

Министерство сельского хозяйства Российской Федерации
**Федеральное государственное образовательное учреждение
высшего профессионального образования "Кубанский государ-
ственный аграрный университет" (КубГАУ)**

Кафедра компьютерных технологий и систем

В.И. Лойко, В.Н. Лаптев, С.В. Лаптев, А.В. Параскевов,
В.В. Ткаченко

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Методические указания к лабораторным занятиям студентов спе-
циальностей

"Информационные системы и технологии" и
"Прикладная информатика"

Краснодар
2013

УДК 004.42 (073.8)
ББК 32.973.26-018
А-45

Рецензенты:

Барановская Т.П. – профессор, доктор экономических наук, заведующая кафедрой системного анализа и обработки информации Кубанского государственного аграрного университета (г. Краснодар);

Ключко В.И. – профессор, доктор технических наук, заведующий кафедрой вычислительной техники и автоматизированных систем (ВТиАС) Кубанского государственного технологического университета (г. Краснодар)

Алгоритмы и структуры данных: Методические указания к лабораторным занятиям студентов специальностей "Информационные системы и технологии" и "Прикладная информатика" / В. И. Лойко, В. Н. Лаптев, С. В. Лаптев, А. В. Параскевов, В. В. Ткаченко. – Краснодар: ФГОУ ВПО КубГАУ, 2011. – 142 с.

Составлены в соответствии с рабочей программой дисциплины **Алгоритмы и структуры данных** для студентов второго курса специальности "Информационные системы и технологии" "Прикладная информатика".

Содержит описание лабораторных работ, методические указания к их выполнению и требования к оформлению отчета.

Работа выполнена по решению методической комиссии факультета прикладной информатики и кафедры компьютерных технологий и систем (протокол №9 от 28.02.2013 г.)

Содержание

ВВЕДЕНИЕ	4
1. СТРУКТУРЫ ДАННЫХ	7
ЛЗ №1 /4 часа/. Полустатические структуры данных.....	7
ЛЗ №2 /4 часа/. Списковые структуры данных	25
ЛЗ №3 /4 часа/. Бинарные деревья (создание и обход)	38
2. АЛГОРИТМЫ ПОИСКА	50
ЛЗ №4 /4 часа/. Исследование методов линейного и бинарного поиска	50
ЛЗ №5 /4 часа/. Исследование методов оптимизации поиска	67
ЛЗ №6 /4 часа/. Поиск по дереву с включением и исключением	76
3. АЛГОРИТМЫ СОРТИРОВКИ.....	95
ЛЗ №7 /4 часа/. Сортировки методами прямого включения и выбора	95
ЛЗ №8 /4 часа/. Сортировки методом прямого обмена и с помощью дерева.	109
ЛЗ №9 /4 часа/. Улучшенные методы сортировки.	120
ЗАКЛЮЧЕНИЕ	131
ЛИТЕРАТУРА	133
ПРИЛОЖЕНИЯ.....	135
Приложение 1. Календарно-тематический план изучения дисциплине	135
Приложение 2. Программа самостоятельной работы студентов по дисциплине	136
Приложение 3. Вопросы для подготовки к экзамену по дисциплине	139
Приложение 4. Перечень учебно-методических материалов, используемых по дисциплине	140
Приложение 5. Программное обеспечение, используемое при изучении дисциплины	141

ВВЕДЕНИЕ

Компьютер - это машина, которая обрабатывает информацию. Информация – это данные, наделяемые определенным качественным содержанием (смыслом). Изучение науки об ЭВМ предполагает изучение того, каким образом структуры данных формируются внутри цифровой электронно-вычислительной машины (ЦЭВМ), как обрабатываются и могут эффективно использоваться для быстрого решения практических задач. Следовательно, для изучения учебной дисциплины "Алгоритмы и структуры данных" студенту особенно важно понять базовые, фундаментальные основы организации информации и алгоритмы результативной работы с ней.

Так как вычислительная техника базируется на изучении информации – структур данных, которым присваивается определенный смысл, то первый возникающий вопрос заключается в том, что такое информация. В этом контексте понятие "информация" в вычислительной технике сходно с понятием "слова" и структур из слов, которые наполнены конкретным смыслом. Это как в любом языке. Каждая буква алфавита любого языка в отдельности не несет в себе никакого содержания (смысла), а вот структуры из букв алфавита – слова уже наполнены глубоким смыслом. Из букв алфавита людьми по общепринятым правилам конструируются простые и сложные структуры (слова и предложения), наделяемые конкретным содержанием. Они и составляют основу информационного общения людей и накопления знаний - "работающих" на человека смысловых структур языка, позволяющих ему моделировать реальные процессы развития природы и общества, а путем практической проверки моделей познавать их, обеспечивая тем самым свою успешную адаптации к непрерывным изменениям среды обитания.

Базовой единицей информации, т.е. ее единственной "буквой" является бит, который может принимать два взаимоисключающих значения. Для представления двух возможных состояний некоторого бита используются двоичные цифры - нуль и единица. Если живое или неживое устройство может находиться более чем в двух состояниях, то тот факт, что оно находится в одном из этих состояний, уже требует нескольких битов информации.

Число битов, необходимых для кодирования символа (цифры или буквы) в конкретной ЦЭВМ, в большинстве таблиц кодировки равно восьми, и такая группа битов называется байтом.

Компьютерная память представляет собой совокупность битов, в любой момент функционирования в ЦЭВМ каждый из битов памяти имеет значение 0 или 1 (сброшен или установлен). Состояние бита называется его значением или содержимым.

Биты в памяти ЦЭВМ группируются в элементы большего размера, например в байты. Несколько байтов объединяются в группы, называемые словами (как в любом языке). Каждому байту назначается адрес (числовой код), идентифицирующий конкретный элемент памяти среди множества аналогичных элементов. Этот адрес называется ячейкой, а содержимое ячейки есть значение битов, которые ее составляют.

Итак, мы видим, что бит как первичный элемент данных в ЦЭВМ не имеет сам по себе конкретного смысла. Однако с некоторой конкретной битовой комбинацией или структурой битов (элементарных элементов позиционной двоичной системы счисления) может быть связано любое смысловое значение. Именно интерпретация битовой комбинации придает ей заданный смысл, т.е. делает ее информацией.

Метод интерпретации битовой информации часто называется типом данных. В силу использования различных систем счисления (как правило, кратных двоичной), разные ЦЭВМ имеют свой набор типов данных. Здесь 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 как разные классы данных. Эти объявления детерминируют объем памяти, необходимый для размещения отдельных элементов и структур из них, способ интерпретации их элементов и другие важные детали, необходимые для разработки эффективных алгоритмов и программ компьютерной обработки структур данных. Под компьютерной программой понимается алгоритм обработки данных, представленный в понятной исполнителю форме, т.е. ЦЭВМ. По существу объявления сообщают интерпретатору точное значение используемых символов машинных операций.

В методических указаниях к лабораторным занятиям по дисциплине "Алгоритмы и структуры данных" авторы делают упор на строгое выполнение студентами всех представленных в нем заданий по разработке алгоритмов и соответствующих им компьютерных программ на алгоритмическом языке высокого уровня C++. Это, с нашей точки зрения, позволит студентам не только быстро освоить типовые структуры данных и эффективные алгоритмы их компьютерной обработки, но создаст добротный фундамент для научного просвещения их духа и реа-

лизации честолюбивых замыслов стать настоящими профессионалами в области перспективного научно-прикладного направления – информационные технологии (ИТ). Мы желаем их творческих успехов в этой весьма трудной, но архи важной для них самостоятельной работе накопления личного практического опыта успешной работы с алгоритмами обработки все более усложняющихся и структур данных (знаний).

1. СТРУКТУРЫ ДАННЫХ

ЛЗ №1 /4 часа/. Полустатические структуры данных

1.1. 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++;

1.2. 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.

1.3. 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.

1.4. Краткая теория

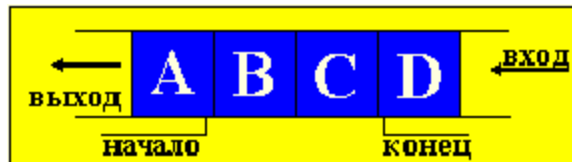
Понятие очереди всем хорошо известно из повседневной жизни. Элементами очереди в общем случае являются заказы на то или иное обслуживание: выбить чек на нужную сумму в кассе магазина; получить нужную информацию в справочном бюро, выполнить очередную операцию по обработке детали на данном станке в автоматической 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.

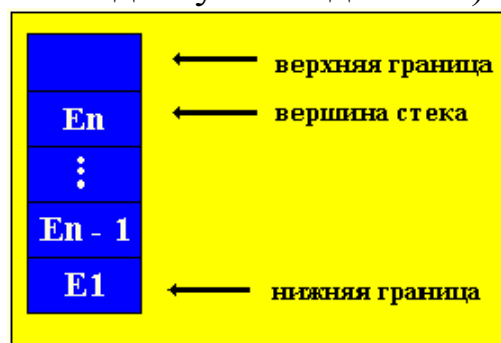
Различают два основных вида очередей, отличающихся по дисциплине обслуживания находящихся в них элементов:

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).



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 “пусто”
  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 эле-

мента 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

1.5. 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.

1.6. Примеры алгоритмов и приложений (стеки и операций 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 элементов
#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 пуст"<<endl;
    else
    do
    {cout<<s[next-1]<<" "<<endl;next--;}
    while(next!=0);
}

```

```

void Clear()
{ next=0; }

```

```

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

```

```
void Stacktop()
```

```
{
    if (next==0) cout<<"Ошибка: стек пуст"<<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 считывания эле-
ментов"<<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 удалением 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 *st* 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 УДАЛЕ-
НИЯ ЭЛЕМЕНТА */
int Peek_el(STC *s,EL *el)
{ if (s->un == NULL)

```

```

    return -2; /* стек не создан */
if (s->uv < 0)
    return -1; /* стек пусм */
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) ");
      flushall();
      if (getchar() != 'y')
        { printf("\n Работаем со старым стеком");
          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;
}

```

```

}
/* **** Конечная строка файла **** */

```

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 Элемент вы-
бран\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 исполняемого фай-
ла\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;
```

```
}
```

```
/* **** */
```

1.7. 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.

1.8. Составить отчет по лабораторной работе и защитить его у преподавателя

ЛЗ №2 /4 часа/. Списковые структуры данных

2.1. 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++;

2.2. 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.

2.3. 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.

2.4. 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.

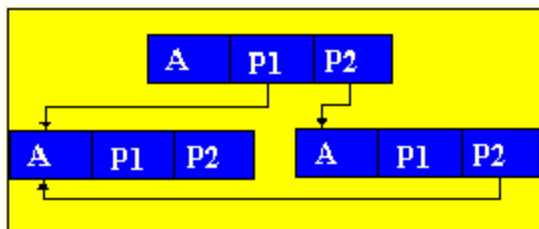
Однако использование при программировании только статических объектов может вызвать определенные трудности, особенно с точки зрения получения эффективной машинной программы, а такая эффективность бывает чрезвычайно важной при решении ряда задач. Дело в

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.

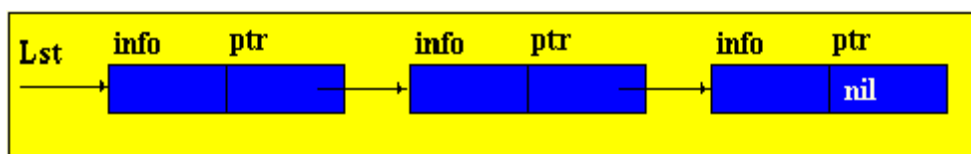
Наиболее распространенными динамическими структурами являются связанные списки. С точки зрения логического представления различают *линейные* и *нелинейные* списки. В линейных списках связи строго упорядочены. Указатель предыдущего элемента дает адрес последующего элемента или наоборот.

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.

А. Линейные однонаправленные списки (односвязные списки)



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

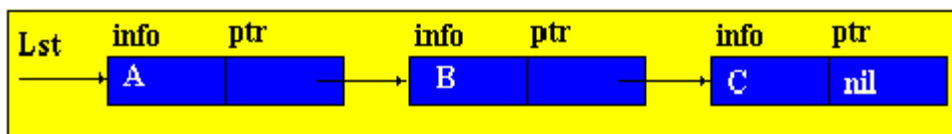
- информационное поле *info* и
- поле указателя *ptr*.

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*.

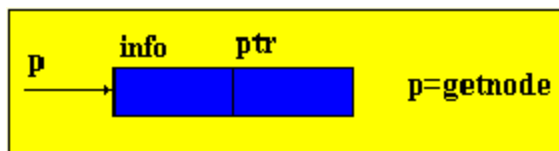
Операции с односвязными списками.

1. Вставка элемента в начало односвязного списка.

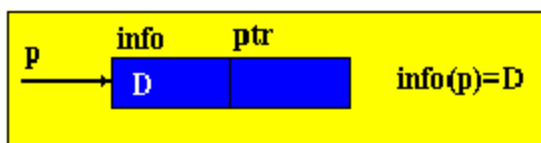


Вставим в начало данного списка элемент, информационное поле которого содержит переменную *D*. Чтобы это осуществить, необходимо произвести следующие действия :

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



- b) Информационному полю созданного элемента присвоить значение *D*.

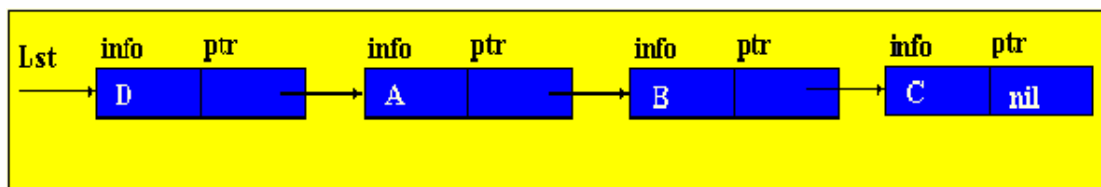


- c) Связать новый элемент со списком.

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

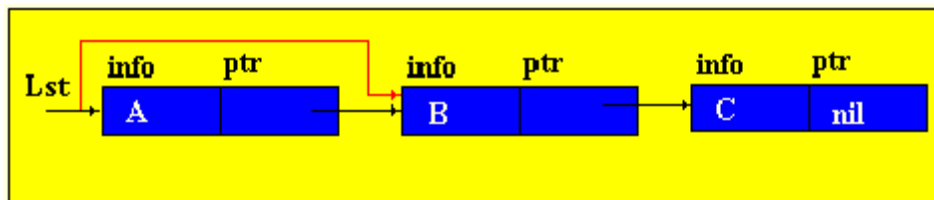
- d) Перенести указатель *lst* на начало списка.

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).

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

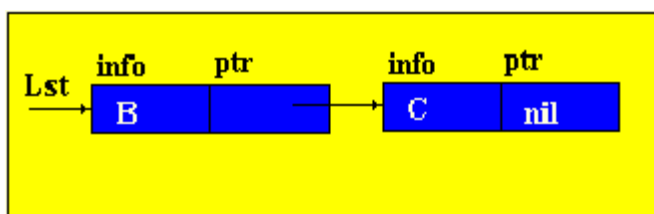
с) Перенести указатель lst на новое начало списка.

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

д) Удалить элемент на который указывает указатель p.

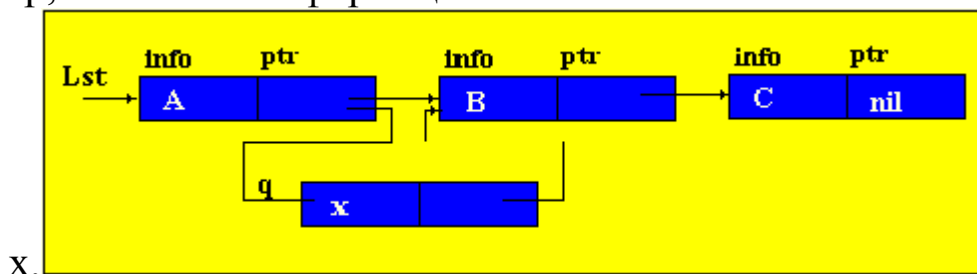
$\text{Freenode}(P)$

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.

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

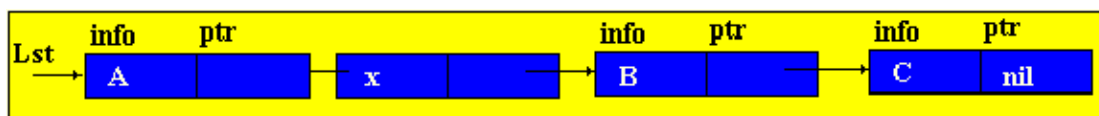
с) Связать элемент x с элементом B.

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

д) Связать элемент A с элементом x.

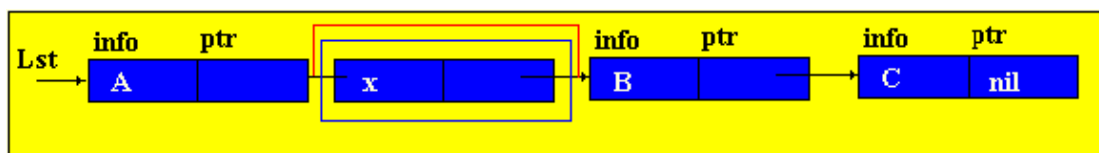
$\text{Ptr}(p) = Q$ - это значит, что следующим за элементом A будет элемент на который указывает указатель Q.

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 :

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

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

б) Поставить за элементом A элемент B.

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

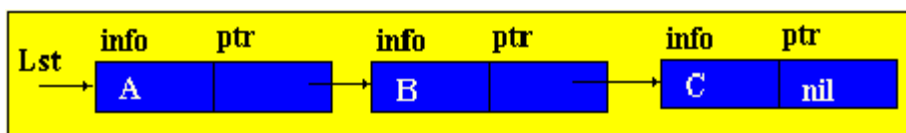
с) Запомнить информацию, которая содержится в поле info удаляемого элемента.

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

д) Удалим элемент с указателем Q.

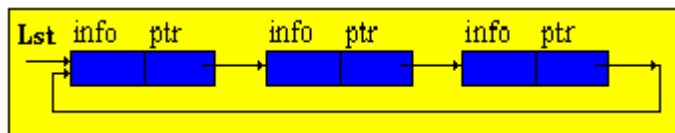
$\text{Freenode}(Q)$

Окончательно:



В. 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

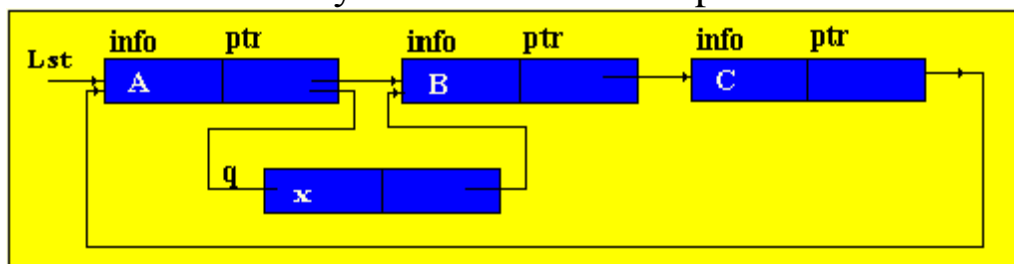
Чтобы осуществить эту операцию необходимо произвести следующие действия:

a) Создать пустой элемент на который указывает указатель q
 $q = \text{getnode}$

b) Внести x в информационное поле созданного элемента
 $\text{info}(q) = x$

c) Связать элемент X с элементом B

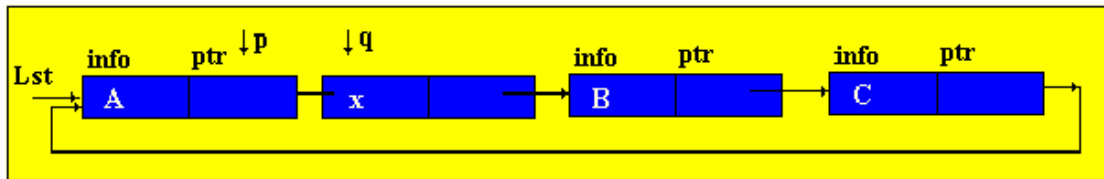
$\text{ptr}(q) = \text{ptr}(p)$ - это означает, что указателю созданного элемента присваивается значение указателя элемента p .



d) Связать элемент A с элементом X

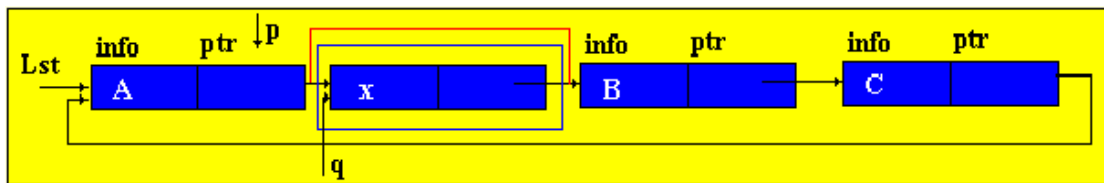
$\text{ptr}(p) = q$ - это означает, что следующим за элементом A будет элемент на который указывает указатель q .

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

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

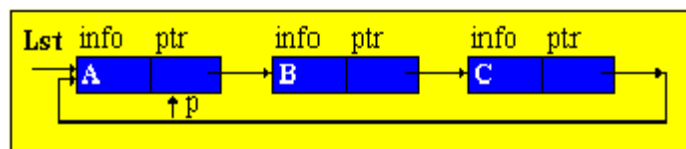
с) Запомнить информацию, которая содержится в поле info удаляемого элемента.

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

д) Удалить элемент с указателем q.

$\text{Freenode}(q)$

Окончательно:



2.5. Контрольные вопросы по теории

А. 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=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.

В. 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?
 - 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 двуправленном списке?

- в обоих;
- влево;
- вправо.

2.6. Примеры алгоритмов и приложений (списковые структуры)

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 реализации 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 продолжения"<<endl;
        getch();
    }
}
```

```

else

{
First=First->Next;
delete p;
cout<<"Первый элемент списка удален"<<endl;
    cout<<"Нажмите любую клавишу для продолжения"<<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.

2.7. Варианты заданий

А. 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.

В. 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. Выиграет 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, вытащенных 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.

2.8. Составить отчет по лабораторной работе, и защитить его у преподавателя

ЛЗ №3 /4 часа/. Бинарные деревья (создание и обход)

3.1. 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++.

3.2. 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.

3.1 Содержание отчета по ЛР

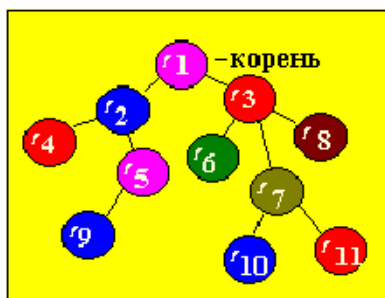
- наименование ЛР и ее цель;
- задание на ЛР согласно варианту;
- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;
- результаты работы программы.

3.4. 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

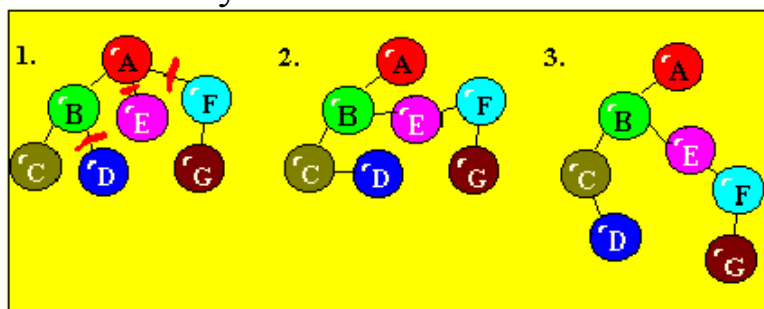
Деревья наиболее удобно представлять в памяти ЭВМ в виде связанных нелинейных списков. Элемент должен содержать *INFO*-поле, где содержится характеристика узла. Следующее поле определяет степень исхода узла и количество полей указателей равно степени исхода. Каждый указатель элемента ориентирует данный 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.

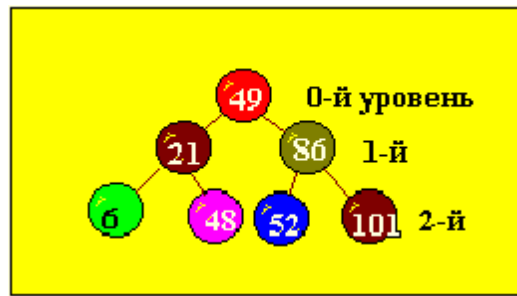


Построение бинарных деревьев. Согласно представлению деревьев в памяти ЭВМ каждый элемент (узел бинарного дерева) будет записью, содержащей четыре поля. Значением этих полей будут соответственно

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

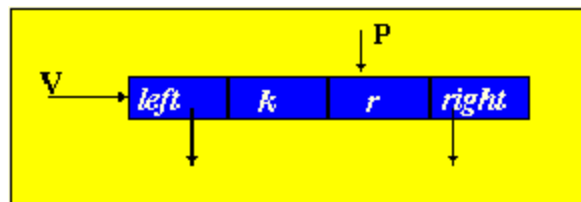
При построении необходимо помнить, что левый сын имеет ключ меньше чем отец (родитель). Значение ключа правого сына больше значения ключа отца.

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.



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

$r = \text{rec}$

$k = \text{key}$

$V = P$

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

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

tree - указатель на корень дерева

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

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

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

while not eof do

$p = \text{tree}$

$q = \text{tree}$

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

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

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

$q = p$

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

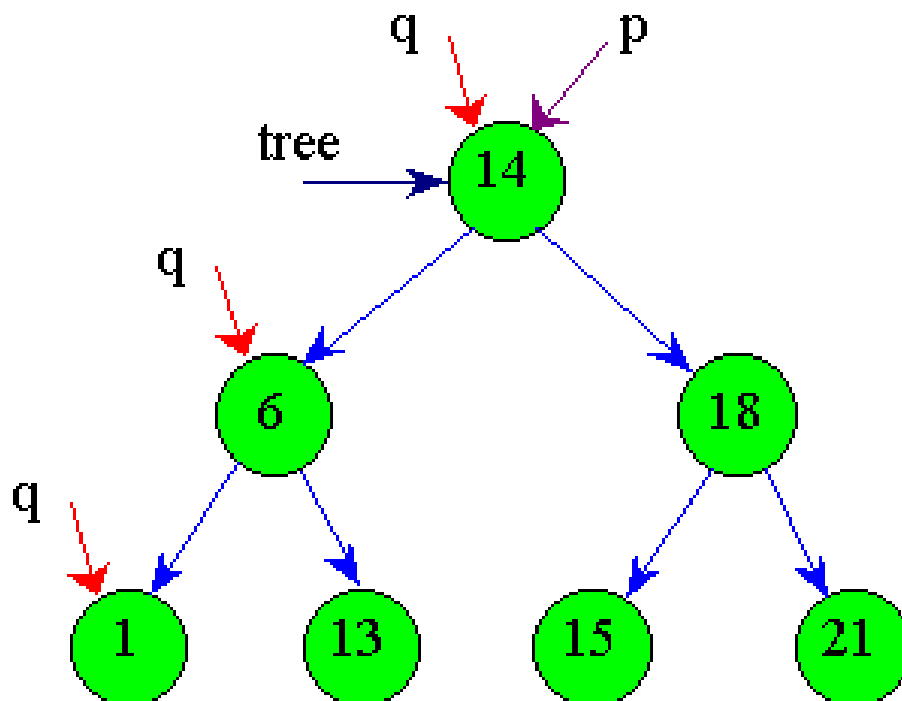
$p = \text{left}(p)$

```

        else
            p = right(p)
        endif
    endwhile
    if p=nil then writeln ('Это корень')
    tree=v
    else
        if key < k(q) then
            left(q) = v
        else
            right(q) = v
        endif
    endif
endwhile
return

```

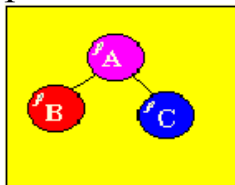
Если по этому алгоритму строить упорядоченное бинарное дерево, то при поступлении ключей в последовательности 14, 18, 6, 21, 1, 13, 15 получим дерево, изображенное на рисунке ниже.



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
  
```

3.5. 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:

- $\text{Element} = \text{Запись}$
 $\text{Left}, \text{Right}$: Указатели
 Rec : Запись;

- $\text{Element} = \text{Запись}$
 Left : Указатель
 Key : Ключ
 Rec : Запись;

- $\text{Element} = \text{Запись}$
 $\text{Left}, \text{Right}$: 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 .

3.6. Примеры алгоритмов и приложений (бинарные деревья)

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

иска, вставки и удаления элементов из бинарного дерева будут рассмотрены в лабораторной работе № 6.

3.7. Варианты заданий

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,

П - 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 непутого 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.

3.8. Составить отчет по лабораторной работе, и защитить его у преподавателя

2. АЛГОРИТМЫ ПОИСКА

ЛЗ №4 /4 часа/. Исследование методов линейного и бинарного поиска

4.1. 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++.

4.2. 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.

4.3. 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.

4.4. 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*).

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", 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

$$\text{found OR } ((L > R) \text{ AND } (A_k : 0 \leq k < L : a_k < x) \text{ AND } (A_k : R < k < N : a_k > x))$$

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 равенство 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

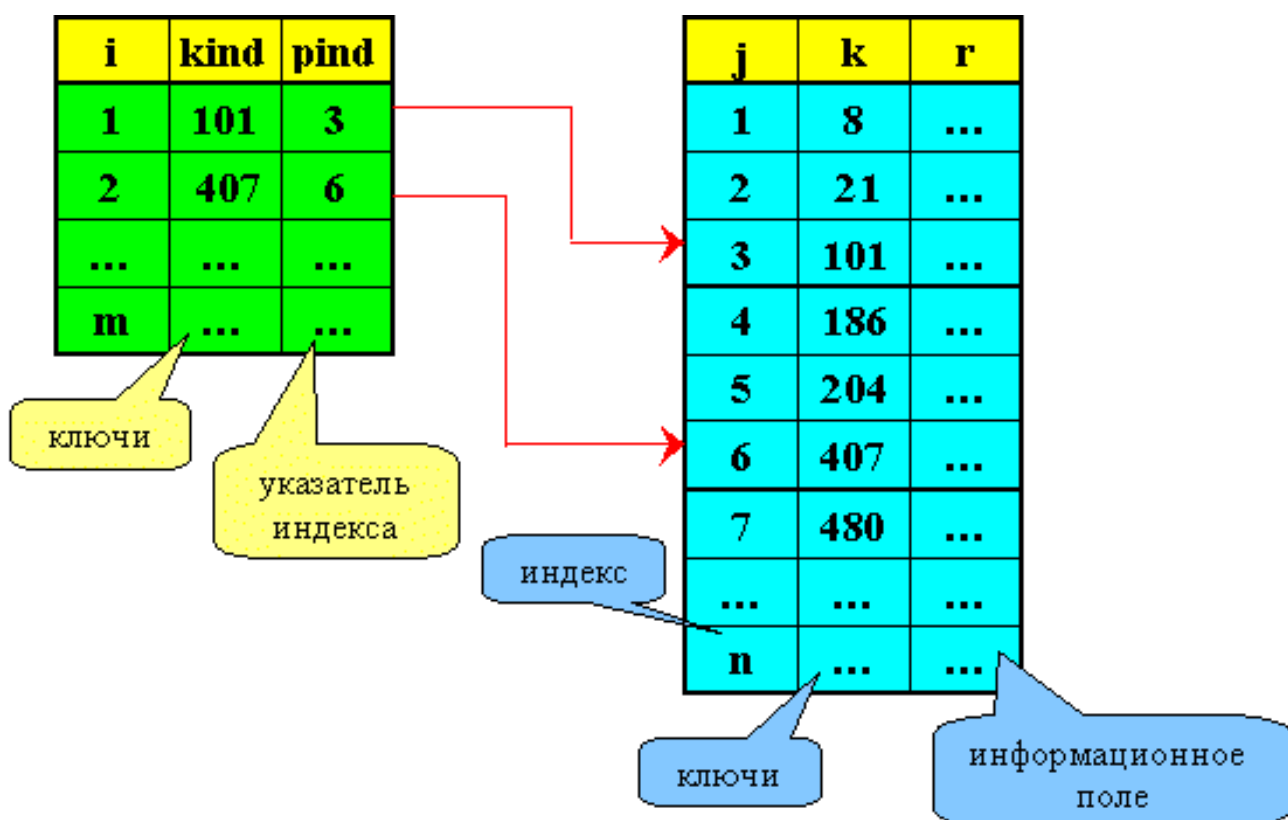
END

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 упо-

рядоченной 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 ($i \leq m$) and ($kind(i) \leq key$) *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.

4.6. Примеры алгоритмов и приложений

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 следующим:

```
/* 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 ис-

пользовать 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 С++ с использованием барьера.

```

/* **** */
/* ПОИСК ПО СОВПАДЕНИЮ КЛЮЧА - ЗАГРАЖДАЮЩИЙ ЭЛЕ-
МЕНТ */

#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 ЗАГРАЖДАЮЩИМ
ЭЛЕМЕНТОМ */
int poisk2(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 порядке\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 УПОРЯДОЧЕННОЙ 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 порядке-----
"<<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.

4.7. Варианты заданий

1. Найти наименьший элемент в упорядоченном массиве A с помощью линейного, бинарного и индексно-последовательного поиска.
2. Найти элементы в упорядоченном массиве A , которые больше 30, с помощью линейного, бинарного и индексно-последовательного поиска.
3. Вывести на экран все числа массива A кратные 3 (3,6,9,...) с помощью линейного, бинарного и индексно-последовательного поиска.
4. Найти все элементы, модуль которых больше 20 и меньше 50 в упорядоченной таблице, с помощью с помощью линейного, бинарного и индексно-последовательного поиска.
5. Вывести на экран все числа упорядоченного массива A кратные 4 (4,8,...) с помощью с помощью линейного, бинарного и индексно-последовательного поиска.

6. Вывести на экран сообщение, каких чисел больше относительно 50 в упорядоченной таблице с помощью линейного, 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, линейный, бинарный и индексно-последовательный поиск.

10. Поиск каждого кратного двум элемента в упорядоченном массиве помощью линейного, бинарного и индексно-последовательного поиска.

11. Найти элемент с заданным ключом и число сравнений для его нахождения в упорядоченном массиве A с помощью линейного, бинарного и индексно-последовательного поиска.

12. Найти элемент с ключом, равным сумме индексов упорядоченного массива A с помощью линейного, бинарного и индексно-последовательного поиска.

4.8. Составить отчет по лабораторной работе и защитить его у преподавателя

ЛЗ №5 /4 часа/. Исследование методов оптимизации поиска

5.1. Цель работы:

- исследовать и изучить методы поиска с перемещением в начало и транспозицией;
- овладеть умениями и навыками написания на С++ программ по исследованию методов поиска с перемещением в начало и транспозицией.

5.2. 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 С++;
- составить отчет по лабораторной работе и защитить его у преподавателя.

5.3. 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.

5.4. 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.

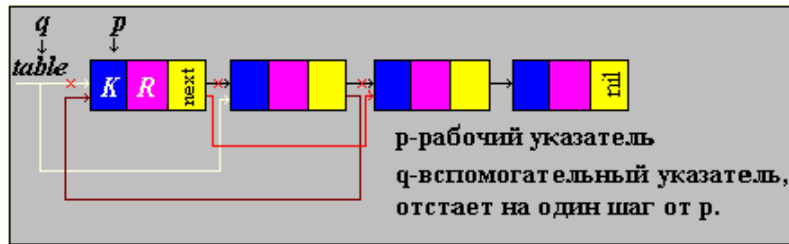
$$P(1) \geq P(2) \geq P(3) \geq \dots \geq P(n)$$

Имеется два основных метода переупорядочивания таблиц поиска:

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 ( $p \neq nil$ ) 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 .

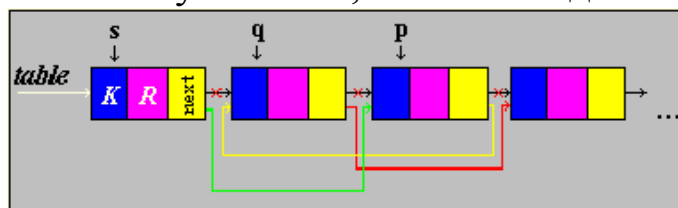


Рисунок иллюстрирует случай, когда найденный элемент третий в списке. Найденный нами третий элемент передвигается на один шаг к голове списка (т.е. становится вторым). Указатель первого элемента перемещается на третий элемент, указатель второго элемента на четвертый, таким образом, третий элемент перемещается 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

```

5.5. 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.

5.6. Примеры алгоритмов и приложений (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;
}
/-----/

```

5.7. Варианты заданий

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 среднее 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

ЛЗ №6 /4 часа/. Поиск по дереву с включением и исключением

6.1. 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 С++ программ поиска по дереву с включением и исключением.

6.2. 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 С++;
- составить отчет по лабораторной работе и защитить его у преподавателя.

6.3. Содержание отчета по ЛР

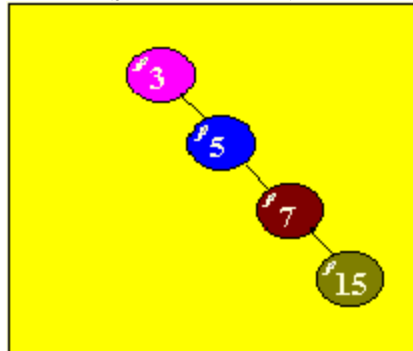
- наименование ЛР и ее цель;
- задание на ЛР согласно варианту;
- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;
- результаты работы программы.

6.4. 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 $\log_2 N$.

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

```

А. 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, определяющей 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
    end
end

```

endif

endif

search = v

return

Вспомогательный указатель *q* здесь отстает на один шаг от рабочего *p*.

В. 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.

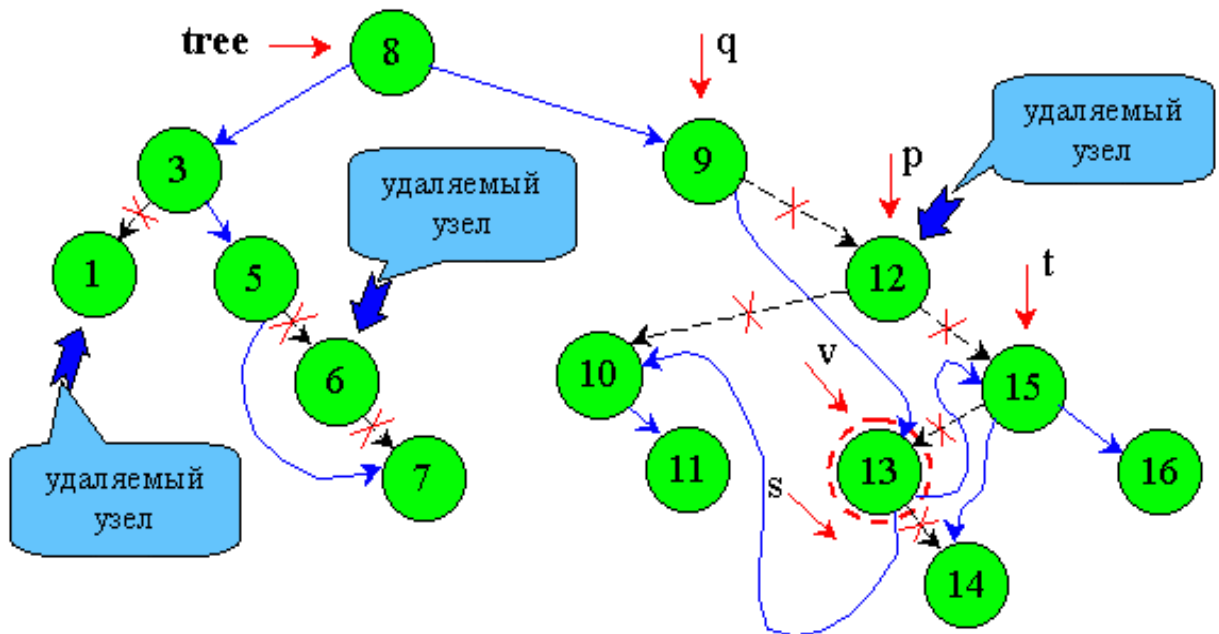
Предшественник - это самый правый элемент левого поддеревы (для достижения этого элемента необходимо перейти в следующий узел по левой ветви, а затем двигаться только по правой ветви этого узла до тех пор, пока очередная ссылка не будет равна *nil*).

Приемник - это самый левый элемент правого поддеревы (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 поддеревы (11). Приемником узла (12) - самый левый узел правого поддеревы (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,  $s$  - опережает)'
             $t = p$ 
             $v = right(p)$ 
             $s = left(v)$ 
        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

```

```

freenode(p)

```

```

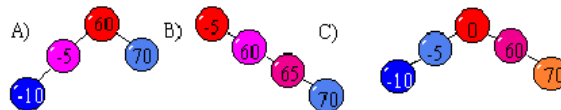
return

```

6.5. Контрольные вопросы по теории (поиск по дереву с включением)

A. Включение элемента в дерево

1. В каком дереве при бинарном поиске нужно перебрать в среднем $N/2$ элементов?

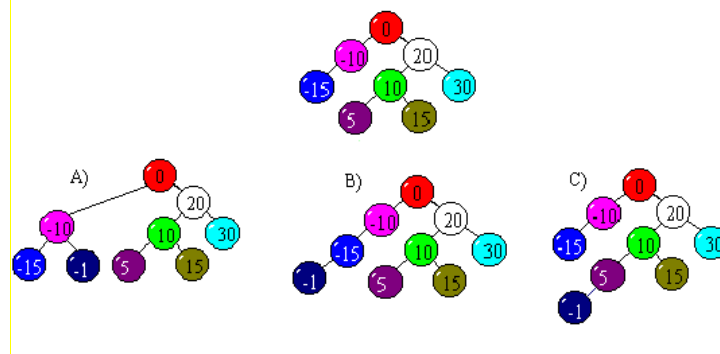


- A;
- B;
- C.

2. Сколько нужно перебрать элементов в сбалансированном дереве при бинарном поиске?

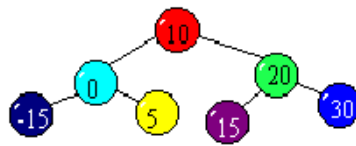
- A - $N/2$;;
- B - $\ln(N)$;;
- C - $\log_2(N)$;;
- D - e^N .

3. Выберите вариант дерева, полученного после вставки узла - 1.



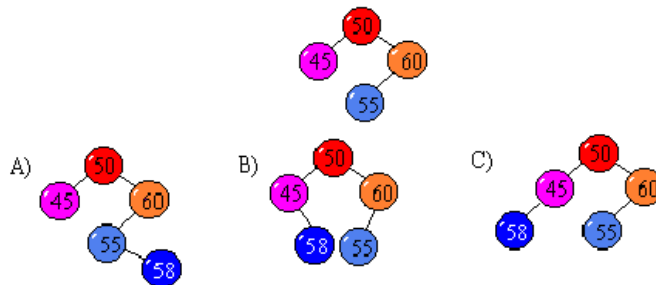
- A;
- B;
- C.

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.

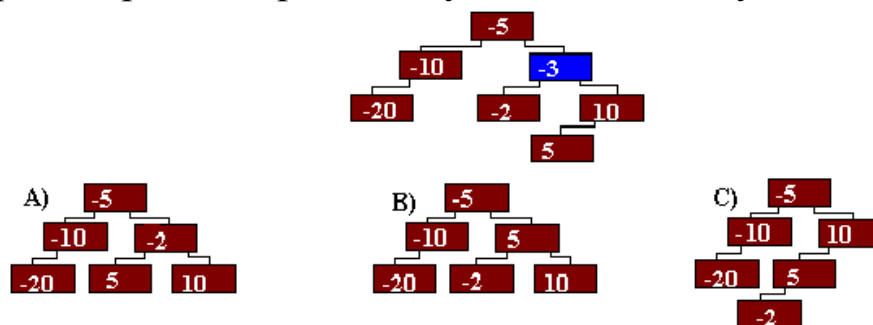
5. Какой вид примет дерево после вставки элемента с ключом 58?



- A;
- B;
- C.

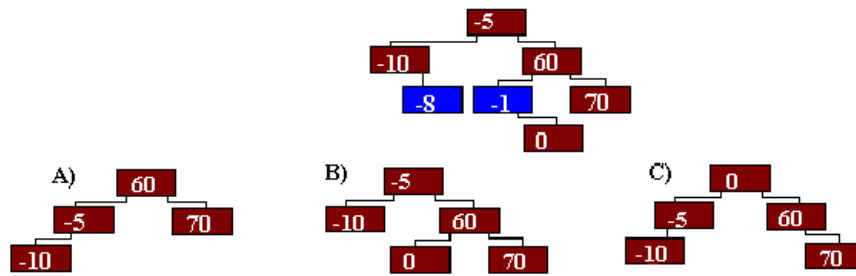
В. Исключение элемента из дерева

1. Выберите вариант дерева, полученного после удаления узла – 3.



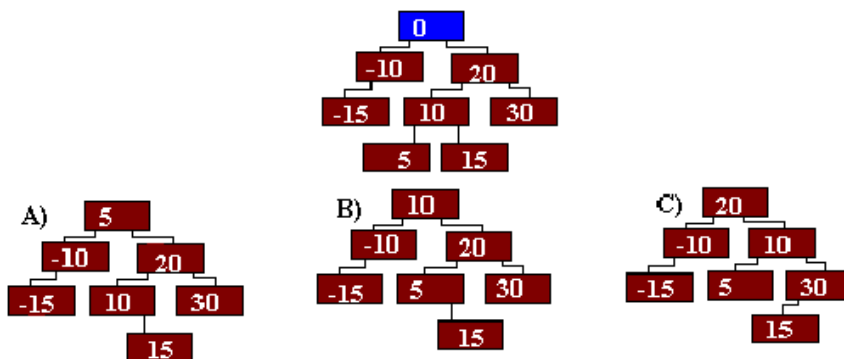
- A;
- B;
- C.

2. Какой вариант дерева получится после удаления элемента – 1, а затем – 8?



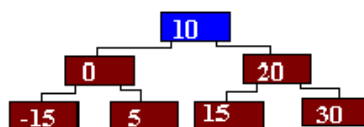
- A;
- B;
- C.

3. Выберите вариант дерева, полученного после удаления узла с индексом 0.



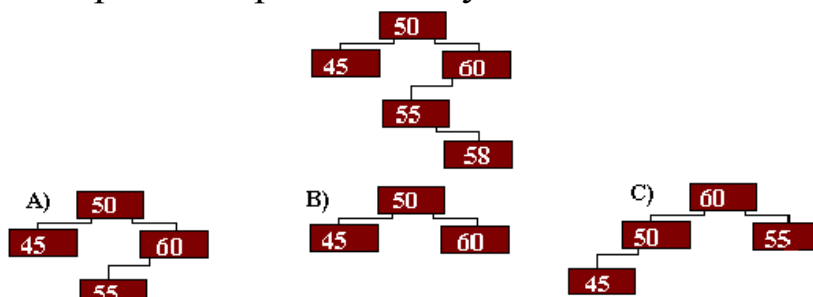
- A;
- B;
- C.

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.

6.6. Примеры алгоритмов и приложений (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->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 вышеописанное 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 элемент 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 (добавляющая) 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 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;
}
}

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;
};
printree(tree);//печать дерева до удаления из него элемента
Q=NULL;
P=tree;
cout<<endl<<"Введите ключ удаляемого элемента: ";
cin>>key;
Poisk_Udalenie(key);

printree(tree);
getch();
}
/*----- */

```

6.7. Варианты заданий

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.

6.8. Составить отчет по лабораторной работе и защитить его у преподавателя

3. АЛГОРИТМЫ СОРТИРОВКИ

ЛЗ №7 /4 часа/. Сортировки методами прямого включения и выбора

7.1. Цель работы:

- исследовать и изучить методы сортировки включением и выбором;
- овладеть умениями и навыками написания на языке программирования С++ программ с использованием сортировки включением и выбором.

7.2. 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 С++;
- составить отчет по лабораторной работе и защитить его у преподавателя.

7.3. 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.

7.4. 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 коротки. 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)

А. 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.

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 отрицательных 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 $O(n^2)$.

В. 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 *B*, 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 следующий:

```

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 обратно 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.

7.5. Контрольные вопросы по теории

А. 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 $M = 0,01 \cdot n \cdot n + 10 \cdot n$?
 - 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.

В. 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.

7.6. Примеры алгоритмов и приложений (по методам сортировки)

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 место 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 делением попо-
лам");
  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]);
}
}
/* ***** */

```

7.7. Варианты заданий

А. 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".

В. Сортировка методом выбора

Создать группу из N студентов.

Ввести их:

- фамилии, имена, годы рождения,
- оценки по предметам:

структуры и алгоритмы данных, высшая математика, физика, про-

граммирование,

- 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

ЛЗ №8 /4 часа/. Сортировки методом прямого обмена и с помощью дерева.

8.1. Цель работы:

- исследовать и изучить методы сортировки с помощью прямого обмена и с помощью дерева;
- овладеть умениями и навыками написания на языке программирования С++ программ сортировки с использованием прямого обмена.

8.2. 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 С++;
- составить отчет по лабораторной работе и защитить его у преподавателя.

8.3. 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.

8.4. 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 основывается 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 элемен-

тов, причем не горизонтальный, а вертикальный, тогда элементы можно интерпретировать 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.

Алгоритм метода прямого обмена (пузырьковой сортировки) в псевдокоде следующий:

```

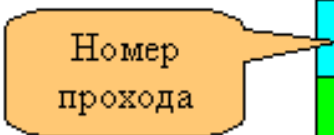
for i = 2 to n
  for j = n to i step -1
    if  $a(j) < a(j - 1)$  then
       $x = a(j - 1)$ 
       $a(j - 1) = a(j)$ 
       $a(j) = x$ 
    endif
  next j
next i
return

```

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	4	1	1	1	1
3	3	4	2	2	2
7	7	3	4	3	3
2	2	7	3	4	4
1	1	2	7	5	5
6	6	5	5	6	6
5	5	6	6	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.

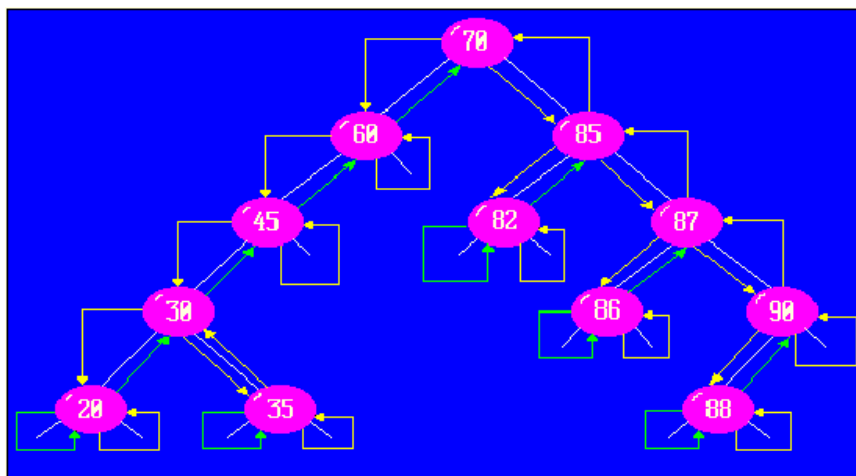
Прежде чем переходить к методам реализации алгоритмов пузырьковой сортировки на языке C++, рассмотрим еще один метод сортировки. Все вышерассмотренные сортировки производились на статических структурах без динамического выделения памяти. Однако в

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 целыми 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

8.5. Контрольные вопросы по теории (сортировка с помощью обмена).

1. Сколько сравнений и перестановок элементов требуется в пузырьковой сортировке?

- $n * \log(n)$;

- $(n*n)/4$;
- $(n*n-n)/2$.

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).

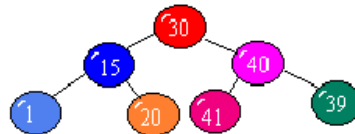
3. Как рассортировать массив быстрее, пользуясь пузырьковым методом?

- 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

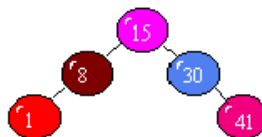


- слева направо получаем последовательность...
- отсортированную по убыванию;
- не отсортированную;
- отсортированную по возрастанию.

6. При обходе дерева слева направо его элемент заносится в массив...

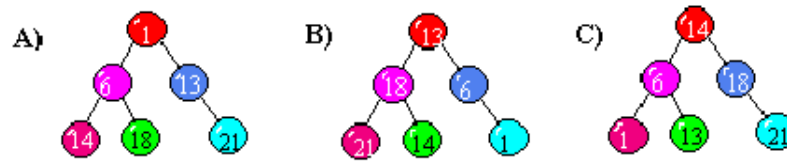
- при втором заходе в элемент;
- при первом заходе в элемент;
- при третьем заходе в элемент.

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.

8.6. Примеры алгоритмов и приложений (сортировка с помощью прямого обмена)

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

    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 );

}

/* ***** */

```

8.7. Варианты заданий

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 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.

8.8. Составить отчет по лабораторной работе и защитить его у преподавателя

ЛЗ №9 /4 часа/. Улучшенные методы сортировки.

9.1. 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++.

9.2. 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.

9.3. 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.

9.4. 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 шагов. 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 да-

лее, 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 еди-

ничным, ведь в самом плохом случае последний проход и делает 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 постановка «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:

$$h_m = 2h_{m-1} + 1, \quad \text{а} \quad \text{количество} \quad \text{шагов}$$

$$t = (\log_2 n) - 1.$$

При такой организации алгоритма его эффективность имеет порядок $O(n^{1.2})$

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

```

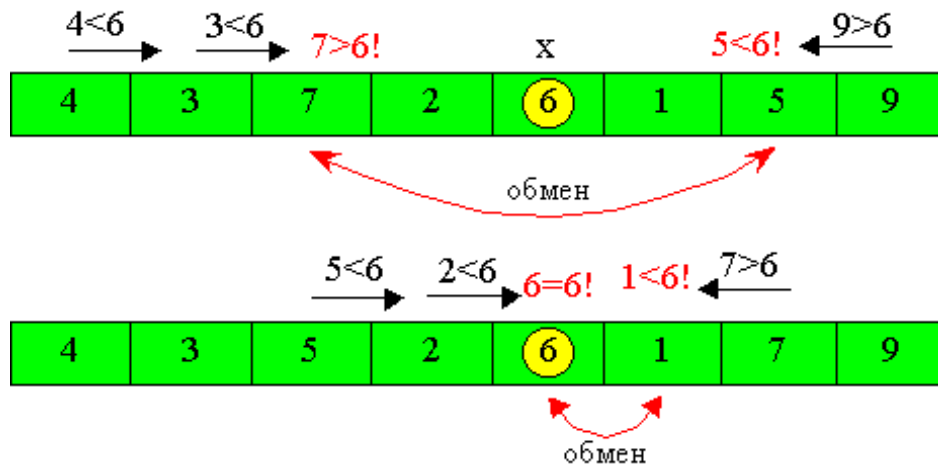
Другим улучшенным методом сортировки является так называемая быстрая сортировка (QuickSort), которая считается сортировкой разделением и носит также название быстрая сортировка Хоара.

Quicksort - метод быстрой сортировки

Быстрая сортировка является усовершенствованием метода, основанного на обмене. В основу данной сортировки положен метод разделения ключей по отношению к выбранному. Он заключается в следующем. Выбираем наугад какой-либо элемент x исходного массива. Будем проматривать 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 . Применив эту процедуру разделения 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 разделения. Если при равновероятном распределении элементов массива в качестве разделения 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 неблагоприятном случае, когда каждый раз для разделения выбирается наибольший об-

рабатываемый 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 (1, n)
return

```

Из всех существующих методов сортировки *QuickSort* самый эффективный. Его эффективность имеет порядок $O(n \log_2 n)$.

9.5. 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.

9.6. Примеры алгоритмов и приложений (улучшенные методы сортировки).

Рассмотрим теперь реализацию функций сортировки методом Шелла и быстрой сортировки на С++ 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);
    }
}
/* **** */

```

9.7. Варианты заданий

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.

9.8. Составить отчет по лабораторной работе и защитить его у преподавателя

ЗАКЛЮЧЕНИЕ

В данных методических указаниях по выполнению лабораторных занятий по дисциплине "Алгоритмы и структуры данных" рассмотрены наиболее распространенные оперативные структуры данных и алгоритмы их обработки, которые традиционно применяются при создании программных систем и комплексов. В силу ограниченности объема этого учебного курса не было уделено внимания таким структурам, как В-деревья и графы, в разделе поиска опущен раздел хеширования. Вообще не рассмотрены структуры данных на внешних носителях и методы работы с ними. В принципе, на базе изученного в данных методических указаниях материала, эти разделы могут быть освоены студентами 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 баз данных и знаний. Другими словами, основные процессы информационных технологий (обработка, обмен, накопление и представление данных) поднялись на следующую ступень, что, естественно, требует новых подходов к организации структур данных в ЭВМ и созданию эффективных алгоритмов и соответствующих им систем программирования. Определяющими факторами к этому являются современные требования к пользовательскому интерфейсу и мультимедийные системы. В представлении данных пользователям появились структуры графических данных и более крупные, интегральные информационные единицы - объекты. Следствием явилось бурное развитие объектно-ориентированных систем программирования, используемых для создания программ, в основе которых лежит обработка объектных структур данных. Обмен объектными структурами в сетях вызван развитием сетевых операционных систем. Обработка данных в многопроцессорных вычислительных системах потребовала создания новых структур данных, основанных на абстрактных представлениях и новых языках программирования.

Таким образом, развитие информационных технологий, их быстрое проникновение во все области жизнедеятельности человека требуют компьютерного отображения информации в виде наглядной визуа-

лизации структур данных. Естественно, каждый новый поступательный шаг информатики будет сопровождаться прогрессом в области практического совершенствования визуального представления структур данных, алгоритмов и компьютерных программ их обработки и представления. Для будущего специалиста по информационным технологиям быстрое осваивание новых методов программирования и структур данных, использование их для грамотного решения возможных практических задач той или иной предметной области имеет первостепенное значение. Эти умения и навыки вырабатываются у специалиста только путем постоянного совершенствования своего профессионального мастерства и настойчивого приобретения драгоценного опыта. Именно на приобретение опыта, выработку умений и навыков эффективной практической реализации структур данных в решение с их помощью конкретных задач нацелены практические занятия по дисциплине "Алгоритмы и структуры данных".

ЛИТЕРАТУРА

1. Адельсон-Вельский Г.М., Ландис Е.М. Один алгоритм организации информации. - Доклады АН СССР, 146, 1962. - С. 263-266.
2. Ахо А. и др. Структуры данных и алгоритмы /А. Ахо, Дж. Хопкрофт, Дж. Ульман: Пер. с англ. - М.: Вильямс, 2000. - 384с.
3. Бертисс А.Т. Структуры данных./Пер. с англ. - М. .Статистика, 1974. - 346 с.
4. Вирт Н. Алгоритмы и структуры данных. - М.: Мир, 1989.
5. Вирт Н. Алгоритмы + структуры данных = программы. /Пер. с англ. - М.: Мир, 1985. - 406с.
6. Вирт Н. Алгоритмы и структуры данных. /Пер. с англ. - М.: Мир, 1989. - 360с.
7. Гудман С. Введение в разработку и анализ алгоритмов. /С. Гудман, С. Хидетниemi С. - М.: Мир, 1981. - 368 с.
8. Гэри М. Вычислительные машины и трудноразрешимые задачи. /М. Гэри., Д. Джонсон - М.: Мир, 1982, - 416с.
9. Евстигнеев В.А. Применение теории графов в программирования. - М.: Наука, 1985. - 352с.
10. Костин А.Е., Шаньгин В.Ф. Организация и обработка структур данных в вычислительных системах. - М.: Высшая школа, 1987.
11. Кнут Д. Искусство программирования для ЭВМ. В 3-х т. - т.1. Основные алгоритмы. - М.: Мир, 1976. - 736 с.
12. Кнут Д. Искусство программирования для ЭВМ. В 3-х т. - т.3. Сортировка и поиск. - М.: Мир, 1978. - 844с.
13. Ленгсам Р.и др. Структуры данных для персональных ЭВМ. - М.: Мир, 1989. – 312 с.
14. Лойко В.И. Алгоритмы и структуры данных: Курс лекций. – Краснодар: КубГАУ, 2006. – 120 с.
15. Лойко В.И. Структуры и алгоритмы обработки данных: учебное пособие. – Краснодар: КубГАУ, 2000. – 261 с.
16. Лойко В.И. Структуры и алгоритмы обработки данных. Методические указания к курсовой работе для студентов всех форм обучения специальности 351400 – «Прикладная информатика (по областям)».- Краснодар: КубГАУ. 2000. - 34 с.
17. Макаровский Б.Н. Информационные системы и структуры данных. - М.: Статистика, 1980. - 199с.
18. Мейер Б., Бодуэн К. Методы программирования. /Б. Мейер, К. Бодуэн. В 2-х т. - М.: Мир, 1982. т.1 - 356с. т.2 - 368с.

19. Нагао М. и др. Структуры и базы данных. / М. Нагао, Т. Катаяма, С. Уэмура. - М.: Мир, 1986. - 197с.
20. Рай ли Д. Абстракция и структуры данных. Вводный курс. - М.: Мир, 1993. -287 с.
21. Разумов О.С. Организация данных в вычислительных системах. - М.: Статистика, 1978. – 184 с.
22. Седжвик Р. Фундаментальные алгоритмы на С++. Анализ/ Структуры данных/ Сортировка/ Поиск. - СПб.: ООО «ДиаСофтЮП», 2002. - 688с.
23. Сибуя М., Ямамото Т. Алгоритмы обработки данных. /М. Сибуя, Т. Ямамото. - М.: Мир, 1986. - 218с.
24. Трамбле Ж., Соренсон П. Введение в структуры данных. /Пер. с англ. - М.: Машиностроение, 1984. - 784с.
25. Топп У. Структуры данных в Си++. /У. Топп, У. Форд. - М.: БИНОМ, 2000. - 816с.
26. Холл П. Вычислительные структуры. Введение в нечисленное программирование. /Пер. с англ. - М.: Мир,1978. – 214 с.
27. Хусаинов Б.С. Структуры и алгоритмы обработки данных, Примеры на языке Си (CD): Учебное пособие. – М.: Финансы и статистика, 2004. - 464 с.

ПРИЛОЖЕНИЯ

Приложение 1. Календарно-тематический план изучения дисциплины

"Алгоритмы и структуры данных" 2 курс, семестр 3

№ п/п	№	Разделы и темы занятий (лекций и практических занятий)	Всего часов по формам обучения			
			Очная		Заочная	
			Л к	Л З	Л к	Л З
		1. Структуры данных				
1		Цели и задачи курса	2			
2		Классификация структур данных	2			
3		Полустатические структуры данных - стеки, очереди, деки	2			
	1	Полустатические структуры данных		4		
4		Линейные динамические структуры	2		2	
5		Реализация стеков с помощью односвязных списков	2			
6		Односвязный список как самостоятельная структура данных	2		2	
	2	Списковые структуры данных (кольцевые списки)		4		4
7		Нелинейные динамические структуры	2			
	3	Бинарные деревья (основные процедуры)		4		
8	2	Реализация стеков с помощью односвязных списков	2			
		2. Алгоритмы поиска				
9		Алгоритм создания дерева бинарного поиска	2			
10		Классификация основных методов поиска	2		2	
	4	Исследование методов линейного и бинарного поиска		4		4
11		Методы оптимального поиска	2			
	5	Исследование методов оптимизации поиска		4		
12		Дерево оптимального поиска	2			
13		Поиск по бинарному дереву	2			
	6	Поиск по бинарному дереву с включением		4		

14	Поиск по бинарному дереву с удалением	2			
	3. Алгоритмы сортировки				
15	Внутренняя и внешняя сортировки	2		2	
16	2 Сортировка методом прямого выбора	2		2	
	7 Сортировка методом прямого включения и выбора		4		
17	Улучшенные методы сортировка	2			
	8 Сортировка с помощью обмена		4		2
18	Сортировка с помощью дерева	2			
	9 Сортировка с помощью дерева		4		
	Итого:	36	36	10	10

П1.2. График выполнения курсового проекта (работы)

Номер недели	Содержание выполняемой работы	Вид отчётности
4	Обзор методов и алгоритмов реализации	Обзорный материал
8	Программа исследования метода или структуры	Работающая программа
12	Предварительные результаты исследования метода или структуры	Предварительный вариант пояснительной записки
16	Окончательный вариант пояснительной записки или программы	Защита курсовой работы

Приложение 2. Программа самостоятельной работы студентов по дисциплине

"Алгоритмы и структуры данных"

под контролем преподавателя для студентов ФПИ ВГОУ ВПО КубГАУ

П2.1. Содержание самостоятельной работы

Самостоятельная работа студента (СРС) по дисциплине "Алгоритмы и структуры данных" осуществляется путем освоения будущими информатиками тем вопросов дисциплины, не включенных в лекционный курс и практические занятия. При выполнении заданий СРС обучаемый обязан самостоятельно изучить эти вопросы.

Цель СРС – самостоятельное изучение студентами теоретических и технологических разделов дисциплины, дающих углубленное пред-

ставление о механизмах эффективного применения алгоритмы и структуры данных при решении практических задач. Задачи СРС – освоение студентом под руководством преподавателя перечисленных ниже вопросов-заданий. В результате этой работы студенты должны:

знать

- методы ветвей и границ, открытой адресации и цепочек, разрешения коллизий, средства реализации рекурсивных алгоритмов;
- суть задачи коммивояжера, алгоритмы составления расписаний и упаковки, средства ;

уметь

- применять хеш-функции, метод ветвей и границ, оценивать эффективность рекурсивных и итеративных алгоритмов;

обладать навыками

- разработки алгоритмов и структур данных, программ выполнения расчетов, подготовки отчетной документации;
- поиска необходимой информации в отраслевых, национальных и мировых базах знаний и данных.

П2.2. Виды и объём самостоятельной работы студента

Вид самостоятельной работы	Все-го часов	Форма контроля
1. Самостоятельное изучение отдельных тем (вопросов)	14 8	Контроль СРС при изучении отдельных тем. Контроль выполнения и защита курсовых работ
2. Подготовка рефератов по индивидуальным заданиям		
3. Подготовка докладов на семинары и конференции		
4. Проведение патентного поиска по тематике курсового проектирования, научной студенческой работы и пр.		
5. Выполнение студенческой научной работы (по тематике изучаемой дисциплины)		
6. Другие виды самостоятельной работы		
Общий объём:	22	

П2.3. Темы (вопросы), выносимые на СРС и рекомендуемая литература

Тема (вопрос)	Основная литература	Дополнительная литература
1. Хеш-функция. Коллизия. Разрешение коллизий методом от-крытой адресации и методом цепочек. Выбор хеш-функций	Лойко В.И. Струк-туры и алгоритмы обработки данных. Учебное пособие для ву-зов – Краснодар:: КубГАУ, 2000.	Вирт Н. Алго-ритмы и струк
2. Средства реализации рекурсивных алгоритмов. Эффективность рекурсивных и итеративных алгоритмов		туры дан-ных -. М. Мир, 2001.
3. Алгоритмы составления расписаний и упаковки. Метод ветвей и границ. Задача коммивояжера		

Приложение 3. Вопросы для подготовки к экзамену по дисциплине
"Алгоритмы и структуры данных"
для студентов ФПИ ФГОУ ВПО КубГАУ

1. Понятие типов и структур данных. Оперативные и внешние структуры.
2. Стандартные и пользовательские типы данных.
3. Определение и представление структур данных.
4. Классификация структур данных. Векторы и массивы как статистические структуры.
5. Записи и таблицы как статические структуры.
6. Понятие списковой структуры. Стек как полу статическая структура. Операция над стеками
7. Очередь как полу статическая структура. Операции над очередью.
8. Кольцевая полу статическая очередь. Операции над кольцевой очередью. Деки, операции над ними.
9. Понятие динамических структур данных. Организация односвязных и двусвязных списков. Простейшие операции над односвязными списками.
10. Реализация стеков с помощью списков.
- 11.Смысл и организация операций создания и удаления элемента динамической структуры. Понятие свободного списка и пула свободных элементов. Утилизация освободившихся элементов.
- 12.Очередь и операции над ней при реализации связными списками
- 13.Операции вставки и извлечения элементов из списка. Сравнение этих операций с аналогичными в массивах. Недостаток связного списка по сравнению с массивом.
- 14.Примеры типичных операций над связными списками.
- 15.Элементы заголовков в списках; нелинейные связные структуры.
- 16.Понятие рекурсивных структур данных. Деревья, их признаки и представления.
- 17.Алгоритм сведения m -арного дерева к бинарному; основные операции над деревьями; виды обхода.
- 18.Понятие поиска, ключей; назначение и структуры алгоритмов поиска.
- 19.Последовательный поиск и его эффективность.
- 20.Индексно-последовательный поиск.

- 21.Переупорядочивание таблицы с учетом вероятности поиска элемента; переупорядочивание путем перестановки в начало списка.
- 22.Метод транспозиции для переупорядочивания таблицы поиска.
- 23.Бинарный поиск
- 24.Алгоритм создания упорядоченного бинарного дерева.
- 25.Поиск по бинарному дереву и поиск с включением.
- 26.Поиск по бинарному дереву с удалением.
- 27.Эффективность поиска по бинарному дереву; алгоритмы прохождения бинарных деревьев.
- 28.Понятие сортировки, ее эффективность; классификация методов сортировки.
- 29.Сортировка методом прямого выбора.
30. Сортировка методом прямого включения.
- 31.Сортировка методом прямого обмена.
- 32.Быстрая сортировка.
- 33.Сортировка Шелла.
- 34.Сортировка с помощью дерева.
- 35.Сравнительный анализ эффективности методов сортировки.
- 36.Не рекурсивный алгоритм симметричного обхода бинарного дерева.

Приложение 4. Перечень учебно-методических материалов, используемых по дисциплине

"Алгоритмы и структуры данных" для студентов ФПИ ФГОУ ВПО КубГАУ

Пособия и методические указания кафедры КТС:

1. Лойко В.И. Алгоритмы и структуры данных: Курс лекций. – Краснодар: КубГАУ, 2006. – 120 с.
2. Лойко В.И. Алгоритмы и структуры данных: Методические указания по подготовке курсовых работ для студентов специальностей 071900 - "Информационные системы в технике и технологиях" и 010502.65 - Прикладная информатика (по областям). /В.И. Лойко, В.Н. Лаптев. – Краснодар: ФГОУ ВПО КубГАУ, 2007. – 36 с.
3. Лойко В.И. Структуры и алгоритмы обработки данных: Учебное пособие. – Краснодар: КубГАУ, 2000. – 261 с.
4. Лойко В.И. Структуры и алгоритмы обработки данных. Методические указания к курсовой работе для студентов всех форм обу-

чения специальности 351400 – «Прикладная информатика (по областям)».- Краснодар: КубГАУ. 2000. - 34 с.

Методические материалы:

1. Слайды мультимедийных лекций по дисциплине "Алгоритмы и структуры данных".
2. Автоматизированная обучающая система (АОС) по дисциплине.
3. Учебники и учебные пособия других авторов.
4. Конспекты лекций, рабочие тетради с ЛЗ, алгоритмы и листинги программ на C++ по дисциплине.

Приложение 5. Программное обеспечение, используемое при изучении дисциплины

"Алгоритмы и структуры данных" для студентов ФПИ ВГОУ ВПО КубГАУ

1. Универсальный интегрированный пакет для офиса Microsoft Office 2003, Professional Edition, содержащий программы:
 - 1.1. MS Word 2003 – текстовый процессор;
 - 1.2. MS Excel 2003 – табличный процессор (электронная таблица);
 - 1.3. MS Access 2003 – система управления базой данных (СУБД);
 - 1.4. MS PowerPoint 2003 – пакет демонстрационной графики (презентаций);
 - 1.5. AdobePhotoshop – графический редактор;
 - 1.6. Internet Explorer 8 - обозреватель Internet
 - 1.7. CorelDRAW – графический редактор;
2. Операционная система:
 - 2.1. Windows XP.
3. Файл-менеджеры:
 - 3.1. Windows Commander 5.0;
 - 3.2. FAR.
4. Интегральная среда программирования C++.
5. Автоматизированная обучающая система (АОС) по дисциплине.

Учебное издание

**Лойко Валерий Иванович
Лаптев Владимир Николаевич
Лаптев Сергей Владимирович
Параскевов Александр Владимирович
Ткаченко Василий Владимирович**

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

**Методические указания к лабораторным занятиям для студентов
специальностей**

**230201.65 - "Информационные системы и технологии" и
080801.65 – "Прикладная информатика (по областям)"**

Литературный редактор: Авторская правка
Оригинал макет: Лаптев С.В.

Заказ № _____ Тираж 100 экз.
Отпечатано в типографии Кубанского государственного аграрного уни-
верситета
350044, г. Краснодар, ул. Калинина, 13