МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Маркетинг и менеджмент»

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Методические рекомендации к самостоятельной работе для студентов специальности 1-28 01 02 «Электронный маркетинг» заочной формы обучения

Часть 2

Могилев 2021

УДК 004.43
ББК 32.973-018
О75

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Маркетинг и менеджмент» «3» февраля 2021 г., протокол № 10

Составитель канд. физ.-мат. наук, доц. С. Н. Батан

Рецензент канд. техн. наук, доц. В. М. Ковальчук

В методических рекомендациях представлены описание аудиторной контрольной работы, критерии её оценки, изложена последовательность изучения теоретических вопросов, приведены основные термины и понятия, а также примерный перечень вопросов для написания аудиторной контрольной работы.

Учебно-методическое издание

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Часть 2

Ответственный за выпуск А. В. Александров

Корректор А. А. Подошевко

Компьютерная верстка Н. П. Полевничая

Подписано в печать . Формат 60×84/16. Бумага офсетная. Гарнитура Таймс.
Печать трафаретная. Усл. печ. л. . Уч.-изд. л. . Тираж 16 экз. Заказ №

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

© Белорусско-Российский
университет, 2021

Содержание

1 Описание аудиторной контрольной работы и критерии её оценки………………………………………………………………. 4
2 Содержание учебного материала…………………………………….. 4
2.1 Тема 1. Динамические структуры данных………………………… 4
2.2 Тема 2. Классы в С++. Объектно-ориентированное программирование……………………………………………………….. 28
3 Примерный перечень вопросов для написания аудиторной контрольной работы…………………………………………… 45
Список литературы…………………………………………………… 46

1 Описание аудиторной контрольной работы и критерии её оценки

Аудиторная контрольная работа (АКР) направлена на проверку подготовленности студента по теоретической части дисциплины.

АКР включает два теоретических вопроса из тринадцати тем.

Для получения зачета по АКР необходимо дать исчерпывающий ответ на оба теоретических вопроса.

АКР оценивается исходя из 10 (десяти) баллов. Критерии оценки представлены в таблице 1. АКР считается зачтенной, если сумма полученных баллов составляет не менее 5 (пяти) баллов.

Таблица 1 – Критерии оценки АКР

Задание	Максимальный балл
Вопрос 1	5
Вопрос 2	5
Итого по заданиям	**10**

2 Содержание учебного материала

2.1 Тема 1. Динамические структуры данных

Списки.

Изучить основные теоретические положения темы в указанной последовательности.

Понятие «***структура данных***» является расширением понятия «***тип данных***».

Структуры данных можно разделить на ***простые*** и ***сложные***. ***Простые структуры*** состоят из единственного элемента, несущего какую-то смысловую нагрузку. Этот элемент можно представить как совокупность байтов или бит, но каждый отдельно взятый байт или бит нельзя рассматривать в отрыве от других. В языке С++ простым структурам соответствует скалярный тип (целые, вещественные, символьные, булевский).

Сложные структуры **–** это набор некоторым образом сгруппированных данных. В С++ сложным структурам соответствует понятие структурированного типа (массив, структура).

Типы структур:

– *однородные,* состоящие из элементов одного типа (массивы);

– *неоднородные* – элементы которых могут иметь различный тип, например, структуры;

– *последовательные* или *несвязанные* структуры данных. В таких структурах все элементы, входящие в ее состав, расположены в памяти друг за другом. Поэтому возможен прямой метод доступа к любому элементу;

– *связанные*, у которых каждый элемент содержит информацию о других элементах этой же структуры. Прямой доступ к элементам связанных структур затруднен, а зачастую и невозможен.

Операции над структурами:

– доступ к любому элементу;
– поиск индекса элемента по его значению;
– добавление нового элемента;
– изменение значений элемента с сохранением упорядоченности;
– удаление элемента;
– вставка нового элемента в произвольное место.

Для каждой из структур данных можно выделить как *удобные*, так и *неудобные* операции (т. е. такие операции, выполнение которых требует существенных затрат времени).

Упражнение. Какие операции являются удобными и неудобными для массивов?

Абстрактные типы данных – это такие структуры, для которых нет соответствующих типов в языке программирования, но которые можно реализовать средствами языка.

Абстрактный тип данных «список»

Определение списков.

Под списками (точнее, под ***однонаправленными линейными списками***) будем понимать связанную структуру данных, каждый элемент которой содержит информацию о последующем элементе списка – т. н. ***ссылку*** на последующий элемент. Последний элемент списка содержит ***пустую*** ссылку. Именно по значению пустой ссылки можно определить, что текущий элемент списка – последний. Кроме этого, список должен содержать специальные данные – ***ссылку на первый элемент***. Структура линейного однонаправленного списка представлена на рисунке 1.

Так как каждый элемент списка содержит, по меньшей мере, два поля, то для его представления удобно использовать запись, содержащую поле **Info** произвольного типа (информационная часть) и поле **Next**, в котором хранится ссылка на последующий элемент. Тип поля **Next** зависит от физической организации списка.

Кроме того, в состав списка должна входить переменная **First**, которая хранит информацию о первом элементе списка.

Для списка удобными являются следующие операции:

– вставка нового элемента в произвольное место списка (особенно – в начало списка);
– удаление элемента из списка.

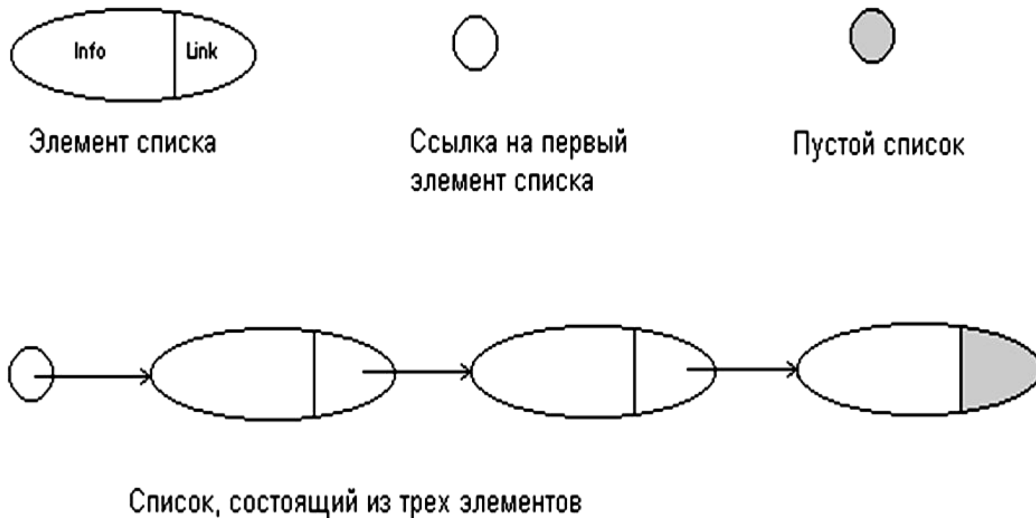

Рисунок 1 – Структура линейного однонаправленного списка

Обе эти операции требуют перестройки одной-двух ссылок.

Для доступа к произвольному элементу списка необходимо пройти всю цепочку предшествующих элементов.

Поиск нужного элемента в отсортированном списке с использованием дихотомии невозможен.

Добавление нового элемента в список (схема) представлено на рисунке 2.

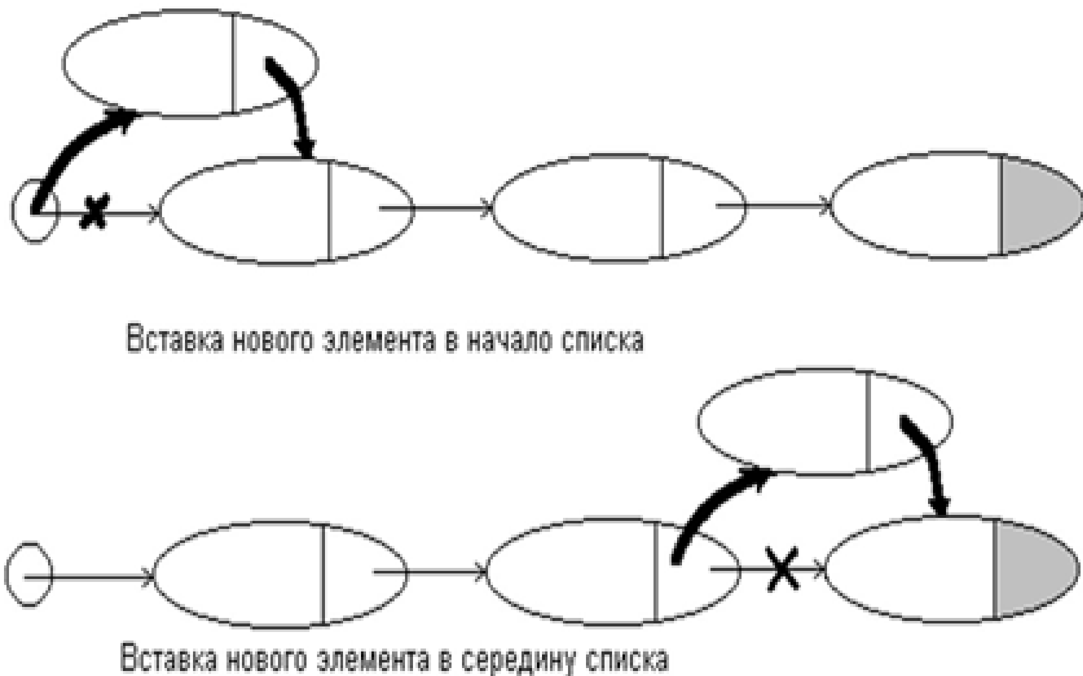

Рисунок 2 – Добавление нового элемента в список (схема)

При удалении элемента из списка в статической памяти он не удаляется из памяти, просто доступ к нему невозможен из-за того, что на удаленный элемент

нет ссылок. Для физического удаления элемента списка из памяти следует выполнить т. н. процедуру ***сборки мусора***, которая предполагает освобождение неиспользуемых областей памяти. Алгоритмы сборки мусора сильно зависят от физической организации списка.

Удаление элемента из списка (схема) представлено на рисунке 3.

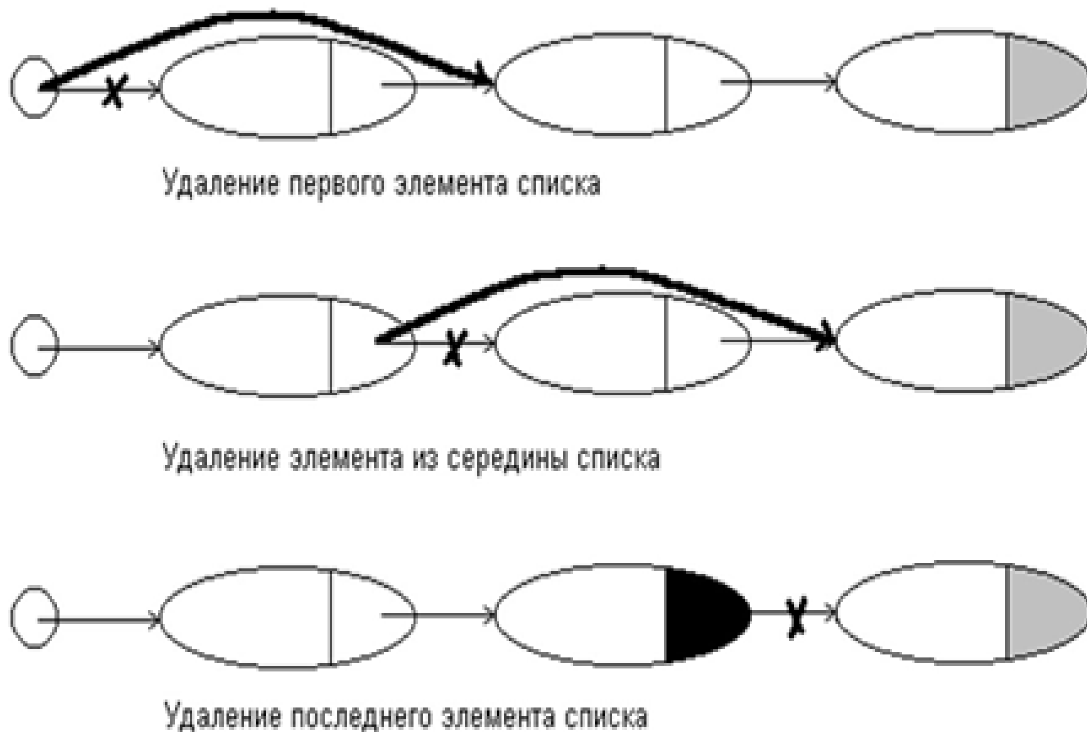

Рисунок 3 – Удаление элемента из списка (схема)

Реализация списков.

Наиболее часто используются следующие способы организации списков:

– с использованием динамической памяти;
– с использованием массивов.

Первый способ предполагает, что для каждого нового элемента списка выделяется участок динамической памяти.

После удаления элемента из списка можно освободить занятую этим элементом память, освобождая себя от дополнительных хлопот по сборке мусора.

Описание списка (реализация через динамическую память)

```
struct ListItem {
int Info;
ListItem* Next;
};
ListItem* First;
```

Теперь для задания пустого списка достаточно написать следующее:

```
First = NULL;
```

Обработка списка.

1 Добавление нового элемента в список:

а) ***в начало списка***:

```
ListItem* P = new ListItem; //заполнение поля P->Info
P->Next = First;
First = P;
```

Алгоритм.

1 Создать новый элемент типа список.

2 Инициализировать его информационное поле.

3 Его ссылке присвоить значение из указателя на начало списка.

4 Изменить указатель на начало списка, занести в него адрес вставленного элемента;

б) ***после элемента, адрес которого находится в указателе Q***:

```
ListItem* P = new ListItem;
//заполнение поля P->Info
P->Next = Q->Next;
Q->Next = P;
```

Алгоритм.

1 Создать новый элемент типа список.

2 Инициализировать его информационное поле.

3 Его ссылке присвоить значение из элемента списка, адрес которого находится в указателе *Q*.

4 Изменить ссылку в элементе, адрес которого находится в указателе *Q*, занести туда адрес вставленного элемента.

2 Удаление элемента из списка:

а) ***из начала списка***:

```
if (First == NULL)
 throw ”List is already empty!”;
ListItem* P = First;
First = First->Next;
delete P;
```

б) ***после элемента, адрес которого находится в указателе Q***:

```
ListItem* P = Q->Next;
if (P == NULL)
```

```
 throw ”Nothing to delete!”;
Q->Next = P->Next;
delete P;
```

При удалении всего списка необходимо вначале уничтожить каждый элемент списка, а уже затем очищать значение указателя **`First`**. Заметим, что перед удалением элемента списка ссылка на него должна быть скопирована!

3 Просмотр всех элементов списка:

```
ListItem* P = First; while (P != NULL)
{
 // выполнить действия
 // над элементами P->Info  P = P->Next;
}
```

Упражнение. Реализовать следующие функции обработки динамического списка:

– добавления нового элемента в начало списка;
– добавления нового элемента после элемента, адрес которого известен;
– поиска заданного элемента списка по его значению;
– добавления нового элемента в список после заданного элемента;
– распечатки значений элементов списка;
– удаления элемента из начала списка;
– удаления элемента после элемента, адрес которого известен;
– удаления заданного элемента списка по его значению;
– уничтожения списка.

Второй способ организации списка предполагает, что в памяти выделяется двумерный массив из N строк (N – максимальное число элементов, которые можно поместить в массив) и двух столбцов.

Пример организации списка представлено на рисунке 4.

Каждая строка соответствует одному элементу списка, причем в первом столбце записывается содержимое элемента, а во втором – индекс строки со следующим элементом. Значение 0 соответствует последнему элементу списка. Кроме того, переменная целого типа хранит номер строки с начальным элементом списка.

Упражнение. Нарисуйте вид списка после вставки в список элемента «Иванов».

Реализация списка в виде массива

```
struct ListItem {
char Info[50];
int Next;
};
```

```
ListItem* List = new ListItem [N];

int First = -1, FirstFree = 0; //со сборкой мусора
for (int i=0; i<N-1; i++)
List[i].Next = i+1;
```

Организация списка в виде масси (пример)

Петров	7
Алексеев	8
Сидоров	-1
Борисов	2

Индекс начального элемента - 5

Сборка мусора при организации списка в виде массива

	4
	3
Петров	7
	0
	6
Алексеев	8
	9
Сидоров	-1
Борисов	2
	10
	11
	-1

Индекс начального элемента - 5
Индекс начального свободного элемента - 1

Рисунок 4 – Организации списка

Другие виды списков.

Двунаправленные или ***двухсвязные*** списки. Элементы такого списка содержат ссылку не только на последующий, но и на предыдущий элемент. Сам список содержит, кроме того, ссылки на первый и последний элементы. Такие списки используются, когда надо часто просматривать элементы списка в разных направлениях, например, при скроллинге различных таблиц.

Кроме того, двунаправленные списки позволяют восстановить значение одной ссылки при ее случайном разрушении и поэтому считаются более надежными структурами данных.

Кольцевые или ***циклические*** списки.

В кольцевых списках последний элемент содержит ссылку на первый. В них теряется смысл понятий «первого» и «последнего» элементов списка, вместо этого уместнее говорить о «текущем» элементе. Для таких списков удобной является операция перемещения текущего элемента вперед (а в случае двунаправленных списков – и назад).

Упражнения.

1 Реализовать обработку двунаправленных линейных и кольцевых списков, представленных в динамической памяти.

2 Реализовать обработку всех видов списков, представленных в виде массива записей.

Стеки. Абстрактный тип данных «стек».

Стек – это специальный тип списка, в котором все вставки и удаления выполняются только на одном конце, называемом вершиной (TOP или HEAD).

Стеком называется совокупность однотипных элементов, над которой определены две основных операции:

1) занесение, или заталкивание, в стек (**push**);

2) извлечение из стека (**pop**). При этом извлекается тот элемент, который был занесен в стек последним. В соответствии с правилами данной операции стек еще называют структурой типа LIFO («last in – first out»).

В классическом стеке недопустимы никакие другие операции, кроме **push** и **pop**. В частности, доступ к хранящимся в стеке элементам возможен только после их извлечения из стека.

Число хранящихся в стеке элементов также нельзя определить. Вместо этого можно обрабатывать нештатную ситуацию, возникающую при выполнении операции **pop** – пустоту стека.

Физическая реализация стека может быть организована по-разному.

Первый способ организации основан на организации стека в виде списка, тогда операции **push** и **pop** сводятся к вставке элемента в начало списка и удалению первого элемента.

Однако этот способ, несмотря на кажущуюся простоту, обладает существенными недостатками:

– требуется дополнительная память на хранение ссылок, хотя их значение изменяется редко;

– операции вставки и удаления являются довольно трудоемкими.

Второй способ организации стека предполагает использование массива, в котором будут храниться элементы стека. Дополнительно (как и в случае реализации списка в виде массива) используется целочисленная переменная – *вершина* стека. Значение, хранящееся в ней, соответствует либо индексу последнего занесенного в стек элемента, либо индексу первого свободного элемента.

В любом случае при занесении в стек значение вершины увеличивается, а при извлечении из стека – уменьшается (естественно, после проверки на пустоту стека). В дальнейшем будем использовать в качестве значения вершины индекс первого свободного места.

Этот способ также не лишен недостатков. Варианты реализации стека на массивах представлены на рисунке 5.

Во-первых, необходимо ограничить количество хранящихся в стеке элементов, а при попытке вставки проверять еще одну нештатную ситуацию – переполнение стека.

Во-вторых, организация массива в статической памяти требует, чтобы это количество было определено в момент написания программы, а не в момент ее выполнения.

Второй недостаток в принципе может быть устранен за счет использования динамической памяти.

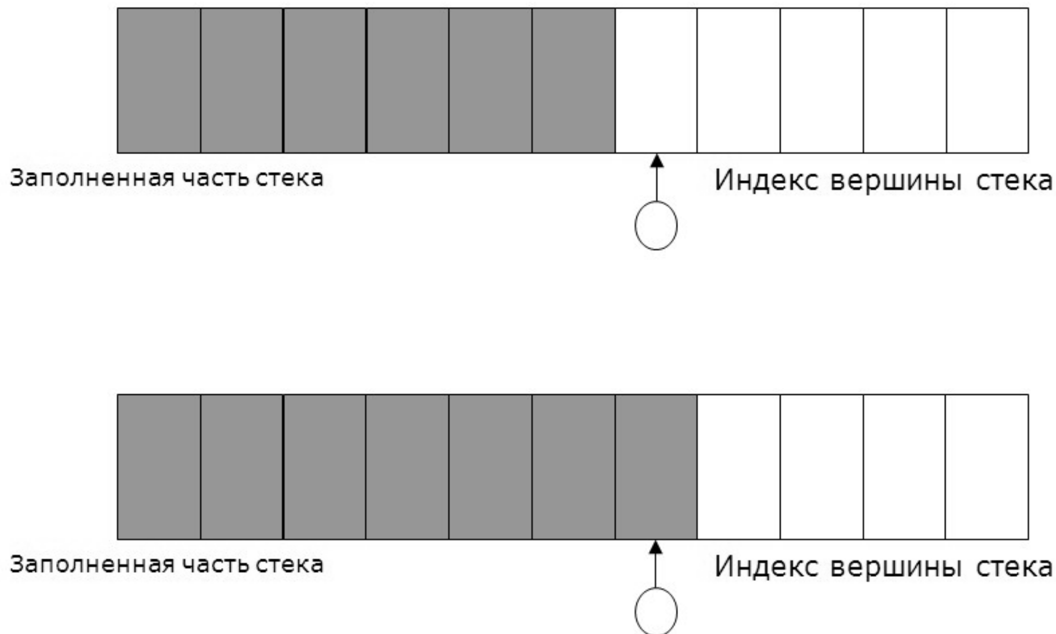

Рисунок 5 – Варианты реализации стека на массивах

Программная реализация стека

```
int *stack, Top  = 0;
…
stack = new int[N];
// push – заносится значение k
if (Top > N-1)
throw "Stack is full!";
stack[Top++] = k;
 // pop – извлекается значение в k
if (Top == 0)
throw "Stack is empty!";
k = stack[--Top];
```

Рассмотрим в качестве примера классическую задачу, использующую принципы работы со стеками (хотя термин «стек» в условии не присутствует и даже не подразумевается!).

Пример – Входная строка содержит алгебраическое выражение, включающее круглые, квадратные и фигурные скобки. Будем считать, что скобки расставлены правильно, если:

– каждой открывающей скобке конкретного типа соответствует следующая за ней закрывающая скобка того же типа;

– нет закрывающих скобок любого типа без предшествующих им открывающих скобок того же типа;

– не допускается рассогласование типов скобок, когда открывающей скобке одного типа соответствует закрывающая скобка другого типа. Требуется проверить правильность расстановки скобок во входной строке.

Польская инверсная запись.

История. Польская инверсная запись (обратная польская нотация) была разработана австралийским философом и специалистом в области теории вычислительных машин Чарльзом Хэмблином в середине 1950-х гг. на основе польской нотации, которая была предложена в 1920 г. польским математиком Яном Лукасевичем. Работа Хэмблина была представлена в июне 1957 г.

Формы записи бинарных операций (пусть задана бинарная операция ◊ над операндами A и B):

– *инфиксная (обычная) форма* A ◊ B;
– *префиксная форма* ◊ A B;
– *постфиксная форма (польская инверсная запись)* A B ◊.

Формы записи унарных операций (пусть задана унарная операция ◊ над операндом A):

– *префиксная (обычная) форма* ◊ A;
– *постфиксная форма (польская инверсная запись)* A ◊.

Инфиксная `((A+B*(D-E))/(F+G)`

Префиксная `/+A*B-DE+FG`

Постфиксная `ABDE-*+FG+/`

Особенности постфиксной записи.

Порядок выполнения операций однозначно задаётся порядком следования знаков операций в выражении, поэтому отпадает необходимость использования скобок и введения приоритетов операций.

Выражение читается слева направо. Когда в выражении встречается знак операции, выполняется соответствующая операция над двумя последними встретившимися перед ним операндами в порядке их записи. Результат операции заменяет в выражении последовательность её операндов и её знак, после чего выражение вычисляется дальше по тому же правилу.

Результатом вычисления выражения становится результат последней вычисленной операции.

В отличие от инфиксной записи, невозможно использовать одни и те же знаки для записи унарных и бинарных операций.

Задачи, возникающие при работе с польской инверсной записью:

– перевод выражения из инфиксной формы в постфиксную;
– вычисление выражения, записанного в постфиксной форме.

Алгоритм перевода выражений в постфиксную форму:

1) операнды (обозначены буквами) записываются в строку в том же порядке, в каком встречаются в исходном выражении;

2) знак арифметической операции заносится в стек при условии, что приоритет данной операции выше приоритета предыдущей операции. В противном случае в строку записывается предыдущий символ операции, а найденный заносится в стек;

3) открывающая скобка заносится в стек. Считается, что ее приоритет ниже приоритета всех арифметических операций;

4) при нахождении закрывающей скобки все содержимое стека до первой открывающей скобки удаляется и записывается в строку.

Открывающая скобка удаляется из стека и в строку не записывается.

***Алгоритм перевода выражений в постфиксную форму* (*псевдокод*) (*для случая односимвольных операндов*)**

```
// Стек и выходная строка изначально пусты
while (входная строка не закончилась) {
c = очередной символ входной строки;
switch (c) of {
     операнд: переносим c в выходную строку;
break;
   ′(′: заносим c в стек;
break;
′)′: переносим из стека в выходную строку все
символы, вплоть до ′(′;
   открывающую скобку выталкиваем из стека;
         break;
знак операции:
if (стек пуст)
заносим c в стек;
   else {
do {
     извлекаем из стека символ t;
if ((t == ′(′) ||
            (приоритет(c) > приоритет(t)){
        заносим t в стек;
break;       }
else
        переносим t в выходную строку;
     } while (стек не пуст);
заносим c в стек;
   }  // else
 } // switch
} // while
переносим оставшиеся в стеке символы с выходную
строку, извлекая их из стека по одному;
```

Алгоритм вычисления выражения, находящегося в постфиксной форме (предполагается, что операнды односимвольные, а их значения известны)

```
// Стек изначально пуст
while (входная строка не закончилась) {
c = очередной символ входной строки;
switch (c) of {
```

```
        операнд: заносим с в стек; break;
     знак операции:
            извлекаем из стека нужное для выполнения
                операции количество операндов;
           вычисляем значение операции;
    заносим его в стек;
      } // switch
    } // while
    извлекаем из стека полученное значение выражения;
```

Упражнения.

1 Реализовать стек с помощью указателей.

2 Написать программу сортировки вагонов.

Задача сортировки вагонов. *Имеется* ***2n*** *вагонов (****n*** *черных и* ***n*** *белых). Нужно составить состав так, чтобы вагоны чередовались. Можно использовать операции* **`ВТупик`**, **`ИзТупика,`** **`Мимо`**. *В тупике может поместиться* ***n*** *вагонов.*

Абстрактный тип данных «очередь».

Очередь – это специальный тип списка, в котором все вставки выполняются в одном конце, называемом последним (**REAR**), а удаления выполняются в другом конце, называемом передним (**FRONT**).

Очередь. Это структура данных, представляющая собой совокупность однотипных элементов, над которой определены две основных операции:

1) вставка в очередь;

2) извлечение из очереди. При этом извлекается тот элемент, который был первым вставлен в очередь. В соответствии с правилами этой операции очередь еще называют структурой типа FIFO («first in – first out»).

Никакие другие операции над классической очередью недопустимы.

Из механизма FIFO следует, что в очереди доступны два элемента – первый и последний.

Физическая организация очереди может быть различная.

Первый способ – в виде списков с указателями на первый и последний элементы списка.

Другой способ организации очереди связан с хранением ее элементов в виде массива. При этом нам надо дополнительно хранить еще два индекса – т. н. ***голову*** и ***хвост*** очереди.

Как и в случае со стеками, в этих переменных можно хранить как индекс занятого, так и индекс свободного элемента. Однако наиболее удобно хранить в качестве хвоста очереди индекс свободного элемента массива, в который будет вставлен новый элемент очереди, а в качестве головы – индекс занятого элемента, который будет удален из очереди первым.

При работе с очередью необходимо обрабатывать следующие нештатные ситуации:

– очередь пуста, а делается попытка удаления элемента из очереди;

– очередь заполнена, а делается попытка вставить новый элемент;

– кроме того, необходимо решить проблему сдвига элементов: если при добавлении очередного элемента указатель хвоста очереди станет больше, чем размер массива, а указатель на голову указывает не на 1-й элемент массива, все элементы передвигаются в начало массива.

Схематическое изображение линейной очереди представлено на рисунке 6.

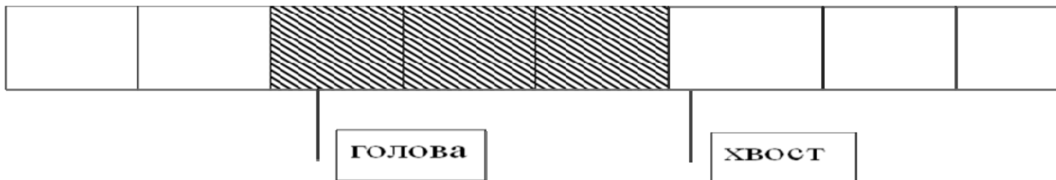

Рисунок 6 – Схематическое изображение линейной очереди

Чтобы избежать ситуации перемещения элементов, используют т. н. ***кольцевую*** или ***циклическую*** очередь: после того, как заполнены элементы массива с большими номерами, заполнение очереди продолжается с области меньших номеров.

Тогда становится понятным, как различить ситуации ***«очередь полна»*** (свободен единственный элемент) и **«*очередь пуста*».**

Программная реализация циклической очереди

```
int *Queue,
Front=0, //голова очереди
Rear=0,  //хвост очереди
N;       //размер очереди
…
Queue = new int[N+1]; // Выделение памяти
int k;
//push(заносится значение k)
if ((Rear+1 == Front)|| ((Rear==N)&&(Front==0))
throw ”Queue is full”;
Queue[Rear++] = k;
if (Rear > N) Rear = 0;
//pop(значение извлекается в k)
if (Rear == Front)
throw ”Queue is empty”;
k = Queue[Front++];
if (Front > N) Front = 0;
```

Если в программе неоднократно выполняются действия над очередью, то красивее описать очередь как структуру и все действия реализовать как функции:

```
struct T {     // тип элементов очереди
int x;
int y; };
struct TQueue // тип циклическая очередь
{
T* Q;
int Front;
int Rear;
int N;
};
TQueue Qu; // переменная циклическая очередь

// функция создания пустой очереди
void Create_Queue(TQueue &Queue,int count)
{
Queue.N=count;
Queue.Q=new T [Queue.N+1];
     Queue.Front=0;
     Queue.Rear=0;
}

void Push(TQueue &Queue,const T &k)
// добавление нового элемента в  очередь
{
     if ((Queue.Rear+1==Queue.Front)||
     ((Queue.Rear==Queue.N)&&(Queue.Front==0)))
throw "Queue is full!";
Queue.Q[Queue.Rear++]=k;
if (Queue.Rear>Queue.N)
    Queue.Rear=0;
}

T Pop(TQueue &Queue)
// извлечение элемента из очереди
{
if (Queue.Rear == Queue.Front)
  throw "Queue is empty!";
T k = Queue.Q[Queue.Front++];
if (Queue.Front > Queue.N)
```

```
Queue.Front = 0;
return k;
}
```

Деревья.

Определение графа.

Неориентированный граф G – это упорядоченная пара ***G* = (*V*, *E*)**, где ***V*** – конечное множество ***вершин*** или ***узлов***; ***E*** – множество неупорядоченных пар различных вершин, называемых ***рёбрами***.

Вершины ***u*** и ***v*** называются ***концевыми*** вершинами ребра ***e* = (*u*, *v*)**.

Два ребра называются ***смежными***, если они имеют общую концевую вершину. Если пары множества ***E*** упорядочены, то граф является ***ориентированным***, а элементы множества ***E*** – ***дугами***.

В неориентированном графе $(v,w) = (w,v)$.

Путь – это последовательность ребер вида $(v_1,v_2), (v_3,v_4),\dots, (v_{n-1},v_n)$.

Говорят, что этот путь идет из вершины v_1 в вершину v_n и имеет длину n-1. Часто его обозначают как последовательность вершин $v_1,v_2,v_3,v_4,\dots, v_{n-1},v_n$.

Путь – конечная последовательность вершин, в которой каждая вершина (кроме последней) соединена со следующей в последовательности вершин ребром.

Путь называется ***простым***, если все ребра и все узлы на нем, кроме, может быть, первого и последнего, различны.

Определение дерева.

Дерево – связный (ориентированный или неориентированный) граф, не содержащий циклов (для любой вершины есть один и только один способ добраться до любой другой вершины).

Корневое дерево.

Корневое дерево определяется как конечное множество ***T*** одного или более узлов со следующими свойствами:

– существует один корень дерева ***T*** ;

– остальные узлы (за исключением корня) распределены среди непересекающихся множеств $T_1,\dots,T_m$, и каждое из множеств является деревом; деревья $T_1,\dots,T_m$ называются ***поддеревьями*** корня ***T***.

Определения, связанные с деревьями.

Степень узла – количество поддеревьев узла.

Концевой узел (лист) – узел со степенью нуль.

Уровень узла определяется следующим образом:

– уровень корня дерева ***T*** равен нулю;

– уровень корней поддеревьев любой вершины дерева на единицу больше, чем уровень этой вершины.

Высота дерева – максимальное значение уровня какой-либо вершины дерева.

Глубина узла v в дереве – длина пути из корня в ***v***.

Высота узла v в дереве – это длина самого длинного пути из **v** в какой-нибудь лист.

Высота дерева – это высота его корня.

Уровень узла равен разности высоты дерева и глубины узла.

Упорядоченным называется дерево, в котором множество сыновей каждого узла упорядочено.

При описании деревьев могут быть использованы термины из ботаники («корень», «лист», «ветвь») либо из генеалогии («отец», «сын», «предок», «потомок», «брат»).

Лес – множество (обычно упорядоченное), не содержащее ни одного непересекающегося дерева или содержащее несколько непересекающихся деревьев.

Ориентированное дерево – это ориентированный граф без циклов, в котором в каждую вершину, кроме одной, называемой *корнем ориентированного дерева*, входит одно ребро. В корень ориентированного дерева не входит ни одного ребра. Иногда термин «ориентированное дерево» сокращают до «дерева».

Дерево – это ориентированный граф без циклов, удовлетворяющий следующим условиям:

– имеется в точности один узел, называемый корнем, в который не входит ни одно ребро;

– в каждый узел, кроме корня входит ровно одно ребро; из корня к каждому узлу идет путь (единственный).

Обходы деревьев.

Обход дерева – операция, связанная с посещением его вершин (под посещением понимается любая операция над вершиной дерева, не затрагивающая структуру дерева).

При обходе каждая вершина должна быть посещена ровно один раз.

Прямой обход дерева:

– посетить корень;

– обойти все поддеревья в направлении слева направо.

Прямой обход (*просмотр в глубину, просмотр сверху вниз*) определяется следующим рекурсивным алгоритмом:

– посетить корень;

– обойти все поддеревья корня, начиная с самого левого.

Обратный обход дерева:

– обойти все поддеревья в направлении слева направо;

– посетить корень.

Обратный обход (*просмотр снизу вверх*) дерева определяется следующим рекурсивным алгоритмом:

– обойти все поддеревья корня, начиная с самого левого;

– посетить корень.

Обход дерева по уровням:

– посетить корень;

– посетить вершины 1-го, 2-го и т. д. уровней в направлении слева направо.

Обход деревьев по уровням. Вначале посещается корень дерева, затем – его сыновья, начиная с самого левого, далее – внуки. Так продолжается до тех пор, пока все вершины не посещены.

Представление деревьев.

Физическая реализация деревьев может быть выполнена различными способами.

1 Поскольку деревья являются частным случаем понятия графа, то для них можно применить любое представление, разработанное для графа общего вида (например, матрицу смежности или матрицу инцидентности).

2 Существуют представления, ориентированные на деревья. Наиболее простым и понятным из них является т. н. *каноническое* представление дерева, когда каждая вершина содержит ссылку на своего отца (корень, естественно, содержит пустую ссылку).

Такое представление удобно, когда надо представить не одно дерево, а *лес* из нескольких деревьев. Но операций, которые удобно выполнять с помощью такого представления, не слишком много. Так, можно легко определить предков каждой вершины, но не ее потомков. Кроме того, модификация дерева затруднена при этом представлении.

3 Следующее представление дерева заключается в том, что каждая его вершина содержит список ссылок на сыновей этой вершины. Теперь можно легко выполнить поиск всех потомков конкретной вершины (но не ее предков!).

Часто два последних представления комбинируют, получая нечто вроде двунаправленного списка. Хотя полученная структура и является несколько избыточной, она позволяет легко проходить дерево в обоих направлениях: от листьев к корням и обратно.

Рассмотрим конкретную реализацию этого представления на примере бинарных деревьев.

Двоичные деревья.

Двоичное (бинарное) дерево (ориентированное) – это ориентированное дерево, в котором исходящие степени вершин (число исходящих рёбер) не превосходят 2.

На двоичном дереве основаны следующие структуры данных:

– двоичное дерево поиска;
– двоичная куча;
– красно-чёрное дерево;
– АВЛ-дерево;
– фибоначчиева куча.

Двоичное дерево поиска.

Двоичное дерево поиска (бинарное поисковое дерево, БПД) – структура данных, предназначенная для выполнения следующих операций:

– поиска элемента по значению;
– добавления нового элемента;
– удаления элемента из дерева.

Структура БПД.

1 Сыновья каждой вершины различаются: выделяются левый и правый сыновья.

2 Значения хранятся в каждой вершине дерева.

3 Для любого узла значения в левом поддереве меньше, а значения в правом поддереве больше значения, хранящегося в корне.

Бинарное поисковое дерево.

Концевой обход бинарного дерева *(симметричный или внутренний):*

– обойти левое поддерево;
– посетить корень;
– обойти правое поддерево.

Механизм поиска в БПД.

Если дерево пусто, то поиск завершился неудачно. В противном случае, сравниваем искомое значение со значением корня. Если эти значения равны, то искомое значение найдено; иначе, рекурсивно выполняем эту же процедуру для левого или правого поддерева в зависимости от того, что больше – значение корня или искомое значение.

Поиск в БПД требует в лучшем случае O(log $\boldsymbol{n}$) операций сравнения, где $\boldsymbol{n}$ – число элементов.

Однако если дерево не сбалансировано (т. е. длина одной из ветвей существенно больше длины другой), то эффективность поиска снижается и в худшем случае может достичь O($\boldsymbol{n}$) операций, когда дерево вырождается в единственную ветвь.

Поиск значения в БПД (пусть *K* – искомое значение; *T* – значение в корне дерева).

```
if (K==T) поиск завершен успешно
else if (K<T)
if (есть левый сын)
  выполнить поиск в левом поддереве
else
  поиск завершен неудачно
 else
if (есть правый сын)
  выполнить поиск в правом поддереве
```

```
else
  поиск завершен неудачно
```

Вставка новой вершины в БПД:

– если БПД пусто, создаем единственную вершину и делаем ее корнем;
– иначе, выполняем поиск вершины с добавляемым значением;
– если поиск успешен, вставка невозможна;
– в противном случае, добавляем новую вершину и делаем ее сыном (левым или правым) той вершины, на которой остановились в процессе поиска.

Физическая реализация БПД.

Наиболее удобной реализацией БПД представляется задание каждой его вершины в динамической памяти. В статическую область помещается лишь указатель на корень дерева:

```
struct TreeItem{
int Info;
TreeItem* Father;
TreeItem* LSon;
TreeItem* RSon;
TreeItem(){LSon=RSon=NULL;}
 };

TreeItem* Root = NULL;
// пустое дерево

bool find(TreeItem* R,int a, TreeItem* &t)
// поиск элемента
// при удачном поиске функция возвращает true,
// а параметр t содержит указатель на
// найденную вершину;
// при неудачном поиске функция возвращает
// false, а параметр t содержит указатель на
// вершину, на которой завершился поиск;
// при попытке поиска в пустом дереве
// параметр t содержит NULL

{ if (R == NULL)
 {
  t = NULL;
  return false;
 }
 t = R;
 for (;;)
 {
```

```
  if (t->Info == a)
         return true;
  if (t->Info > a)
  {
     if (t->LSon == NULL)
return false;
     t = t->LSon;
  }   else   {
     if (t->RSon == NULL)
return false;
     t = t->RSon;
  }
 }
}

bool Insert(TreeItem* &R,int info)
//вставка элемента по значению
//в бинарное поисковое дерево;
//функция возвращает false при обнаружении дубликата

{ TreeItem *s,*q;  if (find(R,info,s))
  return false;
  q = new TreeItem;
q->Info = info;
if (s == NULL)
 { R = q;
   q->Father = NULL;
 }
 else
 { q->Father = s;
 if (s->Info < info)
s->RSon = q;
else
s->LSon = q;
 }
 return true;
}

void print(TreeItem* &s)
{ cout << s->Info << " ";}

void post_order_walk(TreeItem* s,
void vizit(TreeItem* &s))
 // обратный обход дерева
 // выполняя действия над всеми элементами
```

```
 // дерева в функции visit

{
 if (s != NULL)
 {
  post_order_walk(s->LSon,vizit);
  post_order_walk(s->RSon,vizit);
  vizit(s);
 }
}

void EraseItem(TreeItem * &s)
// удаление листа дерева
{
if (s->Father != NULL)
{ if ((s->Father)->LSon == s)
     (s->Father)->LSon = NULL;
  else
         (s->Father)->RSon = NULL;
 }
else Root=NULL; //если лист-корень
delete s;  s = NULL;
}

void post_order(TreeItem* R,
void vizit(TreeItem* &s))
// вспомогательная функция
{
 if (R == NULL) // проверка на пустоту
 {
  cout << "дерево пустое" << endl;
  return;
 }
 post_order_walk(R, vizit);
}
```

Для уничтожения дерева применяется обратный обход. Концевой обход применить нельзя, т. к. посещение вершины приведет к ее удалению и понятие «правое поддерево» станет неопределенным.

```
void Destroy(TreeItem * &R)
// уничтожение дерева
{  post_order(R, EraseItem);
R=NULL;
}
```

```
int main()
{
setlocale(LC_ALL,".1251");
// распечатаем пустое дерево
post_order(Root,print);
int x;
TreeItem* s;
// добавим 8 элементов
for (int i=1; i<9; i++)
{
  cout<<"введите вершину ”<< i << " : ";
  cin >> x;
  Insert(Root,x);
}

post_order(Root,print);
// распечатаем дерево
cout << endl;
// удалить лист
cout <<”какой удалить лист? ”;
cin >> x;
if (find(Root,x,s))
  EraseItem(s); // удаление листа дерева
 else
  cout<<”такого листа нет”<<endl;

post_order(Root,print);
// распечатаем дерево
cout << endl;
// удалить лист  cout << “какой удалить лист? ";
 cin >> x;  if (find(Root,x,s))
  EraseItem(s); // удаление листа дерева  else
  cout<<”такого листа нет”<<endl;

post_order(Root,print);
// распечатаем дерево
cout << endl;
Destroy(Root);// удаление дерева
post_order(Root,print);
// распечатаем пустое дерево
 return 0;
}

void EraseNode(TreeItem* &s)
```

```
// удаление вершины, имеющей не более одного сына
{ TreeItem *q;
if (s->LSon != NULL)
 q=s->LSon;
else
 q=s->RSon;
if (q!=NULL)
 q->Father=s->Father;
if (s->Father == NULL)
      Root=q;
else
 if ((s->Father)->LSon == s)
         (s->Father)->LSon = q;
 else
         (s->Father)->RSon = q;
 delete s;
 s=NULL;
}

 bool Erase(TreeItem* &R,int info)
// удаление вершины по значению
{
 TreeItem *s,*q;
 if (!find(R,info,s))
  return false;
 else
 { if ((s->LSon != NULL)&&(s->RSon != NULL))
   {
q=s->RSon;
     while (q->LSon!=NULL)
 q=q->LSon;
     s->Info=q->Info;
EraseNode(q);
   }
  else EraseNode(s);
  return true;
 }
}

int main() {
 setlocale(LC_ALL,".1251");
 // распечатаем пустое дерево обратным обходом
 post_order(Root,print);
 int n,x;
 TreeItem* s;
```

```
 cout << "сколько вершин?" << endl;
 cin >> n;
 // добавим n элементов
 for (int i=1; i<=n; i++)
 {
  cout << "введите вершину " << i << " : ";
  cin >> x;
  Insert(Root,x);
 }

 post_order(Root,print);// распечатаем дерево
 cout << endl;
 cout << "какую удалить вершину? ";
 cin >> x;
 if (Erase(Root,x)); // удаление вершины дерева
else
  cout << "такой вершины нет"  << endl;
post_order(Root,print);// распечатаем дерево
cout << endl;
cout << "какую удалить вершину? ";
cin >> x;
if (Erase(Root,x)); // удаление вершины дерева
else
  cout << "такой вершины нет"  << endl;
post_order(Root,print);//распечатаем дерево
cout << endl;
Destroy(Root);// удаление дерева
post_order(Root,print);// распечатаем дерево
return 0;
}
```

Выполнить программу для следующего текста:

- 8 вершин: 8 5 10 6 3 12 9 11;
- нарисовать построенное дерево;
- сделать обратный обход;
- удалить: 8;
- нарисовать полученное дерево;
- сделать обратный обход;
- удалить: 5;
- нарисовать полученное дерево;
- сделать обратный обход.

2.2 Тема 2. Классы в C++. Объектно-ориентированное программирование

Общие сведения о классах.

Класс – составной тип данных, элементами которого являются функции и переменные. В основу понятия «класс» положен тот факт, что «над объектами можно совершать различные операции». Свойства объектов описываются с помощью полей классов, а действия над объектами – с помощью функций, которые называются методами класса. Класс имеет *имя*, состоит из полей, называемых *членами класса*, и функций – *методов класса*.

Описание класса имеет следующий формат:

```
    class name // name - имя класса
    { private:
    // Описание закрытых членов и методов класса
protected:
    // Описание защищенных членов и методов класса
public:
    // Описание открытых членов и методов класса
    }
```

Открытые и закрытые члены класса.

В отличие от полей структуры, доступных всегда, в классах могут быть члены и методы различного уровня доступа:

– *открытые* `public` (публичные), вызов открытых членов и методов класса осуществляется с помощью оператора «.» («точка»);

– *закрытые* `private` (приватные), доступ к которым возможен только с помощью открытых методов;

– *защищенные* методы `(protected)`.

После описания класса необходимо описать переменную типа `class`. Например,

```
name_class name;
```

здесь `name_class` – имя класса;

`name` – имя переменной.

В дальнейшем переменную типа `class` будем называть «*объект*» или «*экземпляр класса»*. Объявление переменной типа class (в данном примере переменная `name` типа `name_class`) называется *созданием (инициализацией) объекта (экземпляра класса)*.

После описания переменной можно обращаться к членам и методам класса. Обращение к членам и методам класса осуществляется аналогично обращению к полям структуры с помощью оператора «.» («точка»).

```
name.p1; //Обращение к полю p1
//экземпляра класса name.
name.f1(par1,par2,…parn);//Обращение к методу f1
```

```
    //экземпляра класса name,
    //par1, par2, …, parn – список формальных
//параметров функции f1.
```

Члены класса доступны из любого метода класса и их не надо передавать в качестве параметров функций-методов.

Задача 1. Рассмотрим класс `complex` для работы с комплексными числами.

```
    class complex //Определяем класс complex { public:
double x;    //Действительная часть комплексного числа.
    double y; //Мнимая часть комплексного числа.
    //Метод класса complex – функция modul,
    //для вычисления модуля комплексного числа.
    double modul()
    {
    return pow(x*x+y*y,0.5);
    }
    //Метод класса complex – функция argument,
    //для вычисления аргумента комплексного числа.
    double argument()

    {
    return atan2(y,x)*180/PI;
    }
    //Метод класса complex – функция show_complex,
    //для вывода комплексного числа.
    void show_complex()
    { if (y>=0)
    //Вывод комплексного числа с положительной
    //мнимой частью.
    cout<<x<<"+"<<y<<"i"<<endl; else
    //Вывод комплексного числа с отрицательной
    //мнимой частью.
    cout<<x<<y<<"i"<<endl;
    }
    }; int main() {
    //Определяем переменную chislo типа complex.
    complex chislo;
    //Определяем действительную часть комплексного числа.
    chislo.x=3.5;
    //Определяем мнимую часть комплексного числа.
    chislo.y=-1.432;
    //Вывод комплексного числа, chislo.show_complex() –
    //обращение к методу класса.
    chislo.show_complex();
    //Вывод модуля комплексного числа, chislo.modul() –
    //обращение к методу класса.
    cout<<"Modul' chisla="<<chislo.modul();
    //Вывод аргумента комплексного числа,
```

```
    //chislo.argument() - обращение к методу класса.
    cout<<endl<<"Argument
chisla="<<chislo.argument()<<endl;
    return 1;
    }
```

Использование открытых членов и методов позволяет получить полный доступ к элементам класса, однако это не всегда хорошо. Если все члены класса объявить открытыми, то при непосредственном обращении к ним появится потенциальная возможность внести ошибку в функционирование взаимосвязанных между собой методов класса. Поэтому общим принципом является следующее: «чем меньше открытых данных о классе используется в программе, тем лучше». Уменьшение количества публичных членов и методов позволит минимизировать количество ошибок. Желательно, чтобы все члены класса были закрытыми и тогда невозможно будет обращаться к членам класса непосредственно с помощью оператора «.» Количество открытых методов также следует минимизировать.

Если в описании элементов класса отсутствует указание метода доступа, то члены и методы считаются закрытыми (`private`). Принято описывать методы за пределами класса.

Задача 2. Изменим рассмотренный ранее пример класса `complex`. Добавим метод vvod, предназначенный для ввода действительной и мнимой частей числа, члены класса и метод `show_complex` сделаем закрытыми, а остальные методы открытыми. Текст программы будет иметь вид:

```
    class complex {
    //Открытые методы.
    public:
    void vvod(); double modul(); double argument();
//Закрытые члены и методы.
    private: double x; double y;
    void show_complex(); };
    //Описание открытого метода vvod класса complex.
    void complex::vvod()
    {
    cout<<"Vvedite x\t"; cin>>x; cout<<"Vvedite y\t";
cin>>y;
    // Вызов закрытого метода show_complex
    //из открытого метода vvod.
    show_complex();
    }
    //Описание открытого метода modul класса complex.
    double complex::modul()
    {
    return pow(x*x+y*y,0.5);
```

```
    }
    //Описание открытого метода argument класса complex.
double complex::argument()
    {
    return atan2(y,x)*180/PI;
    }
    //Описание закрытого метода modul класса complex.
    void complex::show_complex()
    { if (y>=0)
       cout<<x<<"+"<<y<<"i"<<endl;
      else cout<<x<<y<<"i"<<endl;
    }
    int main()
    {
    complex chislo; chislo.znach();
    cout<<"Modul kompleksnogo chisla="<<chislo.modul();
cout<<endl<<"Argument kompleksnogo
chisla="<<chislo.argument()<<endl;
    return 1; }
```

В рассмотренном примере показано совместное использование открытых и закрытых элементов класса. Разделение на открытые и закрытые в этом примере несколько искусственное, оно проведено только для иллюстрации механизма совместного использования закрытых или открытых элементов класса. Если попробовать обратиться к методу `show_complex()` или к членам класса `x`, `y` из функции `main`, то компилятор выдаст сообщение об ошибке (доступ к элементам класса запрещен). При создании в программе экземпляра класса формируется указатель `this`, в котором хранится адрес переменной – экземпляра класса. Указатель `this` выступает в роли указателя на текущий объект.

Использование конструкторов.

В предыдущем примере метод `chislo.vvod()` использовался для присвоения начального значения некоторым членам класса, однако для упрощения процесса инициализации объекта предусмотрена специальная функция, которая называется *конструктором*. Имя конструктора совпадает с именем класса, конструктор запускается автоматически при создании экземпляра класса (при объявлении переменной типа `class`). Функции-конструкторы не возвращают значение, но при описании функции конструктора не следует указывать в качестве возвращаемого значения тип `void`.

Конструктор автоматически запускается на выполнение для каждого экземпляра класса при его описании. Чаще всего конструктор служит для инициализации полей экземпляра класса.

Задача 3. Добавим в созданный в предыдущем примере класс `complex` конструктор:

```
//Объявляем класс complex.
//Внутри класса указаны, только прототипы методов,
//а сами функции описаны за пределами класса.
 class complex { public:
//Прототип конструктора класса. complex();
//Прототип метода modul(). double modul();
//Прототип метода argument().
double argument(); private: double x; double y;
//Прототип метода show_complex().
void show_complex(); };
// Главная функция.
int main()
{
//Описываем экземпляр класса, при выполнении
//программы после создания  переменной
//автоматически вызывает констуктор.
complex chislo;
cout<<"Modul kompleksnogo chisla="<<chislo.modul();
cout<<endl<<"Argument kompleksnogo
chisla="<<chislo.argument()<<endl; return 1; }
//Текст функции конструктор класса complex.
complex::complex()
{
cout<<"Vvedite x\t"; cin>>x; cout<<"Vvedite y\t";
cin>>y; show_complex(); }
//Текст метода modul класса complex.
double complex::modul()
{
return pow(x*x+y*y,0.5); }
//Текст метода argument класса complex.
double complex::argument()
{
return atan2(y,x)*180/PI; }
//Текст метода show_complex класса complex.
void complex::show_complex()
{ if (y>=0) cout<<x<<"+"<<y<<"i"<<endl;
else cout<<x<<y<<"i"<<endl;
}
```

Задача 4. С использованием классов решить следующую задачу. Заданы координаты n точек в *k*-мерном пространстве. Найти точки, расстояние между которыми наибольшее и наименьшее.

Для решения задачи создадим класс `prostr`.

Члены класса:

– `int n` – количество точек;

– `int k` – размерность пространства;

– `double **a` – матрица, в которой будут храниться координаты точек; `a[i][j]` – *i*-я координата точки с номером *j*;

– `double min` – минимальное расстояние между точками в k-мерном пространстве;

– `double max` – максимальное расстояние между точками в k-мерном пространстве;

– `int imin, int jmin` – точки, расстояние между которыми минимально;

– `int imax, int jmax` – точки, расстояние между которыми максимально.

Методы класса:

– `prostr()` – конструктор класса, в котором определяются *n* – количество точек, *k* – размерность пространства, выделяется память для матрицы `a` координат точки и вводятся координаты точек;

– `poisk_max()` – функция нахождения точек, расстояние между которыми наибольшее;

– `poisk_min()` – функция нахождения точек, расстояние между которыми наименьшее;

– `vivod_result()` – функция вывода результатов: значений `min, max, imin, jmin, imax, jmax`;

– `delete_a()` – освобождение памяти, выделенной для матрицы `a`. В главной программе необходимо будет описать экземпляр класса и последовательно вызвать методы `poisk_min(), poisk_max(), vivod_result(), delete_a()`.

Текст программы:

```
//Описываем класс prostr
class prostr{
//Открытые методы класса.
public:
//Конструктор класса  prostr();
double poisk_min();
double poisk_max();
int vivod_result();
int delete_a();
//Все члены класса - закрытые.
private: int n;
int k;
double **a;
double min;
double max;
int imin;
int jmin;
int imax;
```

```
int jmax;
};
//Главная функция
void main() {
//Описание переменной - экземпляра класса prostr.
prostr x;
//Вызов метода poisk_max для поиска
//максимального расстояния между
//точками в k-мерном пространстве;
x.poisk_max();
//Вызов метода poisk_min для поиска
//максимального расстояния между
//точками в k-мерном пространстве;
x.poisk_min();
//Вызов метода vivod_result для вывода результатов
x.vivod_result();
//Вызов функции delete_a.
x.delete_a(); }
//Текст функции конструктор класса prostr.
prostr::prostr()
{ int i,j;
cout<<"VVedite razmernost prostrantva ";
cin>>k;
cout<<"VVedite kolichestvo tochek ";
cin>>n;
a=new double*[k];
for(i=0;i<k;i++)
a[i]=new double[n];
for(j=0;j<n;j++)
{
cout<<"VVedite koordinati "<<j<<" tochki"<<endl;
for(i=0;i<k;i++)
 cin>>a[i][j];
}
}
//Текст метода poisk_max класса prostr.
double prostr::poisk_max()
{
int i,j,l;
double s; for(max=0,l=0;l<k;l++)
 max+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
max=pow(max,0.5);
imax=0; jmax=1;
for(i=0;i<n;i++) for(j=i+1;j<n;j++)
{
```

```
 for(s=0,l=0;l<k;l++)
  s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
 s=pow(s,0.5);  if (s>max)
{
 max=s;
 imax=i;
 jmax=j;
}
}
return 0;
}
//Текст метода poisk_min класса prostr. double
prostr::poisk_min()
{ int i,j,l;
  double s; for(min=0,l=0;l<k;l++)
min+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
min=pow(min,0.5); imin=0; jmin=1;
for(i=0;i<k;i++)
 for(j=i+1;j<n;j++)
{
  for(s=0,l=0;l<k;l++)
    s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
  s=pow(s,0.5);
if (s<min)
{ min=s;
 imin=i;
 jmin=j;
}
}
return 0; }
//Текст метода vivod_result класса prostr.
int prostr::vivod_result()
{ int i,j;
for(i=0;i<k;cout<<endl,i++)
 for (j=0;j<n;j++)
    cout<<a[i][j]<<"\t";
cout<<"max="<<max<<"\t nomera
"<<imax<<"\t"<<jmax<<endl; cout<<"min="<<min<<"\t nomera
"<<imin<<"\t"<<jmin<<endl; return 0;
}
//Текст функции деструктор класса prostr.
int prostr::delete_a()
{
```

```
delete [] a;
return 0;
}
```

Так же как и любые другие функции, конструкторы могут *перегружаться*. Перепишем предыдущий пример, добавив в него перегружаемый конструктор, в который можно передавать значения n и k. В этом примере экземпляр класса можно описывать, например, так: `prostr x;` в этом случае конструктор вызывается без параметров или так: `prostr x (3,5);` в этом случае вызывается перегружаемый конструктор с параметрами.

```
class prostr{
public:
prostr(int,int);
prostr();
double poisk_min();
double poisk_max();
int vivod_result();
int delete_a();
private:
int n; int k; double **a;
double min; double max;
int imin; int jmin; int imax; int jmax;
};
void  main()
{
//Можно вызывать конструктор с параметрами prostr
x(3,5);
//или без prostr x; в этом случае будет вызываться
//тот же конструктор, что и в предыдущем примере.
x.poisk_max();
x.poisk_min();
x.vivod_result();
x.delete_a();
}
prostr::prostr()
{ int i,j;
cout<<"VVedite razmernost prostrantva";
cin>>k;
cout<<"VVedite kolichestvo tochek ";
cin>>n;
a=new double*[k];
for(i=0;i<k;i++)
 a[i]=new double[n];
for(j=0;j<n;j++)
```

```
{
cout<<"VVedite koordinati "<<j<<" tochki"<<endl;
for(i=0;i<k;i++)
cin>>a[i][j];
}
}
//Текст второго конструктора
//Нельзя в качестве формальных параметров
//конструктора использовать переменные n и k,
//потому что это имена членов класса.
//Если в качестве формальных параметров указать n и
k,
//то внутри конструктора будут использоваться
локальные
//переменные n и k, но при этом члены класса prost n
и k
//будут не определены.
prostr::prostr(int k1, int n1)
//Входными параметрами являются размерность
//пространства n1 и количество
//точек в пространстве k1.
{
int i,j;
//Присваиваем членам класса n и k значения
//входных параметров конструктора
k=k1; n=n1;
a=new double*[k];
for(i=0;i<k;i++)
 a[i]=new double[n];
for(j=0;j<n;j++)
{
cout<<"VVedite koordinati "<<j<<"tochki"<<endl;
for(i=0;i<k;i++) cin>>a[i][j];
}}
double prostr::poisk_max()
{ int i,j,l;
double s;
for(max=0,l=0;l<k;l++)
max+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
max=pow(max,0.5); imax=0;jmax=1;
for(i=0;i<n;i++) for(j=i+1;j<n;j++)
{
 for(s=0,l=0;l<k;l++)
  s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
 s=pow(s,0.5);
```

```
    if (s>max)
    {
    max=s; imax=i; jmax=j;
    }
    }
    return 0;
    }
    double prostr::poisk_min()
    {
    int i,j,l; double s;
    for(min=0,l=0;l<k;l++)
    min+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
min=pow(min,0.5); imin=0;jmin=1;
    for(i=0;i<k;i++)
     for(j=i+1;j<n;j++)
    {
      for(s=0,l=0;l<k;l++)
       s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
      s=pow(s,0.5);
    if (s<min)
    {
    min=s; imin=i; jmin=j;
    }
    }
    return 0;
    }
    int prostr::vivod_result()
    {
     int i,j;
     for(i=0;i<k;cout<<endl,i++)
     for (j=0;j<n;j++) cout<<a[i][j]<<"\t";
    cout<<"max="<<max<<"\t nomera
"<<imax<<"\t"<<jmax<<endl; cout<<"min="<<min<<"\t nomera
"<<imin<<"\t"<<jmin<<endl;
    return 0;
    }
    int prostr::delete_a()
    {
    delete [] a;
    return 0;
    }
```

Теперь рассмотрим, как можно использовать *параметры по умолчанию* в конструкторе. Оставим в данной задаче только конструктор с параметрами $n1$ и $k1$, но сделаем их по умолчанию равными 2 и 10 соответственно. В этом

случае при описании экземпляра класса без параметров *n* по умолчанию будет равно 2, а *k* – 10. Однако можно описать экземпляр класса, передав в него любые значения *n* и *k*. Например, так: `prostr x(3,5);`

```
class prostr{
public:
//Прототип конструктора с параметрами
//по умолчанию k=2, n=10.
//В качестве имен формальных параметров конструктора
//не могут быть выбраны переменные n и k, потому
//что эти имена совпадают с именами членов класса.
prostr(int k1=2,int n1=10);
double poisk_min();
double poisk_max();
int vivod_result();
int delete_a();
private:
int n; int k; double **a;
double min; double max;
int imin; int jmin; int imax; int jmax;
};
void main()
{
//Экземпляр класса можно описывать так
prostr x(2,3);
// или так - prostr x; в этом случае k=2, n=10.
x.poisk_max();
x.poisk_min();
x.vivod_result();
x.delete_a(); }
//Конструктор с параметрами по умолчанию k=2, n=10.
prostr::prostr(int k1, int n1)
{
int i,j; k=k1; n=n1;
a=new double*[k];
for(i=0;i<k;i++)
a[i]=new double[n];
for(j=0;j<n;j++)
{
cout<<"VVedite koordinati "<<j<<" tochki"<<endl;
for(i=0;i<k;i++)
cin>>a[i][j];
}
}
double prostr::poisk_max()
```

```
{
int i,j,l;
double s;
for(max=0,l=0;l<k;l++)
 max+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
max=pow(max,0.5); imax=0; jmax=1;
for(i=0;i<n;i++)
 for(j=i+1;j<n;j++)
 {
  for(s=0,l=0;l<k;l++)
   s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
  s=pow(s,0.5);
if (s>max)
{
 max=s; imax=i; jmax=j;
}
}
return 0;
}
double prostr::poisk_min()
{
int i,j,l; double s;
for(min=0,l=0;l<k;l++)
 min+=(a[l][0]-a[l][1])*(a[l][0]-a[l][1]);
min=pow(min,0.5); imin=0; jmin=1;
for(i=0;i<k;i++)
 for(j=i+1;j<n;j++)
 {
  for(s=0,l=0;l<k;l++)
   s+=(a[l][i]-a[l][j])*(a[l][i]-a[l][j]);
  s=pow(s,0.5);
if (s<min)
{
min=s; imin=i; jmin=j;
}.
}
return 0;
}
int prostr::vivod_result()
{ int i,j;
for(i=0;i<k;cout<<endl,i++)
for (j=0;j<n;j++) cout<<a[i][j]<<"\t";
cout<<"max="<<max<<"\tnomera
"<<imax<<"\t"<<jmax<<endl; cout<<"min="<<min<<"\tnomera
"<<imin<<"\t"<<jmin<<endl;
return 0;
}
```

```
int prostr::delete_a()
{ delete [] a; return 0;}
```

Еще одним видом конструктора является *конструктор копирования*, который позволяет создать копию экземпляра класса. Это актуально, когда необходимы два экземпляра класса с одними и теми же значениями членов класса. Синтаксис заголовка конструктора копирования следующий:

```
public: name_constructor (type_class & name);
```

здесь name_constructor – имя конструктора копирования;

type_class – имя класса, передаваемого в конструктор копирования (в конструктор копирования можно передавать только экземпляры класса);

name – передаваемый в конструктор копирования экземпляр класса.

Еще один специальный конструктор называется *деструктором*. С++ позволяет построить функцию-деструктор, которая выполняется при уничтожении экземпляра класса, что происходит в следующих случаях:

– при завершении программы или выходе из функции;
– при освобождении памяти, выделенной для экземпляра класса.

Деструктор имеет имя ~ имя_класса, не имеет параметров и не может перегружаться. Если деструктор не определен явно, то будет использоваться стандартный деструктор.

Одной из основных особенностей ООП является возможность наследования. *Наследование* – это способ повторного использования программного обеспечения, при котором новые *производные* классы (наследники) создаются на базе уже существующих *базовых* классов (родителей). При создании новый класс является наследником членов и методов ранее определенного базового класса. Создаваемый путем наследования класс является *производным* (derived class), который в свою очередь может выступать в качестве *базового* класса (based class) для создаваемых классов. Если имена методов производного и базового классов совпадают, то методы производного класса перегружают методы базового класса.

При использовании наследования члены и методы, кроме свойств public и private, могут иметь свойство protected. Для одиночного класса описатели protected и private равносильны. Разница между protected и private проявляется при наследовании, закрытые члены и методы, объявленные в базовом классе как protected, в производном могут использоваться как открытые (public). *Защищенные* (protected) члены и методы являются чем-то промежуточным между public и private.

При создании производного класса используется следующий синтаксис:

```
class name_derived_class: type_inheritance base_class
{
// закрытые члены и методы класса
```

```
    ... public: // открытые члены и методы класса
    ... protected: // защищенные члены и методы класса
...
    };
```

здесь `name_derived_class` – имя создаваемого производного класса;

`type_inheritance` – способ наследования.

Возможны следующие способы наследования: `public`, `private` и `protected`;

`base_class` – имя базового типа.

Следует различать тип доступа к элементам в базовом классе и тип наследования (таблица 2).

Таблица 2 – Типы наследования и типы доступа

Способ доступа	Спецификатор в базовом классе	Доступ в производном классе
private	private protected public	нет private private
protected	private protected public	нет protected protected
public	private protected public	нет protected public

При порождении производного класса из базового, имеющего конструктор, конструктор базового типа необходимо вызывать из конструктора производного класса. Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Если в конструкторе производного класса явный вызов конструктора базового типа отсутствует, то он вызывается автоматически без параметров. В случае нескольких уровней наследования, конструкторы вызываются, начиная с верхнего уровня. В случае нескольких базовых типов, их конструкторы вызываются в порядке объявления.

Если в описании класса есть ссылка на описываемый позже класс, то его надо просто объявить с помощью оператора

```
class new_class;
```

Это описание аналогично описанию прототипов функции.

Зачастую при наследовании появляются методы, которые в различных производных классах работают по различным алгоритмам, но имеют одинаковые выходные параметры и возвращаемое значение. Такие методы называются *виртуальными* и описываются с помощью служебного слова virtual.

Задача 5. Рассмотрим абстрактный базовый класс figure (фигура), на базе которого можно построить производные классы для реальных фигур (эллипс, окружность, квадрат, ромб, прямоугольник, треугольник и т. д.). На листинге

приведены базовый класс figure и производные классы _circle (окружность) и RecTangle (прямоугольник).

```
    //Базовый класс figure.
    class figure { public:
    //n – количество сторон фигуры, для окружности n=1.
int n;
    //p – массив длин сторон фигуры,
    //для окружности в p хранится радиус. float *p;
    //Конструктор.
    figure();
    //Метод вычисления периметра фигуры.
    float perimetr();
    //Метод вычисления площади фигуры.
    virtual float square();
    //Метод вывода информации о фигуре: ее название,
    //периметр, площадь и т.д.
    virtual void show_parametri();
    };
    //Конструктор класса фигуры.
    figure::figure()
    {cout<<"This is abstract constructor"<<endl;}
    //Метод вычисления периметра,
    //он будет перегружаться только в классе _circle.
    float figure::perimetr()
    {int i; float psum;
    for(psum=0,i=0;i<n;psum+=p[i],i++);
    return psum;
    }
    //Метод вычисления площади, пока он абстрактный,
    //в каждом классе будет перегружаться
    //реальным методом.
    float figure::square()
    {cout<<"No square abstract figure"<<endl; return 0;
    }
    //Метод вывода информации о фигуре будет
    //перегружаться в каждом производном классе.
    void figure::show_parametri()
    {cout<<"Abstract figure";}
    //Производный класс _circle (окружность),
    //основанный на классе figure.
    class _circle:public figure { public:
    //Конструтор
    _circle();
```

```
    //Перегружаемые методы
perimetr(),square(),show_parametri().
    float perimetr(); virtual float square(); virtual
void show_parametri();
    };
    //Производный класс RecTangle (прямоугольник),     //
основанный на классе figure. class RecTangle:public
figure {public:
    //Конструктор.
    RecTangle();
    //Перегружаемые методы square(),show_parametri().
virtual float square(); virtual void show_parametri();};
    //Главная функция.
    void main() {_circle RR; RR.show_parametri();
    RecTangle PP;
    PP.show_parametri();
    } //Конструктор класса _circle.
    _circle::_circle()
    {cout<<"Parametri okruzhnosti"<<endl;
    //В качестве сторон окружности выступает
    //единственный параметр радиус. n=1; p=new float[n];
cout<<"Vvedite radius";
    cin>>p[0];}
    //Метод вычисления периметра окружности.
    float _circle::perimetr()
    { return 2*PI*p[0];
    }
    //Метод вычисления площади окружности.
    float _circle::square()
    { return PI*p[0]*p[0];
    }
    //Метод вывода параметров окружности.
    void _circle::show_parametri() {
    //Вывод сообщения о том, что это окружность.
cout<<"This is circle"<<endl; //Вывод радиуса окружности.
cout<<"Radius="<<p[0]<<endl; //Вывод периметра
окружности.
    cout<<"Perimetr="<<perimetr()<<endl;
    //Вывод площади окружности.
    cout<<"Square="<<square()<<endl;
    } //Конструктор класса RecTangle.
    RecTangle::RecTangle()
    { cout<<"Parametri rectangle"<<endl;
    //Количество сторон =4. n=4; p=new float[n];
    //Ввод длин сторон прямоугольника.
```

```
     cout<<"Vvedite dlini storon";
     cin>>p[0]>>p[1]; p[2]=p[0]; p[3]=p[1];
     }
     //Метод вычисления площади прямоугольника.
     float RecTangle::square()
     { return p[0]*p[1];
     }
     //Метод вывода параметров прямоугольника.
     void RecTangle::show_parametri() {
     //Вывод сообщения о том, что это прямоугольник.
cout<<"This is Rectangle"<<endl;
     //Вывод длин сторон прямоугольника.
cout<<"a="<<p[0]<<" b="<<p[1]<<endl; //Вывод периметра
прямоугольника.
     //Классе RecTangle вызывает метод perimetr()
     //базового класса (figure).
     cout<<"Perimetr="<<perimetr()<<endl;
     //Вывод площади прямоугольника.
     cout<<"Square="<<square()<<endl;
     }
```

3 Примерный перечень вопросов для написания аудиторной контрольной работы

1 Общие сведения о динамическом распределении памяти и динамических переменных.

2 Указатели, объявление ссылочных переменных.

3 Управление выделением и освобождением динамической памяти на логическом уровне.

4 Создание однонаправленного списка. Добавление элемента в список. Примеры.

5 Создание однонаправленного списка. Удаление элемента из списка. Примеры.

6 Создание двунаправленного списка. Добавление и удаление элемента из начала списка. Примеры.

7 Создание двунаправленного списка. Удаление элемента из списка по ключу. Примеры.

8 Понятие стека. Примеры использования стека в программировании.

9 Основные приемы и особенности работы со стеками. Добавление элемента в стек. Примеры.

10 Основные приемы и особенности работы со стеками. Удаление элемента из стека. Примеры.

11 Способы организации очереди. Примеры.

12 Обработка очередей. Добавление элемента в очередь. Примеры.

13 Обработка очередей. Удаление элемента из очереди. Примеры.

14 Понятие «дерево». Бинарное дерево поиска.

15 Бинарное дерево. Формирование дерева. Добавление узла в дерево поиска.

16 Бинарное дерево. Рекурсивный и нерекурсивный поиск узла в дереве.

17 Бинарное дерево. Алгоритмы обхода дерева.

18 Бинарное дерево. Удаление узла из дерева.

19 Обработка бинарного дерева. Подсчет узлов дерева.

20 Обработка бинарного дерева. Определение высоты дерева.

21 Базовые принципы объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм.

22 Понятие объекта, класса, метода, свойства. Конструкторы, деструкторы.

Список литературы

1 **Батан, С. Н.** Сборник задач по программированию: учебно-методические материалы / С. Н. Батан, Н. В. Кожуренко . – Могилев: МГУ имени А. А. Кулешова, 2015. – 83 с.

2 **Гавриков, М. М.** Теоретические основы разработки и реализации языков программирования: учебное пособие для студентов вузов, обучающихся по специальности «Программное обеспечение вычислительной техники и автоматизированных систем» направления подготовки дипломированных специалистов «Информатика и вычислительная техника» / М. М. Гавриков, А. Н. Иванченко, Д. В. Гринченков; под ред. А. Н. Иванченко . – Москва: Кнорус, 2016. – 177 с.

3 **Гагарина, Л. Г.** Введение в теорию алгоритмических языков и компиляторов: учебное пособие для студентов высших учебных заведений, обучающихся по направлению 09.03.01 «Информатика и вычислительная техника» / Л. Г. Гагарина, Е. В. Кокорева . – Москва: Форум, 2018. – 175 с.

4 **Иванова, Г. С.** Технология программирования: учебник для студентов высших учебных заведений, обучающихся по направлению «Информатика и вычислительная техника» / Г. С. Иванова. – 3-е изд., стер. – Москва: Кнорус, 2018. – 333 с.

5 **Мороз, Л. А.** Технологии программирования и методы алгоритмизации: контрольные задания / Л. А. Мороз. – Могилев: МГУ имени А. А. Кулешова, 2018. – 66 с.

6 **Орлов, С. А.** Теория и практика языков программирования: учебник по направлению «Информатика и вычислительная техника» / С. А. Орлов. – Санкт-Петербург: ПИТЕР, 2013. – 688 с.

7 **Хорев, П. Б.** Объектно-ориентированное программирование с примерами на C# : учебное пособие для студентов высших учебных заведений, обучающихся по направлениям 01.03.02 «Прикладная математика и информатика» и 09.00.00 «Информатика и вычислительная техника» / П. Б. Хорев . – Москва: Форум, 2018. – 197 с.

8 **Батан, С. Н.** Контрольные задания по курсу «Методы программирования и информатика» / С. Н. Батан. – Могилев: МГУ имени А. А. Кулешова, 2008. – 40 с.

9 **Давыдов, В. Г.** Программирование и основы алгоритмизации: учебное пособие / В. Г. Давыдов. – 2-е изд., стер. – Москва: Высшая школа, 2005. – 447 с.

10 **Давыдов, В. Г.** Технологии программирования C++: учебное пособие для вузов / В. Г. Давыдов. – Санкт-Петербург: BHV-Санкт-Петербург, 2005. – 672 с.

11 **Ишкова, Э. А.** C#. Начало программирования / Э. А. Ишкова. – Москва: БИНОМ, 2011. – 333 с.

12 **Канцедал, С. А.** Алгоритмизация и программирование: учебное пособие / С. А. Канцедал. – Москва: ФОРУМ; ИНФРА-М, 2019. – 352 с.

13 **Коплиен, Д.** Программирование на C++: пер. с англ. / Д. Коплиен . – Санкт-Петербург: ПИТЕР, 2005. – 479 с.

14 **Павловская, Т. А.** C/C++. Структурное программирование: практикум / Т. А. Павловская, Ю. А. Щупак. – Санкт-Петербург: ПИТЕР, 2005. – 239 с.

15 **Павловская, Т. А.** C#. Программирование на языке высокого уровня / Т. А. Павловская. – Санкт-Петербург: ПИТЕР, 2009. – 432 с.

16 **Савич, У.** Программирование на C++ / У. Савич. – 4-е изд. – Санкт-Петербург: BHV-Санкт-Петербург, 2004. – 781 с.

17 **Страуструп, Б.** Язык программирования C++ / Б. Страуструп. – спец. изд. – Москва: БИНОМ, 2008. – 1104 с.

18 **Окулов, С. М.** Программирование в алгоритмах / С. М. Окулов. – 3-е изд. – Москва: БИНОМ, 2007. – 383 с.

19 **Франка, П.** C++: учебный курс: пер. с англ. / П. Франка. – Санкт-Петербург: ПИТЕР, 2001. – 528 с.

20 **Шилдт, Г.** C++: руководство для начинающих: пер. с англ. / Г. Шилдт. – 2-е изд. – Москва; Санкт-Петербург; Киев: Вильямс, 2005. – 672 с.

21 Язык C++: учебное пособие / И. Ф. Астахова [и др.] – Минск: Новое знание, 2003. – 203 с.