
GridView3;	
private:	Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column3;	
private: System::Windows::Forms::Label^ label3;	
private: System::Windows::Forms::Label^ label4;	
private: System::Windows::Forms::DataGridView^	data-
GridView4;	
private:	Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column4;	
private:	Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column5;	
private:	Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column6;	
private: System::Windows::Forms::Label^ label5;	
private: System::Windows::Forms::DataGridView^	data-
GridView5;	
private:	Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column7;	
private:	Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column8;	
private:	Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column9;	
private: System::Windows::Forms::Label^ label6;	
private: System::Windows::Forms::DataGridView^	data-
GridView6;	
private:	Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column10;	
private:	Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column11;	



```

private:
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column12;

private:
    /// <summary>
    /// Требуется переменная конструктора.
    /// </summary>
    System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
    /// <summary>
    /// Обязательный метод для поддержки конструктора - не
изменяйте
    /// содержимое данного метода при помощи редактора
кода.
    /// </summary>
    void InitializeComponent(void)
    {
        this->label1 = (gcnew Sys-
tem::Windows::Forms::Label());
        this->comboBox1 = (gcnew Sys-
tem::Windows::Forms::ComboBox());
        this->button1 = (gcnew Sys-
tem::Windows::Forms::Button());
        this->dataGridView1 = (gcnew Sys-
tem::Windows::Forms::DataGridView());
        this->Column1 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
        this->button2 = (gcnew Sys-
tem::Windows::Forms::Button());
        this->label2 = (gcnew Sys-
tem::Windows::Forms::Label());
        this->button3 = (gcnew Sys-
tem::Windows::Forms::Button());
        this->button4 = (gcnew Sys-
tem::Windows::Forms::Button());
        this->dataGridView2 = (gcnew Sys-
tem::Windows::Forms::DataGridView());
        this->Column2 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
        this->dataGridView3 = (gcnew Sys-
tem::Windows::Forms::DataGridView());

```

```

        this->Column3 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
        this->label3 = (gcnew Sys-
tem::Windows::Forms::Label());
        this->label4 = (gcnew Sys-
tem::Windows::Forms::Label());
        this->dataGridView4 = (gcnew Sys-
tem::Windows::Forms::DataGridView());
        this->Column4 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
        this->Column5 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
        this->Column6 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
        this->label5 = (gcnew Sys-
tem::Windows::Forms::Label());
        this->dataGridView5 = (gcnew Sys-
tem::Windows::Forms::DataGridView());
        this->Column7 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
        this->Column8 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
        this->Column9 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
        this->label6 = (gcnew Sys-
tem::Windows::Forms::Label());
        this->dataGridView6 = (gcnew Sys-
tem::Windows::Forms::DataGridView());
        this->Column10 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
        this->Column11 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
        this->Column12 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());

        (cli::safe_cast<System::ComponentModel::ISupportInitialize^
>dataGridView1))->BeginInit();

        (cli::safe_cast<System::ComponentModel::ISupportInitialize^
>dataGridView2))->BeginInit();

```

```

        (cli::safe_cast<System::ComponentModel::ISupportInitialize^ >(this-
>dataGridView3))->BeginInit();

        (cli::safe_cast<System::ComponentModel::ISupportInitialize^ >(this-
>dataGridView4))->BeginInit();

        (cli::safe_cast<System::ComponentModel::ISupportInitialize^ >(this-
>dataGridView5))->BeginInit();

        (cli::safe_cast<System::ComponentModel::ISupportInitialize^ >(this-
>dataGridView6))->BeginInit();
        this->SuspendLayout();
        //
        // label1
        //
        this->label1->AutoSize = true;
        this->label1->Font = (gcnew Sys-
tem::Drawing::Font(L"Microsoft Sans Serif", 9.75F, Sys-
tem::Drawing::FontStyle::Bold, System::Drawing::GraphicsUnit::Point,
        static_cast<System::Byte>(204)));
        this->label1->Location = System::Drawing::Point(12,
9);

        this->label1->Name = L"label1";
        this->label1->Size = System::Drawing::Size(172, 16);
        this->label1->TabIndex = 0;
        this->label1->Text = L"Размерность массива";
        //
        // comboBox1
        //
        this->comboBox1->Font = (gcnew Sys-
tem::Drawing::Font(L"Microsoft Sans Serif", 9.75F, Sys-
tem::Drawing::FontStyle::Bold, System::Drawing::GraphicsUnit::Point,
        static_cast<System::Byte>(204)));
        this->comboBox1->FormattingEnabled = true;
        this->comboBox1->Items->AddRange(gcnew
cli::array< System::Object^ >(4) { L"10", L"100", L"1000", L"10000" });
        this->comboBox1->Location = Sys-
tem::Drawing::Point(24, 36);
        this->comboBox1->Name = L"comboBox1";
        this->comboBox1->Size = Sys-
tem::Drawing::Size(138, 24);

```

```

        this->comboBox1->TabIndex = 1;
        //
        // button1
        //
        this->button1->Font = (gcnew Sys-
tem::Drawing::Font(L"Microsoft Sans Serif", 9.75F, Sys-
tem::Drawing::FontStyle::Bold, System::Drawing::GraphicsUnit::Point,
        static_cast<System::Byte>(204)));
        this->button1->Location = Sys-
tem::Drawing::Point(201, 13);
        this->button1->Name = L"button1";
        this->button1->Size = System::Drawing::Size(149,
47);

        this->button1->TabIndex = 2;
        this->button1->Text = L"Создать массив";
        this->button1->UseVisualStyleBackColor = true;
        this->button1->Click += gcnew Sys-
tem::EventHandler(this, &Form1::button1_Click);
        //
        // dataGridView1
        //
        this->dataGridView1->AllowUserToAddRows =
false;

        this->dataGridView1->AllowUserToDeleteRows =
false;

        this->dataGridView1->
>ColumnHeadersHeightSizeMode = Sys-
tem::Windows::Forms::DataGridViewColumnHeadersHeightSizeMode::AutoSi
ze;

        this->dataGridView1->Columns->AddRange(gcnew
cli::array< System::Windows::Forms::DataGridViewColumn^ >(1) {this-
>Column1 });

        this->dataGridView1->Location = Sys-
tem::Drawing::Point(24, 75);
        this->dataGridView1->Name = L"dataGridView1";
        this->dataGridView1->RowHeadersVisible = false;
        this->dataGridView1->Size = Sys-
tem::Drawing::Size(103, 213);
        this->dataGridView1->TabIndex = 3;
        //
        // Column1
        //

```

```

        this->Column1->HeaderText = L"Массив А";
        this->Column1->Name = L"Column1";
        //
        // button2
        //
        this->button2->Font = (gcnew Sys-
tem::Drawing::Font(L"Microsoft Sans Serif", 9.75F, Sys-
tem::Drawing::FontStyle::Bold, System::Drawing::GraphicsUnit::Point,
        static_cast<System::Byte>(204)));
        this->button2->Location = Sys-
tem::Drawing::Point(404, 66);
        this->button2->Name = L"button2";
        this->button2->Size = System::Drawing::Size(227,
70);

        this->button2->TabIndex = 4;
        this->button2->Text = L"Сортировка методом пря-
мого включения";

        this->button2->UseVisualStyleBackColor = true;
        this->button2->Click += gcnew Sys-
tem::EventHandler(this, &Form1::button2_Click);
        //
        // label2
        //
        this->label2->Location = Sys-
tem::Drawing::Point(404, 143);
        this->label2->Name = L"label2";
        this->label2->Size = System::Drawing::Size(227, 59);
        this->label2->TabIndex = 5;
        //
        // button3
        //
        this->button3->Font = (gcnew Sys-
tem::Drawing::Font(L"Microsoft Sans Serif", 9.75F, Sys-
tem::Drawing::FontStyle::Bold, System::Drawing::GraphicsUnit::Point,
        static_cast<System::Byte>(204)));
        this->button3->Location = Sys-
tem::Drawing::Point(404, 143);
        this->button3->Name = L"button3";
        this->button3->Size = System::Drawing::Size(227,
67);

        this->button3->TabIndex = 6;

```

```

        this->button3->Text = L"Сортировка методом пря-
мого выбора";
        this->button3->UseVisualStyleBackColor = true;
        this->button3->Click += gcnew Sys-
tem::EventHandler(this, &Form1::button3_Click);
        //
        // button4
        //
        this->button4->Font = (gcnew Sys-
tem::Drawing::Font(L"Microsoft Sans Serif", 9.75F, Sys-
tem::Drawing::FontStyle::Bold, System::Drawing::GraphicsUnit::Point,
static_cast<System::Byte>(204)));
        this->button4->Location = Sys-
tem::Drawing::Point(404, 216);
        this->button4->Name = L"button4";
        this->button4->Size = System::Drawing::Size(227,
72);
        this->button4->TabIndex = 7;
        this->button4->Text = L"Сортировка с помощью
прямого обмена (пузырьковая)";
        this->button4->UseVisualStyleBackColor = true;
        this->button4->Click += gcnew Sys-
tem::EventHandler(this, &Form1::button4_Click);
        //
        // dataGridView2
        //
        this->dataGridView2->AllowUserToAddRows =
false;
        this->dataGridView2->AllowUserToDeleteRows =
false;
        this->dataGridView2-
>ColumnHeadersHeightSizeMode = Sys-
tem::Windows::Forms::DataGridViewColumnHeadersHeightSizeMode::AutoSi
ze;
        this->dataGridView2->Columns->AddRange(gcnew
cli::array< System::Windows::Forms::DataGridViewColumn^ >(1) {this-
>Column2});
        this->dataGridView2->Location = Sys-
tem::Drawing::Point(145, 75);
        this->dataGridView2->Name = L"dataGridView2";
        this->dataGridView2->RowHeadersVisible = false;

```

```

        this->dataGridView2->Size = System::Drawing::Size(103, 213);
        this->dataGridView2->TabIndex = 8;
        //
        // Column2
        //
        this->Column2->HeaderText = L"Массив В";
        this->Column2->Name = L"Column2";
        //
        // dataGridView3
        //
        this->dataGridView3->AllowUserToAddRows =
false;

        this->dataGridView3->AllowUserToDeleteRows =
false;

        this->dataGridView3-
>ColumnHeadersHeightSizeMode = System::Windows::Forms::DataGridViewColumnHeadersHeightSizeMode::AutoSize;
        this->dataGridView3->Columns->AddRange(gcnew
cli::array< System::Windows::Forms::DataGridViewColumn^ >(1) {this-
>Column3});
        this->dataGridView3->Location = System::Drawing::Point(266, 75);
        this->dataGridView3->Name = L"dataGridView3";
        this->dataGridView3->RowHeadersVisible = false;
        this->dataGridView3->Size = System::Drawing::Size(104, 213);
        this->dataGridView3->TabIndex = 9;
        //
        // Column3
        //
        this->Column3->HeaderText = L"Массив С";
        this->Column3->Name = L"Column3";
        //
        // label3
        //
        this->label3->AutoSize = true;
        this->label3->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 18, System::Drawing::FontStyle::Bold, System::Drawing::GraphicsUnit::Point,
static_cast<System::Byte>(204)));

```

```

        this->label3->Location = System::Drawing::Point(127, 310);
        this->label3->Name = L"label3";
        this->label3->Size = System::Drawing::Size(344, 29);
        this->label3->TabIndex = 10;
        this->label3->Text = L"Результаты исследования";
        //
        // label4
        //
        this->label4->AutoSize = true;
        this->label4->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 11.25F, System::Drawing::FontStyle::Bold, System::Drawing::GraphicsUnit::Point,
            static_cast<System::Byte>(204)));
        this->label4->Location = System::Drawing::Point(21,
351);

        this->label4->Name = L"label4";
        this->label4->Size = System::Drawing::Size(240, 18);
        this->label4->TabIndex = 11;
        this->label4->Text = L"1) Метод прямого включения";

        //
        // dataGridView4
        //
        this->dataGridView4->AllowUserToAddRows =
false;

        this->dataGridView4->AllowUserToDeleteRows =
false;

        this->dataGridView4->ColumnHeadersHeightSizeMode =
System::Windows::Forms::DataGridViewColumnHeadersHeightSizeMode::AutoSize;

        this->dataGridView4->Columns->AddRange(gcnew cli::array< System::Windows::Forms::DataGridViewColumn^ >(3) {this->Column4,
            this->Column5, this->Column6});
        this->dataGridView4->Location = System::Drawing::Point(15, 372);
        this->dataGridView4->Name = L"dataGridView4";
        this->dataGridView4->RowHeadersWidthSizeMode =
Sys-

```



```

tem::Windows::Forms::DataGridViewRowHeadersWidthSizeMode::AutoSizeT
oAllHeaders;
        this->dataGridView4->Size = System
tem::Drawing::Size(551, 89);
        this->dataGridView4->TabIndex = 12;
        //
        // Column4
        //
        this->Column4->HeaderText = L"Массив А";
        this->Column4->Name = L"Column4";
        //
        // Column5
        //
        this->Column5->HeaderText = L"Массив В";
        this->Column5->Name = L"Column5";
        //
        // Column6
        //
        this->Column6->HeaderText = L"Массив С";
        this->Column6->Name = L"Column6";
        //
        // label5
        //
        this->label5->AutoSize = true;
        this->label5->Font = (gcnew Sys-
tem::Drawing::Font(L"Microsoft Sans Serif", 11.25F, Sys-
tem::Drawing::FontStyle::Bold, System::Drawing::GraphicsUnit::Point,
        static_cast<System::Byte>(204)));
        this->label5->Location = System::Drawing::Point(21,
464);

        this->label5->Name = L"label5";
        this->label5->Size = System::Drawing::Size(214, 18);
        this->label5->TabIndex = 13;
        this->label5->Text = L"2) Метод прямого выбора";
        //
        // dataGridView5
        //
        this->dataGridView5->AllowUserToAddRows =
false;

        this->dataGridView5->AllowUserToDeleteRows =
false;

```

```

        this->dataGridView5-
>ColumnHeadersHeightSizeMode = System::Windows::Forms::DataGridViewColumnHeadersHeightSizeMode::AutoSize;
        this->dataGridView5->Columns->AddRange(gcnew
cli::array< System::Windows::Forms::DataGridViewColumn^ >(3) {this-
>Column7,
        this->Column8, this->Column9});
        this->dataGridView5->Location = System::Drawing::Point(15, 499);
        this->dataGridView5->Name = L"dataGridView5";
        this->dataGridView5->RowHeadersWidthSizeMode =
Sys-
tem::Windows::Forms::DataGridViewRowHeadersWidthSizeMode::AutoSizeT
oAllHeaders;
        this->dataGridView5->Size = System::Drawing::Size(551, 85);
        this->dataGridView5->TabIndex = 14;
        //
        // Column7
        //
        this->Column7->HeaderText = L"Массив А";
        this->Column7->Name = L"Column7";
        //
        // Column8
        //
        this->Column8->HeaderText = L"Массив В";
        this->Column8->Name = L"Column8";
        //
        // Column9
        //
        this->Column9->HeaderText = L"Массив С";
        this->Column9->Name = L"Column9";
        //
        // label6
        //
        this->label6->AutoSize = true;
        this->label6->Font = (gcnew Sys-
tem::Drawing::Font(L"Microsoft Sans Serif", 9.75F, Sys-
tem::Drawing::FontStyle::Bold, System::Drawing::GraphicsUnit::Point,
        static_cast<System::Byte>(204)));

```

```

587);
        this->label6->Location = System::Drawing::Point(24,
        this->label6->Name = L"label6";
        this->label6->Size = System::Drawing::Size(398, 16);
        this->label6->TabIndex = 15;
        this->label6->Text = L"3) Метод прямого обмена(
пузырьковая сортировка)";
        //
        // dataGridView6
        //
        this->dataGridView6->AllowUserToAddRows =
false;
        this->dataGridView6->AllowUserToDeleteRows =
false;
        this->dataGridView6-
>ColumnHeadersHeightSizeMode = System::
Windows::Forms::DataGridViewColumnHeadersHeightSizeMode::AutoSize;
        this->dataGridView6->Columns->AddRange(gcnew
cli::array< System::Windows::Forms::DataGridViewColumn^ >(3) {this-
>Column10,
        this->Column11, this->Column12});
        this->dataGridView6->Location = System::
Drawing::Point(15, 615);
        this->dataGridView6->Name = L"dataGridView6";
        this->dataGridView6->RowHeadersWidthSizeMode =
System::
Windows::Forms::DataGridViewRowHeadersWidthSizeMode::AutoSizeTo
oAllHeaders;
        this->dataGridView6->Size = System::
Drawing::Size(551, 82);
        this->dataGridView6->TabIndex = 16;
        //
        // Column10
        //
        this->Column10->HeaderText = L"Массив А";
        this->Column10->Name = L"Column10";
        //
        // Column11
        //
        this->Column11->HeaderText = L"Массив В";
        this->Column11->Name = L"Column11";

```

```

//
// Column12
//
this->Column12->HeaderText = L"Массив С";
this->Column12->Name = L"Column12";
//
// Form1
//
this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
this->ClientSize = System::Drawing::Size(659, 710);
this->Controls->Add(this->dataGridView6);
this->Controls->Add(this->label6);
this->Controls->Add(this->dataGridView5);
this->Controls->Add(this->label5);
this->Controls->Add(this->dataGridView4);
this->Controls->Add(this->label4);
this->Controls->Add(this->label3);
this->Controls->Add(this->dataGridView3);
this->Controls->Add(this->dataGridView2);
this->Controls->Add(this->button4);
this->Controls->Add(this->button3);
this->Controls->Add(this->label2);
this->Controls->Add(this->button2);
this->Controls->Add(this->dataGridView1);
this->Controls->Add(this->button1);
this->Controls->Add(this->comboBox1);
this->Controls->Add(this->label1);
this->Name = L"Form1";
this->Text = L"Пример курсовой работы по АИСД
C++";

```

```

(cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this->dataGridView1))->EndInit();

```

```

(cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this->dataGridView2))->EndInit();

```

```

(cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this->dataGridView3))->EndInit();

```

```

        (cli::safe_cast<System::ComponentModel::ISupportInitialize^
>dataGridView4))->EndInit();

        (cli::safe_cast<System::ComponentModel::ISupportInitialize^
>dataGridView5))->EndInit();

        (cli::safe_cast<System::ComponentModel::ISupportInitialize^
>dataGridView6))->EndInit();
        this->ResumeLayout(false);
        this->PerformLayout();

    }
#pragma endregion
    void Generate (int N,int*A) // функция генерирующая
матрицу случайных чисел
    {
        Random^ rand=gcnew Random();
        for (int i=0;i<N;i++)
            A[i]=rand->Next(100);
    }
    void obj (DataGridView^ datagridview1)
    {
        datagridview1->RowCount=2;
        datagridview1->Rows[0]->HeaderCell-
>Value="Сравнений";
        datagridview1->Rows[1]->HeaderCell-
>Value="Перемещений";
    }

    void SortPrVkl(DataGridView^ dataGridView1, Data-
GridView^ dataGridView2, int a)
    {
        int p=0,s=0; //p-переменная для подсчета кол-ва пе-
ремещений, s-сравнений
        int i,j,x;
        for (i=1; i<dataGridView1->RowCount; i++)
        {
            x=System::Convert::ToInt32(dataGridView1-
>Rows[i]->Cells[0]->Value);
            for (j=i;j>0;j--)
            {

```

```

        s++;

        if(x<System::Convert::ToInt32(dataGridView1->Rows[j-1]->Cells[0]-
>Value))
        {
            dataGridView1->Rows[j]-
>Cells[0]->Value=dataGridView1->Rows[j-1]->Cells[0]->Value;
            p++;
        }
    }
    if (dataGridView1->Rows[j]->Cells[0]-
>Value==dataGridView1->Rows[j+1]->Cells[0]->Value)
    {
        dataGridView1->Rows[j]->Cells[0]-
>Value=x;
        p++;
    }
}
dataGridView2->Rows[0]->Cells[a]->Value=s;
dataGridView2->Rows[1]->Cells[a]->Value=p;
}

void SortPrVib(DataGridView^ dataGridView1, DataGridView^ dataGridView2, int a)
{
    int p=0,s=0; //p-переменная для подсчета кол-ва пе-
ремещений, s-сравнений
    int i,j,x,k;
    for (i=0; i<dataGridView1->RowCount-1; i++)
    {
        x=System::Convert::ToInt32(dataGridView1-
>Rows[i]->Cells[0]->Value); k=i;
        for (j=i+1; j<dataGridView1->RowCount; j++)
        { s++;
            if (System::Convert::ToInt32(dataGridView1-
>Rows[j]->Cells[0]->Value)<x)
            { k=j;
x=System::Convert::ToInt32(dataGridView1->Rows[k]->Cells[0]->Value);
                dataGridView1->Rows[k]->Cells[0]-
>Value=dataGridView1->Rows[i]->Cells[0]->Value;
                p++;
            }
        }
    }
}

```

```

        dataGridView1->Rows[i]->Cells[0]->Value=x;
        p++;
    }
}
dataGridView5->Rows[0]->Cells[a]->Value=s;
dataGridView5->Rows[1]->Cells[a]->Value=p;
}
void SortObmen (DataGridView^ dataGridView1, DataGridView^ dataGridView2, int a)
{
    int p=0,s=0; //p-переменная для подсчета кол-ва перемещений, s-сравнений
    int i,j,x;
    for (i=0; i<dataGridView1->RowCount; i++)
    {
        for (j=dataGridView1->RowCount-1; j>i; j--)
        {
            s++;
            if (System.Convert.ToInt32(dataGridView1->Rows[j-1]->Cells[0]->Value)>System.Convert.ToInt32(dataGridView1->Rows[j]->Cells[0]->Value))
            {
                x=System.Convert.ToInt32(dataGridView1->Rows[j-1]->Cells[0]->Value);
                dataGridView1->Rows[j-1]->Cells[0]->Value=dataGridView1->Rows[j]->Cells[0]->Value;
                dataGridView1->Rows[j]->Cells[0]->Value=x;
                p=p+2;
            }
        }
    }
    dataGridView2->Rows[0]->Cells[a]->Value=s;
    dataGridView2->Rows[1]->Cells[a]->Value=p;
}
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{
    dataGridView2->RowCount=Convert.ToInt32(comboBox1->SelectedItem);

```

```

        dataGridView3-
>RowCount=Convert::ToInt32(comboBox1->SelectedItem);
        n=Convert::ToInt32(comboBox1-
>SelectedItem);

        mas = new int[n];
        Generate(n,mas);
        dataGridView1->RowCount=n;
        for( int i=0;i<n; i++)
            dataGridView1->Rows[i]->Cells[0]-
>Value=mas[i];

        for( int i=0;i<dataGridView1->RowCount;
i++)
            dataGridView2->Rows[i]->Cells[0]-
>Value=i;

        for( int i=0;i<dataGridView1->RowCount;
i++)
        {
            dataGridView3->Rows[i]->Cells[0]-
>Value=dataGridView1->RowCount-i-1;
        }
        obj(dataGridView4);
        obj(dataGridView5);
        obj(dataGridView6);
    }

private: System::Void button2_Click(System::Object^ sender, Sys-
tem::EventArgs^ e)
    {
        SortPrVkl(dataGridView1,dataGridView4,0);
        SortPrVkl(dataGridView2,dataGridView4,1);
        SortPrVkl(dataGridView3,dataGridView4,2);
    }

private: System::Void button3_Click(System::Object^ sender, Sys-
tem::EventArgs^ e)
    {
        SortPrVib(dataGridView1,dataGridView5,0);
        SortPrVib(dataGridView2,dataGridView5,1);
        SortPrVib(dataGridView3,dataGridView5,2);
    }

```



```

private: System::Void button4_Click(System::Object^ sender, Sys-
tem::EventArgs^ e)
{
    SortObmen(dataGridView1,dataGridView6,0);
    SortObmen(dataGridView2,dataGridView6,1);
    SortObmen(dataGridView3,dataGridView6,2);
}
};
}

```

### *Листинг контрольного примера*

```

#pragma once
#include<vcclr.h>
namespace Конкретныйпример {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
    struct Student
    {

        gcroot <String^> fio;
        int bal;

    };

    /// <summary>
    /// Сводка для Form1
    /// </summary>
    public ref class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
        {
            InitializeComponent();
            //

```

```

        //TODO: добавьте код конструктора
        //
    }

protected:
    /// <summary>
    /// Освободить все используемые ресурсы.
    /// </summary>
    ~Form1()
    {
        if (components)
        {
            delete components;
        }
    }

private: System::Windows::Forms::Label^ label1;
protected:
private:    System::Windows::Forms::DataGridView^      data-
GridView1;
private:                                     Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column1;
private:                                     Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column2;
private: System::Windows::Forms::Button^ button1;
private:    System::Windows::Forms::DataGridView^      data-
GridView2;
private:                                     Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column3;
private:                                     Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn^ Column4;

private:
    /// <summary>
    /// Требуется переменная конструктора.
    /// </summary>
    System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
    /// <summary>
    /// Обязательный метод для поддержки конструктора - не
изменяйте

```

кода. /// содержимое данного метода при помощи редактора

```
/// </summary>
void InitializeComponent(void)
{
    this->label1 = (gcnew Sys-
tem::Windows::Forms::Label());
    this->dataGridView1 = (gcnew Sys-
tem::Windows::Forms::DataGridView());
    this->Column1 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
    this->Column2 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
    this->button1 = (gcnew Sys-
tem::Windows::Forms::Button());
    this->dataGridView2 = (gcnew Sys-
tem::Windows::Forms::DataGridView());
    this->Column3 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());
    this->Column4 = (gcnew Sys-
tem::Windows::Forms::DataGridViewTextBoxColumn());

    (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>dataGridView1))->BeginInit();

    (cli::safe_cast<System::ComponentModel::ISupportInitialize^>(this-
>dataGridView2))->BeginInit();
    this->SuspendLayout();
    //
    // label1
    //
    this->label1->AutoSize = true;
    this->label1->Font = (gcnew Sys-
tem::Drawing::Font(L"Microsoft Sans Serif", 12, stat-
ic_cast<System::Drawing::FontStyle>((System::Drawing::FontStyle::Bold |
System::Drawing::FontStyle::Italic)),
        System::Drawing::GraphicsUnit::Point, stat-
ic_cast<System::Byte>(204)));
    this->label1->Location = System::Drawing::Point(71,
9);

    this->label1->Name = L"label1";
    this->label1->Size = System::Drawing::Size(231, 20);
```

```

        this->label1->TabIndex = 0;
        this->label1->Text = L"Информация о студентах:";
        //
        // dataGridView1
        //
        this->dataGridView1-
>ColumnHeadersHeightSizeMode = System::Windows::Forms::DataGridViewColumnHeadersHeightSizeMode::AutoSize;
        this->dataGridView1->Columns->AddRange(gcnew
cli::array< System::Windows::Forms::DataGridViewColumn^ >(2) {this-
>Column1,
        this->Column2});
        this->dataGridView1->Location = System::Drawing::Point(24, 63);
        this->dataGridView1->Name = L"dataGridView1";
        this->dataGridView1->Size = System::Drawing::Size(351, 193);
        this->dataGridView1->TabIndex = 1;
        //
        // Column1
        //
        this->Column1->HeaderText = L"Ф.И.О.";
        this->Column1->Name = L"Column1";
        this->Column1->Width = 210;
        //
        // Column2
        //
        this->Column2->HeaderText = L"Бал рейтинга";
        this->Column2->Name = L"Column2";
        //
        // button1
        //
        this->button1->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif", 9.75F, System::Drawing::FontStyle::Bold, System::Drawing::GraphicsUnit::Point,
        static_cast<System::Byte>(204)));
        this->button1->Location = System::Drawing::Point(24, 277);
        this->button1->Name = L"button1";
        this->button1->Size = System::Drawing::Size(351,
42);

```

```

        this->button1->TabIndex = 2;
        this->button1->Text = L"Отсортировать список ме-
тодом прямого включения";
        this->button1->UseVisualStyleBackColor = true;
        this->button1->Click += gcnw System::
tem::EventHandler(this, &Form1::button1_Click);
        //
        // dataGridView2
        //
        this->dataGridView2-
>ColumnHeadersHeightSizeMode = System::
tem::Windows::Forms::DataGridViewColumnHeadersHeightSizeMode::AutoSi
ze;
        this->dataGridView2->Columns->AddRange(gcnw
cli::array< System::Windows::Forms::DataGridViewColumn^ >(2) {this-
>Column3,
        this->Column4});
        this->dataGridView2->Location = System::
tem::Drawing::Point(24, 339);
        this->dataGridView2->Name = L"dataGridView2";
        this->dataGridView2->Size = System::
tem::Drawing::Size(351, 193);
        this->dataGridView2->TabIndex = 3;
        //
        // Column3
        //
        this->Column3->HeaderText = L"Ф.И.О.";
        this->Column3->Name = L"Column3";
        this->Column3->Width = 210;
        //
        // Column4
        //
        this->Column4->HeaderText = L"Бал рейтинга";
        this->Column4->Name = L"Column4";
        //
        // Form1
        //
        this->AutoScaleDimensions = System::
tem::Drawing::SizeF(6, 13);
        this->AutoScaleMode = System::
tem::Windows::Forms::AutoScaleMode::Font;
        this->ClientSize = System::Drawing::Size(407, 545);

```

```

        this->Controls->Add(this->dataGridView2);
        this->Controls->Add(this->button1);
        this->Controls->Add(this->dataGridView1);
        this->Controls->Add(this->label1);
        this->Name = L"Form1";
        this->Text = L"Контрольный пример";
        this->Load += gcnew System::EventHandler(this,
&Form1::Form1_Load);

        (cli::safe_cast<System::ComponentModel::ISupportInitialize^ >(this-
>dataGridView1))->EndInit();

        (cli::safe_cast<System::ComponentModel::ISupportInitialize^ >(this-
>dataGridView2))->EndInit();
        this->ResumeLayout(false);
        this->PerformLayout();

    }
#pragma endregion
    private: System::Void button1_Click(System::Object^ sender,
System::EventArgs^ e)
    {

        dataGridView2->RowCount=11;
        Student Pers[1];
        Student Arr[10]; //копируем введенные в
DataGridView1 данные в структуру Student
        int i,j;
        for(i=0;i<10;i++)
        {

Arr[i].fio=System::Convert::ToString(dataGridView1->Rows[i]->Cells[0]-
>Value);

Arr[i].bal=System::Convert::ToInt32(dataGridView1->Rows[i]->Cells[1]-
>Value);

        }

        for (i=8; i>=0; i--)
        {
            Pers[0].bal=Arr[i].bal;
            Pers[0].fio=Arr[i].fio;

```

```

        j=i;
        while (
(j<9)&&(Pers[0].bal<Arr[j+1].bal))
        {
            Arr[j].bal=Arr[j+1].bal;
            Arr[j].fio=Arr[j+1].fio;
            j++;
        }

        Arr[j].bal=Pers[0].bal;
        Arr[j].fio=Pers[0].fio;
    }

    for(int i=0;i<10;i++)
    {
        dataGridView2->Rows[i]->Cells[0]-
>Value=Arr[i].fio;
        dataGridView2->Rows[i]->Cells[1]-
>Value=Arr[i].bal;
    }

}

private: System::Void Form1_Load(System::Object^ sender, Sys-
tem::EventArgs^ e)
{
    dataGridView1->RowCount=11;
}

};
}

```

## ЗАКЛЮЧЕНИЕ

В учебном пособии были рассмотрены наиболее распространенные оперативные структуры данных и алгоритмы их обработки, которые традиционно применяются при создании программных систем и комплексов. В силу ограниченности объемом курса не было уделено внимания таким структурам, как В - деревья и графы, в разделе поиска опущен раздел хеширования. Однако, на базе уже рассмотренного материала эти разделы могут быть легко изучены самостоятельно.

Современное состояние и тенденции развития вычислительной техники как основного инструмента информатики таковы, что наряду с увеличением функциональности вычислительная техника приобретает свойства, позволяющие работать на ней пользователю, не разбирающемуся в программировании. Бурно развиваются в последнее время локальные, корпоративные и глобальные вычислительные сети. Создаются мощные накопители данных. Другими словами, основные процессы информационных технологий (обработка, обмен и накопление данных) поднялись на следующую ступень, что, естественно, требует новых подходов к организации данных в ЭВМ и созданию соответствующих систем программирования. Определяющими факторами к этому являются современные требования к пользовательскому интерфейсу и мультимедийные системы. Появились структуры графических данных и более крупные, интегральные информационные единицы - объекты. Следствием явилось бурное развитие объектно-ориентированных систем программирования: Visual BASIC, Visual PASCAL, Visual C<sup>++</sup> и т.д., используемых для создания программ, в основе которых лежит обработка объектных структур данных. Обмен объектными структурами в сетях вызван развитием сетевых операционных систем: Intranetware, Solaris, Windows NT и т.д. Обработка данных на



многопроцессорных вычислительных системах потребовала создания новых структур данных, основанных на абстрактных представлениях и новых языков программирования: Modula 2, ADA, OCCAM.

Таким образом, развитие информационных технологий, их проникновение во все области жизнедеятельности человека требуют компьютерного отображения информации в виде соответствующих структур данных и, естественно, каждый новый поступающий шаг информатики будет сопровождаться соответствующим шагом в области структур данных.

## ЛИТЕРАТУРА

1. Бертисс А.Т. Структуры данных./-Пер.с англ., М.:Статистика,1974.
2. Вирт Н. Алгоритмы и структуры данных./- М.: Мир,1989.
3. Д. Райли. Абстракция и структуры данных. Вводный курс. /-М.: Мир, 1993.
4. Костин А.Е.,Шаньгин В.Ф. Организация и обработка структур данных в вычислительных системах./- М.: Высшая школа,1987.
5. Ленгсам и др. Структуры данных для персональных ЭВМ./ - М.: Мир, 1989.
6. Трамбле Ж., Соренсон П. Введение в структуры данных./ - М.: Машиностроение, 1982.

## ПРИЛОЖЕНИЕ.

### ТЕСТЫ С ОТВЕТАМИ

Лабораторная работа 1. Полустатические структуры данных (стеки).

1. В чём особенности очереди ?
  - открыта с обеих сторон (верный);
  - открыта с одной стороны на вставку и удаление;
  - доступен любой элемент.
2. В чём особенности стека ?
  - открыт с обеих сторон на вставку и удаление;
  - доступен любой элемент;
  - открыт с одной стороны на вставку и удаление (верный).
3. Какую дисциплину обслуживания принято называть FIFO ?
  - стек;
  - очередь (верный);
  - дек.
4. Какая операция читает верхний элемент стека без удаления ?
  - pop;
  - push;
  - stackpop (верный).
5. Каково правило выборки элемента из стека ?
  - первый элемент;
  - последний элемент (верный);
  - любой элемент.

Лабораторная работа 2. Списковые структуры данных (одно-связные очереди).

1. Как освободить память от удаленного из списка элемента ?

- p=getnode;
  - ptr(p)=nil;
  - freenode(p) (верный);
  - p=lst.
2. Как создать новый элемент списка с информационным полем D ?
- p=getnode;
  - p=getnode; info(p)=D (верный);
  - p=getnode; ptr(D)=lst.
3. Как создать пустой элемент с указателем p ?
- p=getnode (верный);
  - info(p);
  - freenode(p);
  - ptr(p)=lst.
4. Сколько указателей используется в односвязных списках ?
- 1 (верный);
  - 2;
  - сколько угодно.
5. В чём отличительная особенность динамических объектов ?
- порождаются непосредственно перед выполнением программы;
  - возникают уже в процессе выполнения программы (верный);
  - задаются в процессе выполнения программы.
  -
6. При удалении элемента из кольцевого списка...
- список разрывается;
  - в списке образуется дыра;
  - список становится короче на один элемент (верный).
7. Для чего используется указатель в кольцевых списках ?
- для ссылки на следующий элемент;
  - для запоминания номера сегмента расположения элемента;
  - для ссылки на предыдущий элемент (верный);

- для расположения элемента в списке памяти.
8. Чем отличается кольцевой список от линейного ?
- в кольцевом списке последний элемент является одновременно и первым;
  - в кольцевом списке указатель последнего элемента пустой;
  - в кольцевых списках последнего элемента нет (верный);
  - в кольцевом списке указатель последнего элемента не пустой.
9. Сколько указателей используется в односвязном кольцевом списке ?
- 1(верный);
  - 2;
  - сколько угодно.
10. В каких направлениях можно перемещаться в кольцевом двунаправленном списке ?
- в обоих (верный);
  - влево;
  - вправо.

Лабораторная работа 3. Бинарные деревья (основные процедуры).

1. Для включения новой вершины в дерево нужно найти узел, к которому её можно присоединить. Узел будет найден, если очередной ссылкой, определяющей ветвь дерева, в которой надо продолжать поиск, окажется ссылка:
- $p = \text{right}(p)$ ;
  - $p = \text{nil}$  (верный);
  - $p = \text{left}(p)$ .
2. Для написания процедуры над двумя деревьями необходимо описать элемент типа запись, который содержит поля:
- $\text{Element} = \text{Запись}$   
     $\text{Left}, \text{Right}$  : Указатели

Рес : Запись;

- Element=Запись  
Left : Указатель  
Key : Ключ  
Рес : Запись;
- Element=Запись (верный)  
Left, Right : Указатели  
Key : Ключ  
Рес : Запись.

3. В памяти ЭВМ бинарное дерево удобно представлять в виде:

- связанных линейных списков;
- массивов;
- связанных нелинейных списков (верный).

4. Элемент  $t$ , на который нет ссылок:

- корнем (верный);
- промежуточным;
- терминальным (лист).

5. Дерево называется полным бинарным, если степень исходов вершин равна:

- 2 или 0 (верный);
- 2;
- $M$  или 0;
- $M$ .

Лабораторная работа 4. Исследование методов линейного и бинарного поиска.

1. Где эффективен линейный поиск ?

- в списке;
- в массиве;
- в массиве и в списке (верный).

2. Какой поиск эффективнее ?

- линейный;
- бинарный (верный);
- без разницы.

3. В чём суть бинарного поиска ?

- нахождение элемента массива  $x$  путём деления массива пополам каждый раз, пока элемент не найден (верный);
- нахождение элемента  $x$  путём обхода массива;
- нахождение элемента массива  $x$  путём деления массива.

4. Как расположены элементы в массиве бинарного поиска ?

- по возрастанию (верный);
- хаотично;
- по убыванию.

5. В чём суть линейного поиска ?

- производится последовательный просмотр от начала до конца и обратно через 2 элемента;
- производится последовательный просмотр элементов от середины таблицы;
- производится последовательный просмотр каждого элемента (верный).

Лабораторная работа 5. Исследование методов поиска с перемещением в начало и транспозицией.

1. Где наиболее эффективен метод транспозиций ?

- в массивах и в списках (верный);
- только в массивах;
- только в списках.

2. В чём суть метода перестановки ?

- найденный элемент помещается в голову списка (верный);
- найденный элемент помещается в конец списка;
- найденный элемент меняется местами с последующим.

3. В чём суть метода транспозиции ?

- перестановка местами соседних элементов;

- нахождение одинаковых элементов;
- перестановка найденного элемента на одну позицию в сторону начала списка (верный).

4. Что такое уникальный ключ ?

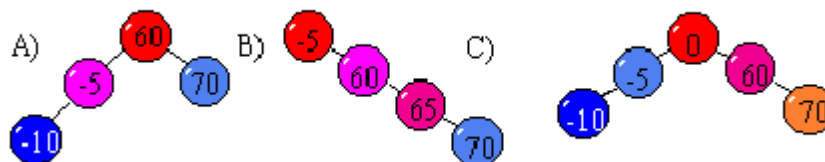
- если разность значений двух данных равна ключу;
- если сумма значений двух данных равна ключу;
- если в таблице есть только одно данное с таким ключом (верный).

5. В чём состоит назначение поиска ?

- среди массива данных найти те данные, которые соответствуют заданному аргументу (верный);
- определить, что данных в массиве нет;
- с помощью данных найти аргумент.

Лабораторная работа 6. Поиск по дереву с включением и исключением.

1. В каком дереве при бинарном поиске нужно перебрать в среднем  $N/2$  элементов ?



- A;
- B (верный);
- C.

2. Сколько нужно перебрать элементов в сбалансированном дереве ?

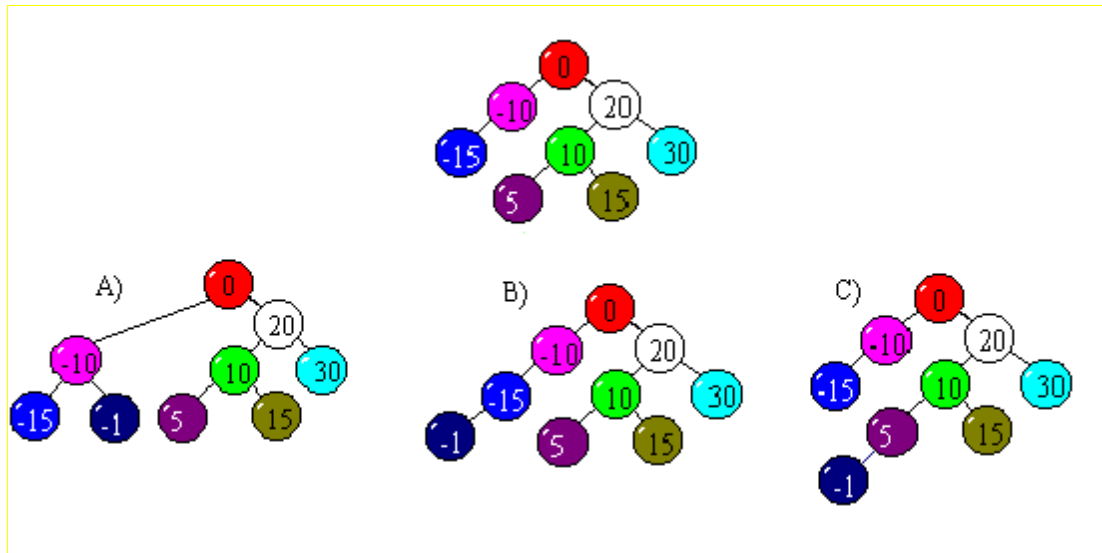
- E)  $N/2$ ;
- F)  $\ln(N)$ ;
- G)  $\log_2(N)$ ;
- H)  $e^N$ .

- A;



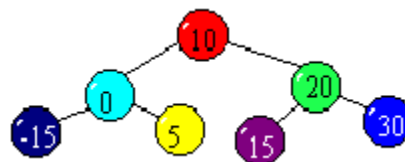
- В;
- С (верный);
- D.

3. Выберите вариант дерева, полученного после вставки узла -1.



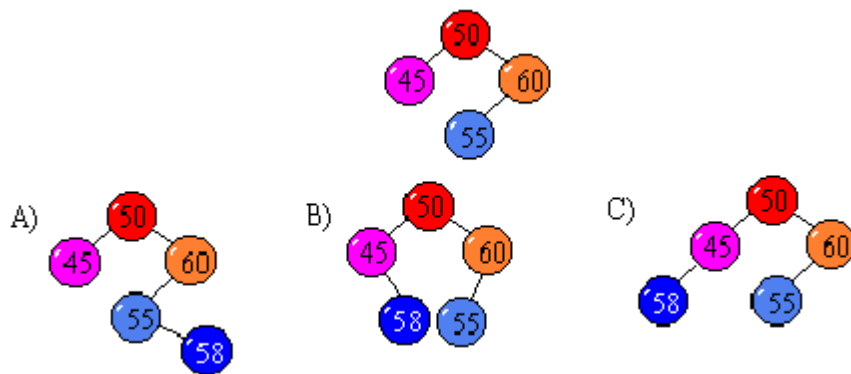
- A (верный);
- В;
- С.

4. К какому элементу присоединить элемент 40 для вставки его в данное дерево ?



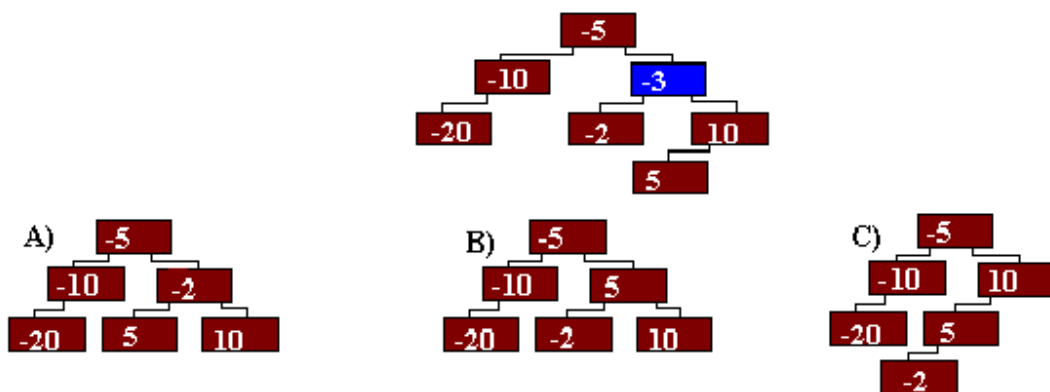
- к 30-му (верный);
- к 15-му;
- к -15-му;
- к 5-му.

5. Какой вид примет дерево после вставки элемента с ключом 58 ?



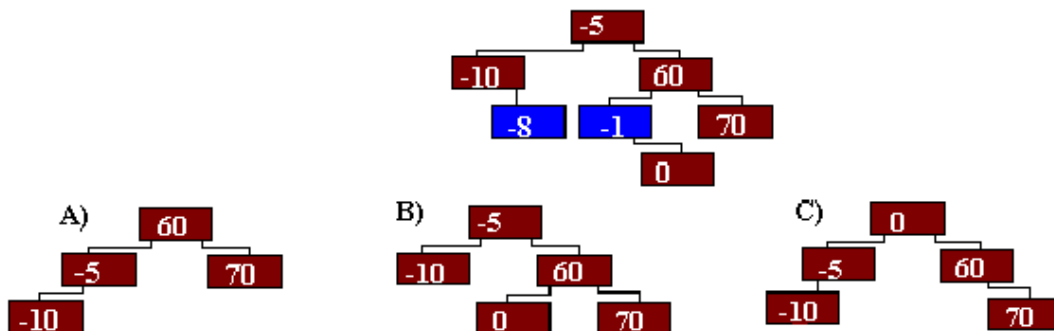
- A (верный);
- B;
- C.

6. Выберите вариант дерева, полученного после удаления узла  $-3$ .



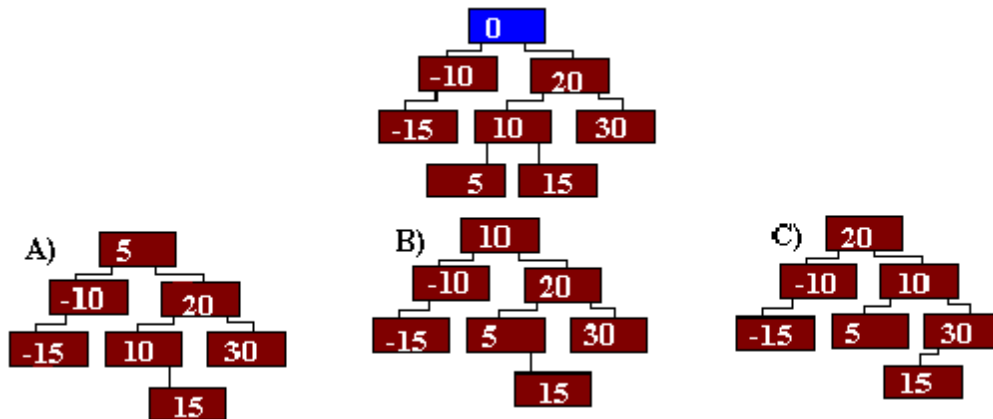
- A;
- B (верный);
- C.

7. Какой вариант дерева получится после удаления элемента  $-1$ , а затем  $-8$  ?



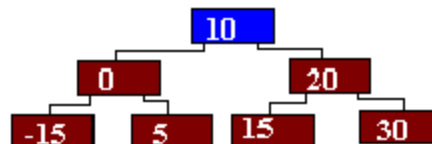
- А;
- В (верный);
- С.

8. Выберите вариант дерева, полученного после удаления узла с индексом 0.



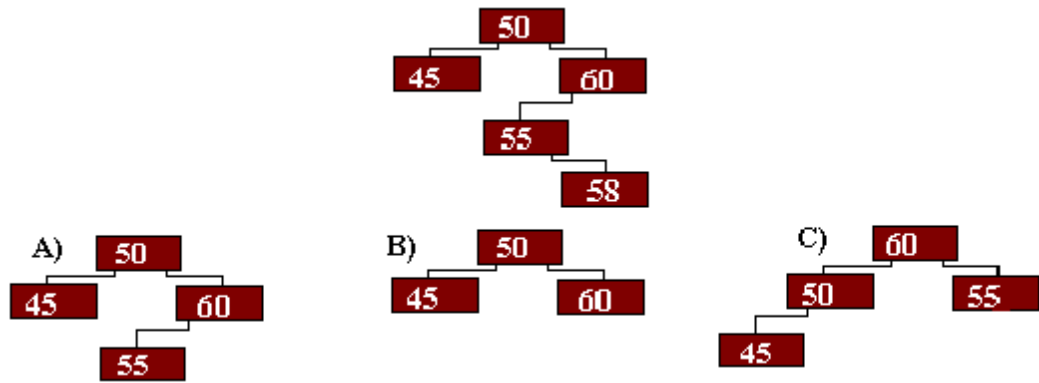
- А (верный);
- В;
- С.

9. Какие из следующих пар чисел могут стать корнями дерева после удаления элемента 10 в соответствии с двумя способами удаления узла, имеющего двух сыновей ?



- 0 или 15;
- 0 или 20;
- 5 или 30;
- 5 или 15 (верный).

10. Какой вид примет дерево после удаления элемента с ключом 58 ?



- А (верный);
- В;
- С.

Лабораторная работа 7. Сортировки методами прямого включения и выбора.

1. Даны три условия окончания просеивания при сортировке прямым включением. Найдите среди них лишнее.
  - найден элемент  $a(i)$  с ключом, меньшим чем ключ у  $x$ ;
  - найден элемент  $a(i)$  с ключом, большим чем ключ у  $x$  (верный);
  - достигнут левый конец готовой последовательности.
2. Какой из критериев эффективности сортировки определяется формулой  $M=0,01*n*n+10*n$  ?
  - число сравнений (верный);
  - время, затраченное на написание программы;
  - количество перемещений;
  - время, затраченное на сортировку.
3. Как называется сортировка, происходящая в оперативной памяти ?
  - сортировка таблицы адресов;
  - полная сортировка;
  - сортировка прямым включением;
  - внутренняя сортировка (верный);
  - внешняя сортировка.

4. Как можно сократить затраты машинного времени при сортировке большого объёма данных ?
- производить сортировку в таблице адресов ключей (верный);
  - производить сортировку на более мощном компьютере;
  - разбить данные на более мелкие порции и сортировать их.
5. Существуют следующие методы сортировки. Найдите ошибку.
- строгие;
  - улучшенные;
  - динамические (верный).
6. Метод сортировки называется устойчивым, если в процессе сортировки...
- относительное расположение элементов безразлично;
  - относительное расположение элементов с равными ключами не меняется (верный);
  - относительное расположение элементов с равными ключами изменяется;
  - относительное расположение элементов не определено.
7. Улучшенные методы имеют значительное преимущество:
- при большом количестве сортируемых элементов (верный);
  - когда массив обратно упорядочен;
  - при малых количествах сортируемых элементов;
  - во всех случаях.
8. Что из перечисленных ниже понятий является одним из типов сортировки ?
- внутренняя сортировка (верный);
  - сортировка по убыванию;
  - сортировка данных;
  - сортировка по возрастанию.
9. Сколько сравнений требует улучшенный алгоритм сортировки ?

- $n \cdot \log(n)$  (верный);
  - $e^n$ ;
  - $n \cdot n/4$ .
10. К какому методу относится сортировка, требующая  $n \cdot n$  сравнений ключей ?
- прямому (верный);
  - бинарному;
  - простейшему;
  - обратному.

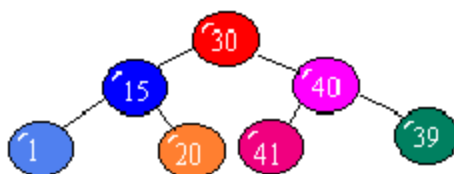
Лабораторная работа 8. Сортировки методом прямого обмена и с помощью дерева.

1. Сколько сравнений и перестановок элементов требуется в пузырьковой сортировке ?
  - $n \cdot \log(n)$ ;
  - $(n \cdot n)/4$  (верный);
  - $(n \cdot n - n)/2$ .
2. Сколько дополнительных переменных нужно в пузырьковой сортировке помимо массива, содержащего элементы ?
  - 0 (не нужно);
  - всего 1 элемент (верный);
  - $n$  переменных (ровно столько, сколько элементов в массиве).
3. Как рассортировать массив быстрее, пользуясь пузырьковым методом ?
  - одинаково (верный);
  - по возрастанию элементов;
  - по убыванию элементов.
4. В чём заключается идея метода QuickSort ?
  - выбор  $1, 2, \dots, n$  – го элемента для сравнения с остальными;
  - разделение ключей по отношению к выбранному (верный);
  - обмен местами между соседними элементами.

5. Массив сортируется “пузырьковым” методом. За сколько проходов по массиву самый “лёгкий” элемент в массиве окажется вверху ?

- за 1 проход (верный);
- за  $n-1$  проходов;
- за  $n$  проходов, где  $n$  – число элементов массива.

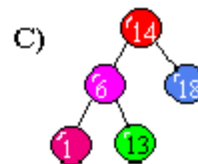
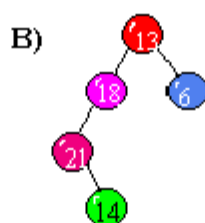
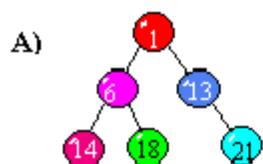
6. При обходе дерева



слева направо получаем последовательность...

- отсортированную по убыванию;
- неотсортированную (верный);
- отсортированную по возрастанию.

7. Какое из трёх деревьев не является строго сбалансированным ?

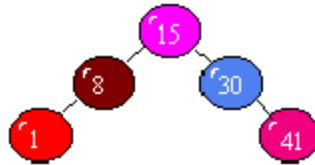


- А;
- В(верный);
- С.

8. При обходе дерева слева направо его элемент заносится в массив...

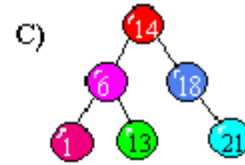
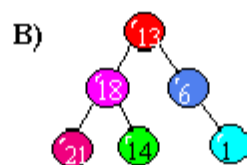
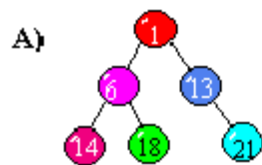
- при втором заходе в элемент (верный);
- при первом заходе в элемент;
- при третьем заходе в элемент.

9. Элемент массива с ключом  $k=20$  необходимо вставить в изображённое дерево так, чтобы дерево осталось отсортированным. Куда его нужно вставить ?



- левым сыном элемента 30 (верный);
- левым сыном элемента 41;
- левым сыном элемента 8.

10. При обходе какого дерева слева направо получается отсортированный по возрастанию массив ?



- А;
- В;
- С (верный).



У ч е б н о е   и з д а н и е

**Лойко** Валерий Иванович  
**Лаптев** Сергей Владимирович

Структуры и алгоритмы  
обработки данных

*Учебное пособие*

В авторской редакции.

Подписано в печать       .11.2013 г. Формат  $60 \times 84^{1/16}$ .  
Тираж 125 экз. Усл. печ. л. – 21,43. Уч.-изд. л. – 20,1 .  
Заказ № .

Типография  
Кубанского государственного аграрного университета

350044, г. Краснодар, ул. Калинина, 13

