МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Тульский государственный университет»

Институт Прикладной математики и компьютерных наук
Кафедра «Вычислительная техника»

Утверждено на заседании кафедры
«Вычислительная техника»
«___» _____ 20__г., протокол №____

Заведующий кафедрой

_____А.Н. Ивутин

# МЕТОДИЧЕСКИЕ УКАЗАНИЯ
## по выполнению лабораторных работ по дисциплине (модулю) «Структуры и алгоритмы обработки данных»

**основной профессиональной образовательной программы высшего образования – программы бакалавриата**

по направлению подготовки
**09.03.01 Информатика и вычислительная техника**

с направленностью (профилем)
**Электронно-вычислительные машины, комплексы, системы и сети**

Форма обучения: *заочная*

Идентификационный номер образовательной программы:

Тула 2020 год

**СОДЕРЖАНИЕ**

Лабораторная работа № 1 ..... 3
СПИСКОВЫЕ СТРУКТУРЫ ДАННЫХ ..... 3
Лабораторная работа № 2 ..... 8
КОЛЬЦЕВЫЕ СПИСКИ ..... 8
Лабораторная работа № 3 ..... 11
ОБРАБОТКА СТРОК ..... 11
Лабораторная работа № 4 ..... 14
СТРУКТУРЫ ДАННЫХ ТИПА ДЕРЕВО ..... 14
Лабораторная работа № 5 ..... 25
СТРУКТУРА ДАННЫХ МНОЖЕСТВО ..... 25
Лабораторная работа № 6 ..... 30
КРИПТОГРАФИЯ ..... 30
Лабораторная работа № 7 ..... 33
РЕКУРСИВНЫЕ АЛГОРИТМЫ ..... 33
Лабораторная работа № 8 ..... 36
ОБОБЩЕННОЕ ПРОГРАММИРОВАНИЕ В ЯЗЫКЕ С++ ..... 36
Лабораторная работа № 9 ..... 45
ЖАДНЫЕ АЛГОРИТМЫ ..... 45
Лабораторная работа № 10 ..... 49
ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ ..... 49

## Лабораторная работа № 1.

# СПИСКОВЫЕ СТРУКТУРЫ ДАННЫХ

**Цель работы:** исследовать и изучить списковые структуры данных, научиться писать программы по исследованию списковых структур на языке программирования.

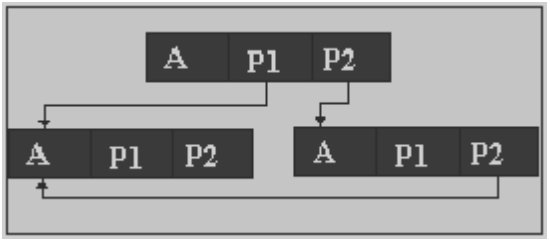
**Краткие теоретические сведения**

Статические программные объекты – это объекты, которые порождаются непосредственно перед выполнением программы, существуют в течение всего времени ее выполнения и размер значений которых не изменяется по ходу выполнения программы.

Программные объекты, которые создаются в процессе выполнения программы, называют динамическими объектами.

Для работы с динамическими объектами предусматривается специальный тип данных - так называемый ссылочный тип. Значением этого типа является ссылка на какой - либо программный объект, по которой осуществляется непосредственный доступ к этому объекту. Для описания действий над динамическими объектами каждому такому объекту в программе сопоставляется статическая переменная ссылочного типа - в терминах этих ссылочных переменных и описываются действия над соответствующими динамическими объектами.

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.

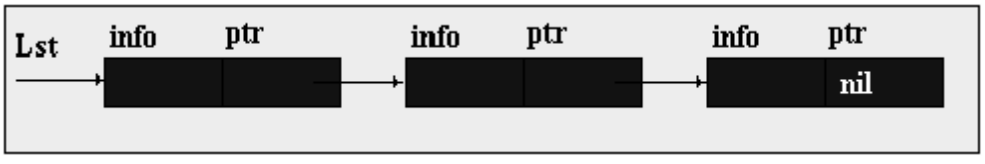
Наиболее распространенными динамическими структурами являются связанные списки. С точки зрения логического представления различают линейные и нелинейные списки. В линейных списках связи строго упорядочены. Указатель предыдущего элемента дает адрес последующего элемента или наоборот.

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.

**Линейные однонаправленные списки**



Под односвязными списками понимают упорядоченную последовательность элементов, каждый из которых имеет 2 поля: информационное поле *info* и поле указателя *ptr* .

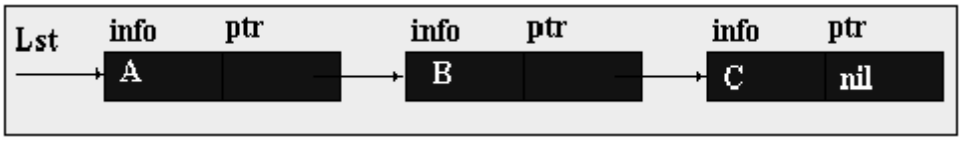
Особенностью указателя является то, что он дает только адрес последующего элемента списка. У однонаправленных списков поле указателя последнего элемента в списке является пустым null.

*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* = null.

**Алгоритм**

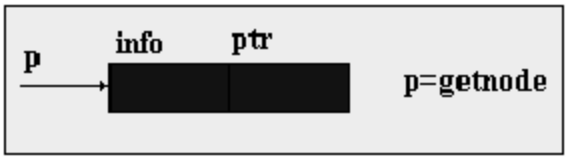
Операции с односвязными списками.

1. Вставка элемента в начало односвязного списка.

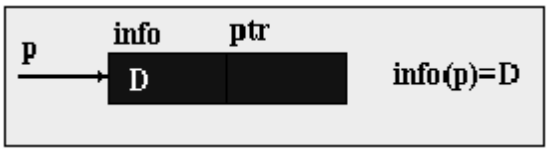



Вставим в начало данного списка элемент, информационное поле которого содержит переменную D. Чтобы это осуществить, необходимо произвести следующие действия:

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



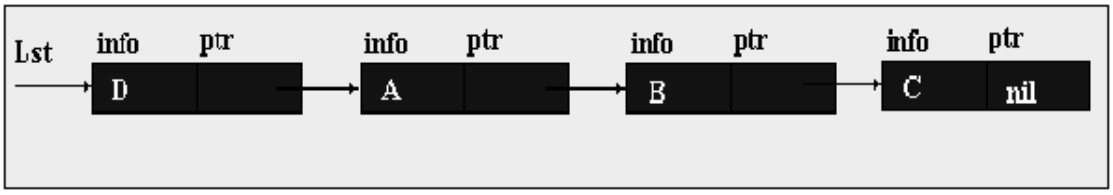

b) Информационному полю созданного элемента присвоить значение D.




c) Связать новый элемент со списком.

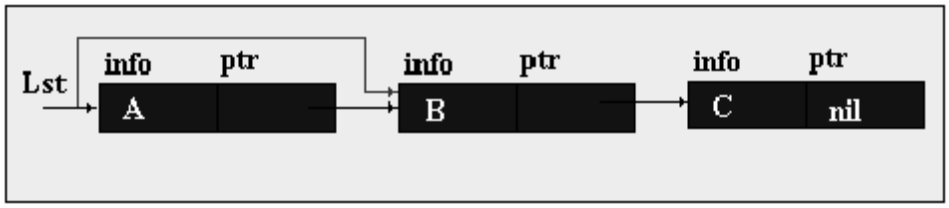
Ptr(p)=lst (lst - уже не указывает на начало списка)

d) Перенести указатель lst на начало списка.




**Удаление элемента из начала односвязного списка**

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




Чтобы это осуществить, необходимо произвести следующие действия :

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

P=lst

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

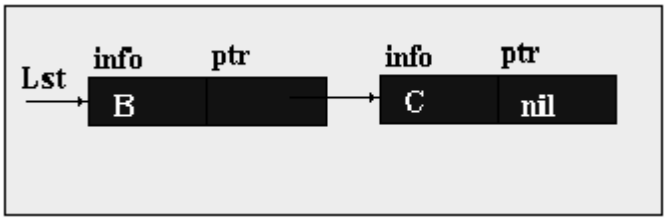
X=info( P )

c) Перенести указатель lst на новое начало списка.

lst=ptr( P )

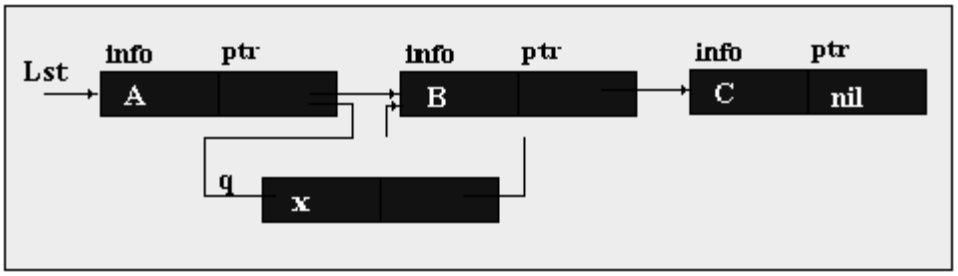
d) Удалить элемент на который указывает указатель p.

Freenode( P )




**Вставка элемента в список**

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




Чтобы это осуществить необходимо произвести следующие действия :

a) Создать пустой элемент на который указывает указатель q.

Q=getnode

b) Внести x в информационное поле созданного элемента.

Info(Q)=x

c) Связать элемент x с элементом B.

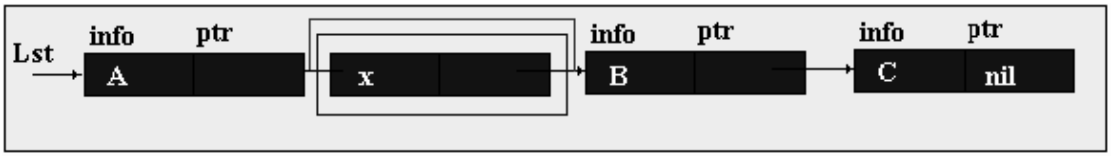
Ptr(Q)=Ptr(P) - это значит, что указателю созданного элемента присваивается значение указателя элемента p.

d) Связать элемент A с элементом x.

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


**Удаление элемента из односвязного списка**

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




Чтобы это осуществить необходимо произвести следующие действия :

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

Q=ptr(p)

b) Поставить за элементом A элемент B.

Ptr(p)=Ptr(Q)

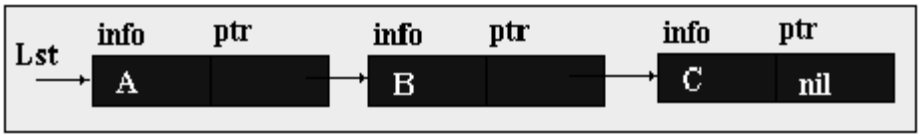
c) Запомнить информацию, которая содержится в поле info удаляемого элемента.

K=info(Q)

d) Удалим элемент с указателем Q.

Freenode(Q)

Окончательно :




**Порядок выполнения работы**

1. изучить краткие теоретические сведения;
2. по заданию, данному преподавателем, разработать алгоритм программы решения задачи;
3. написать программу на языке C++(два варианта: с использованием STL, без использования STL);
4. отладить программу;
5. оформить отчет (в отчете должны быть: тема, цель работы, задание, листинг программы, тестовый пример, результаты работы программы.

**Задания**

1. Написать программу передвижения элемента списка 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.

**Варианты заданий**

1. Список растений
2. Список  футбольных команд
3. Список продуктов питания
4. Список СУБД
5. Список инструментальных средств разработки ПО
6. Список языков программирования
7. Список пород собак
8. Список пород кошек
9. Список алгоритмов
10. Список группы

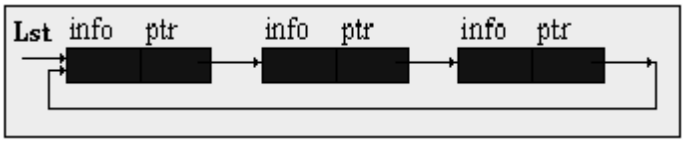
# Лабораторная работа № 2

# КОЛЬЦЕВЫЕ СПИСКИ

**Цель работы:** исследовать и изучить кольцевые списки на примере основных процедур. овладеть навыками написания программ по исследованию основных процедур списковых структур на языке программирования C++, C#.

**Краткие теоретические сведения**

Рассмотрим кольцевые списки.




Как видно на 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.

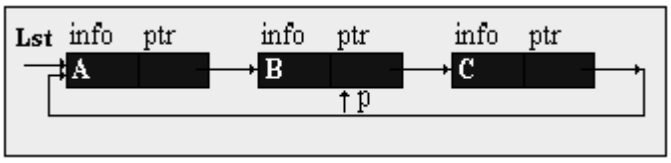
Чтобы закольцевать список необходимо присвоить указателю последнего элемента указатель начала списка (Ptr(p)=lst).

Ptr(p) - указатель последнего элемента;

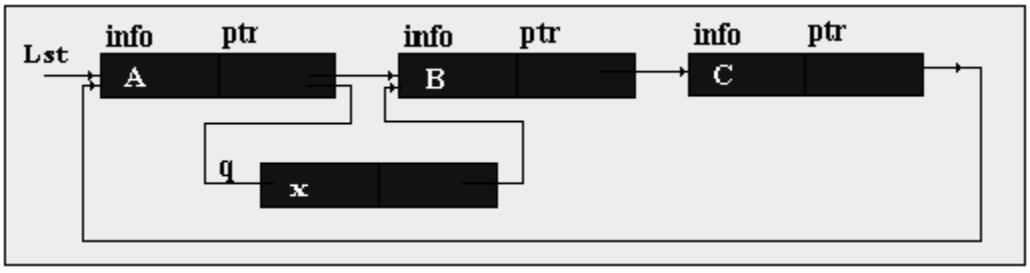
Lst - указатель начала списка.

***Операции с кольцевыми списками:***

**Вставка элемента в кольцевой список**




Чтобы это осуществить необходимо произвести следующие действия:

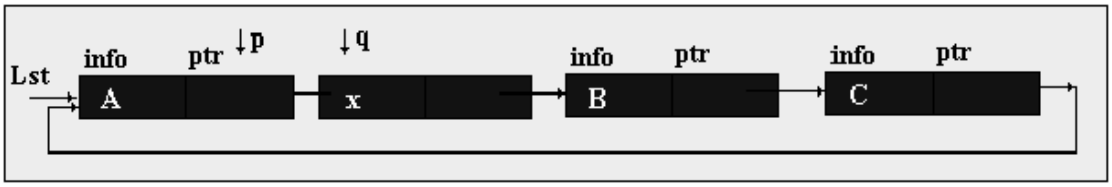



a) Создать пустой элемент на который указывает указатель q
q=getnode

b) Внести x в информационное поле созданного элемента
info(q)=x

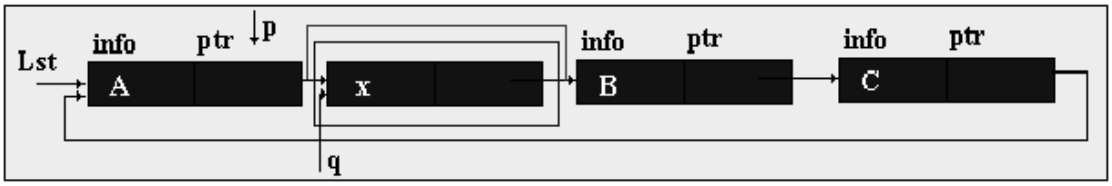
c) Связать элемент X с элементом B
ptr(q)=ptr(p) - это означает, что указателю
созданного элемента присваивается значение указателя элемента p.

d) Связать элемент A с элементом X
ptr(p)=q - это означает, что следующим за элементом
A будет элемент на который указывает указатель q.

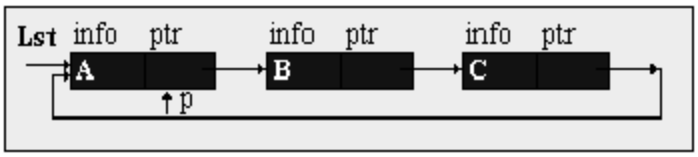



**Удаление элемента из кольцевого списка**

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




Чтобы это осуществить, необходимо произвести следующие действия:
a) Ввести указатель q, который будет указывать на удаляемый элемент.
q=ptr(p)
b) Поставить за элементом A элемент B
ptr(p)=ptr(q)
c) Запомнить информацию, которая содержится в поле info удаляемого элемента.
k=info(q)
d) Удалить элемент с указателем q.
Freenode(q)




**Порядок выполнения работы:**

- изучить краткие теоретические сведения;
- по заданию, данному преподавателем, разработать алгоритм программы решения задачи;
- написать программу на языке C++
- отладить программу;
- оформить отчет (тема, цель работы, листинг программы, контрольный пример).

**Варианты заданий**

1. Дан кольцевой список, содержащий 20 фамилий игроков футбольной команды. Разбить игроков на 2 группы по 10 человек. Во вторую группу попадает каждый 12-й человек.

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

3. Задача Джозефуса.

Отряд солдат окружен превосходящими силами противника. Надежда на победу без подкрепления исключается, однако для прорыва из лагеря имеется одна лошадь. Солдаты решают выбрать одного человека и послать его за помощью. Они становятся в круг и из шляпы выбирается **число *n*** и одно из их **имен.** Производится счет по часовой стрелке **по кругу**, начиная с солдата с выбранным именем. Когда счетчик достигает *n,* соответствующий солдат **удаляется из круга,** а счет продолжается снова, начиная со следующего солдата. **Последний оставшийся** в круге солдат посылается за подмогой. Определите **порядок удаления** солдат из круга и имя оставшегося солдата.

4. Даны 2 кольцевых списка, содержащие фамилии участников лотереи и наименования призов. Выиграет N человек (каждый К-й). Число для пересчета призов - t.

5. Даны 2 списка, содержащих фамилии учащихся и номера экзаменационных билетов. Число пересчета для билетов - Е, для учащихся - К. Определить номера билетов, вытащенных учащимися.

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

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

8. Даны 2 списка, содержащие перечень товаров, производимых Концернами Bosh и Candy. Создать список товаров, выпускаемых как одной так и другой фирмой.

9. Даны 2 списка, содержащие фамилии хоккеистов основного состава команды и запасного. Произвести К замен.

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

# Лабораторная работа № 3

# ОБРАБОТКА СТРОК

**Цель работы:** Рассмотреть представление данных в виде строки. Изучить основные способы обработки строк.

**Краткие теоретические сведения**

Первый тип строк предполагает, что строка определяется как символьный массив, который завершается нулевым символом ( '\0' ). Таким образом, строка с завершающим нулем состоит из символов и конечного нуль-символа. Другой тип представления строк в C++ заключается в применении объектов класса string, который является частью библиотеки классов C++. Таким образом, класс string подразумевает объектно-ориентированный подход к обработке строк. Объявляя символьный массив, предназначенный для хранения строки с завершающим нулем, необходимо учитывать признак ее завершения и задавать длину массива на единицу больше длины самой большой строки из тех, которые предполагается хранить в этом массиве. Например, при объявлении массива str, в который предполагается поместить 10-символьную строку, следует использовать следующую инструкцию: char st r [ 11] ; Заданный здесь размер (11) позволяет зарезервировать место для нулевого символа в конце строки. C++ позволяет определять строковые литералы. Строковый литерал — это список символов, заключенный в двойные кавычки. Например: "Привет" "Мне нравится C++" "Mars" "". Строка, приведенная последней (" "), называется нулевой. Она состоит только из одного нулевого символа (признака завершения строки).

Оператор ">>" прекращает считывание строки, как только встречает символ пробела, табуляции или новой строки. Если в программе необходимо считать строку с клавиатуры, вызовите функцию gets (), а в качестве аргумента передайте имя массива, не указывая индекса. После выполнения этой функции заданный массив будет содержать текст, введенный с клавиатуры. Функция gets () считывает вводимые пользователем символы до тех пор, пока он не нажмет клавишу <Enter>. Для вызова функции gets () в программу необходимо включить заголовок <cstdio>.

Например:

```
# include <iostream>
# include <cstdio>
using namespace std;
int main()
char str[80] ;
cout <<"Введите строку: ";
gets(str);
cout <<"Вот ваша строка: ";
cout <<st r ;
return 0;
}
```

Язык C++ поддерживает функции обработки строк. Например: strcpy(), strcat(), strlen(), strcmp(). Для вызова всех этих функций в программу необходимо включить заголовок <cstring>.

В языке C++ для удобной работы со строками есть класс string, для использования которого необходимо подключить заголовочный файл string.

Строки можно объявлять и одновременно присваивать им значения:

string S1, S2 = "Hello";

К отдельным символам строки можно обращаться по индексу, как к элементам массива или С-строк. Например S[0]- это первый символ строки. Для того, чтобы узнать

длину строки можно использовать метод size() строки. Например, последний символ строки S это S[S.size() - 1].

Строки можно создавать с использованием следующих конструкторов:
string() - конструктор по умолчанию (без параметров) создает пустую строку.
string(string & S) - копия строки S
string(size_t n, char c) - повторение символа с заданное число n раз.
string(size_t c) - строка из одного символа c.
string(string & S, size_t start, size_t len) - строка, содержащая не более, чем len символов данной строки S, начиная с символа номер start.

Вывод строки: cout << S;

Считывание строки: cin >> S;

В этом случае считывается строка, пропуская пробелы и концы строк. Это удобно для того, чтобы разбивать текст на слова, или чтобы читать данные до конца файла при помощи. Можно считывать строки до появления символа конца строки при помощи функции getline. Сам символ конца строки считывается из входного потока, но к строке не добавляется: getline(cin S);

Метод size() возращает длину строки. Метод length(), который также возвращает длину строки. Метод resize(n) изменяет длину строки, новая длина строки становится равна n. Метод clear() - очищает строку. Метод empty() - возвращает true, если строка пуста, false - если непуста. Push_back(c) добавляет в конец строки символ c, вызывается с одним параметром типа char. Append() добавляет в конец строки несколько символов, другую строку или фрагмент другой строки. Имеет много способов вызова. Erase(pos) - удаляет из строки S с символа с индексом pos и до конца строки. Insert() вставляет в середину строки несколько символов, другую строку или фрагмент другой строки. Substr() возвращает подстроку данной строки начиная с символа с индексом pos и до конца строки. Find() ищет в данной строке первое вхождение другой строки str. Возвращается номер первого символа, начиная с которого далее идет подстрока, равная строке str. Если эта строка не найдена, то -1.

**Порядок выполнения работы**

1. Изучить краткие теоретические сведения;
2. Выполнить задание1 и задание 2, на языке C++;
3. Отладить программу;
4. Оформить отчет (тема, цель работы, листинг программы, по каждому заданию привести по 3 контрольных примера).

**Варианты заданий**

**Задание 1.**

1. Дана строка. Подсчитать количество содержащихся в ней цифр.
2. Дана строка. Подсчитать количество содержащихся в ней прописных латинских букв.
3. Дана строка. Подсчитать общее количество содержащихся в ней строчных латинских и русских букв.
4. Дана строка. Преобразовать в ней все прописные латинские буквы в строчные.
5. Дана строка. Преобразовать в ней все строчные буквы (как латинские, так и русские) в прописные.
6. Дана строка. Преобразовать в ней все строчные буквы (как латинские, так и русские) в прописные, а прописные — в строчные.
7. Дана строка. Если она представляет собой запись целого числа, то вывести 1, если вещественного (с дробной частью) — вывести 2; если строку нельзя преобразовать в число, то вывести 0. Считать, что дробная часть вещественного числа отделяется от его целой части десятичной точкой «.».

8. Дано целое положительное число. Вывести символы, изображающие цифры этого числа (в порядке слева направо).
9. Дано целое положительное число. Вывести символы, изображающие цифры этого числа (в порядке справа налево).
10. Дана строка, изображающая целое положительное число. Вывести сумму цифр этого числа.

**Задание 2.**

1. Дано целое число N (> 0) и строка S. Преобразовать строку S в строку длины N следующим образом: если длина строки S больше N, то отбросить первые символы, если длина строки S меньше N, то в ее начало добавить символы «.» (точка).
2. Даны целые положительные числа N1 и N2 и строки S1 и S2. Получить из этих строк новую строку, содержащую первые N1 символов строки S1 и последние N2 символов строки S2 (в указанном порядке).
3. Дан символ C и строка S. Удвоить каждое вхождение символа C в строку S.
4. Дан символ C и строки S, S0. Перед каждым вхождением символа C в строку S вставить строку S0.
5. Дан символ C и строки S, S0. После каждого вхождения символа C в строку S вставить строку S0.
6. Даны строки S и S0. Проверить, содержится ли строка S0 в строке S. Если содержится, то вывести True, если не содержится, то вывести False.
7. Даны строки S и S0. Найти количество вхождений строки S0 в строку S.
8. Даны строки S и S0. Удалить из строки S первую подстроку, совпадающую с S0. Если совпадающих подстрок нет, то вывести строку S без изменений.
9. Даны строки S и S0. Удалить из строки S последнюю подстроку, совпадающую с S0. Если совпадающих подстрок нет, то вывести строку S без изменений.
10. Даны строки S и S0. Удалить из строки S все подстроки, совпадающие с S0. Если совпадающих подстрок нет, то вывести строку S без изменений.

## Лабораторная работа № 4

# СТРУКТУРЫ ДАННЫХ ТИПА ДЕРЕВО

**Цель работы:** Ознакомится с представлением данных в виде деревьев. Изучить основные алгоритмы обхода деревьев.

**Краткие теоретические сведения**

Элементы могут образовывать и более сложную структуру, чем линейный список. Часто данные, подлежащие обработке, образуют иерархическую структуру, подобную изображенной на рис. 1, которую необходимо отобразить в памяти компьютера и, соответственно, описать в структурах данных.

Каждый элемент такой структуры может содержать ссылки на элементы более низкого уровня иерархии, а может быть, и на объект, находящийся на более высоком уровне иерархии.

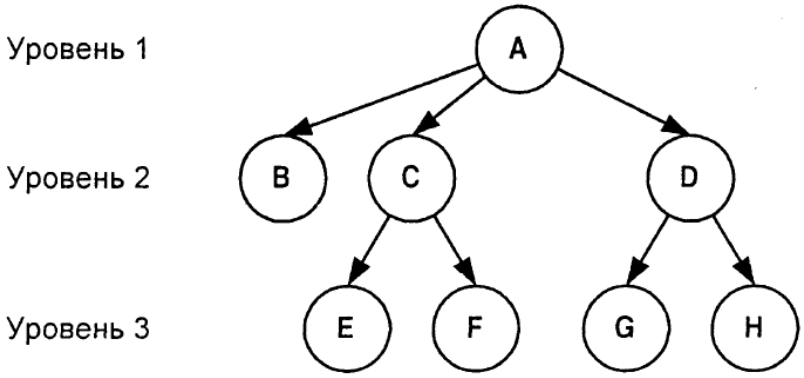



Рис. 1. Структура дерева

Если абстрагироваться от конкретного содержания объектов, то получится математический объект, называемый деревом (точнее, корневым деревом).

Корневым деревом называется множество элементов, в котором выделен один элемент, называемый корнем дерева, а все остальные элементы разбиты на несколько непересекающихся подмножеств, называемых поддеревьями исходного дерева, каждое из которых, в свою очередь, есть дерево.

При представлении в памяти компьютера элементы дерева (узлы) связывают между собой таким образом, чтобы каждый узел (который согласно определению обязательно является корнем некоторого дерева или поддерева) был связан с корнями своих поддеревьев.

Наиболее распространенными способами представления дерева являются следующие три (или их комбинации).

При первом способе каждый узел (кроме корня) содержит указатель на породивший его узел, т. е. на элемент, находящийся на более высоком уровне иерархии. Для корня дерева соответствующий указатель будет пустым. При таком способе представления, имея информацию о местоположении некоторого узла, можно, прослеживая указатели, подниматься на более высокие уровни иерархии.

```
struct node {
        int key;
        node *parent;
}
```

К сожалению, этот способ представления непригоден, если требуется не только «подниматься вверх» по дереву, но и «спускаться вниз», и нет возможности независимо получать информацию о местоположении узлов дерева. Тем не менее, такое

представление дерева иногда используется в алгоритмах, где прохождение узлов всегда осуществляется в восходящем порядке. Преимуществом такого способа представления дерева является то, что в нем используется минимальное количество памяти, причем практически вся она эффективно используется для представления связей.

Второй способ представления применяют, если каждый узел дерева имеет не более двух (в более общем случае — не более A) поддеревьев. Тогда можно включить в представление узла указатели на корни поддеревьев. В этом случае дерево называют двоичным (бинарным), а два поддерева каждого узла называют соответственно левым и правым поддеревьями этого узла.

Разумеется, узел может иметь только одно — левое или правое — поддерево (эти две ситуации в бинарных деревьях обычно считаются различными) или может вообще не иметь поддеревьев (и в этом случае узел называется концевым узлом или листом дерева).

При этом способе представления достаточно иметь ссылку на корень дерева, чтобы получить доступ к любому узлу дерева, спускаясь по указателям, однако память при таком способе представления используется не столь эффективно.

Описание структуры бинарного дерева, содержащего целые числа в узлах, может выглядеть так:

```
sruct node {
        int key;
        node *parent;
        node *left;
        node *right;
}
```

Каждый узел, содержит дополнительно две ссылки на корни левого и правого поддерева (left и right соответственно).

Если бинарное дерево имеет N узлов, то при таком способе представления всегда остаются пустыми более половины указателей. Тем не менее, такое представление является настолько удобным, что потерями памяти обычно пренебрегают.

Третий способ представления состоит в том, что в каждом элементе содержатся два указателя, причем один из них служит для представления списка поддеревьев, а второй для связывания элементов в этот список. Формально описание структуры для этого представления может быть тем же самым, что и для случая бинарного дерева, но смысл содержащихся в этом описании указателей меняется.

При третьем способе представления деревьев часто используется генеалогическая терминология: узлы, содержащиеся в поддеревьях каждого узла, называются его потомками, а корни этих поддеревьев, расположенные на одном уровнеиерархии, называют братьями. Эту терминологию можно отразить в описании структуры, и тогда поля left и right будут называться son и brother соответственно.

```
struct node {
        int key;
        node *parent;
        node *son;
        node *brother;
}
```

Иногда используются и более экзотические способы представления деревьев. Например, в некоторых случаях связи между узлами дерева можно вообще не представлять с помощью указателей, а «вычислять», если узлы дерева образуют регулярную структуру, не меняющуюся или меняющуюся очень мало в процессе работы с деревом.

Рассмотрим пример. Пусть узлы бинарного дерева пронумерованы, начиная с корня и далее вниз по иерархическим уровням.

При этом узлы расположены настолько плотно, что предком узла с номером i всегда является узел с номером i/2. В этом случае узлы дерева можно расположить в массиве, причем местоположение каждого узла будет задано его индексом в массиве, а максимальное число узлов указывается сразу же при создании дерева. Это дерево обладает еще одним свойством: каждый узел дерева содержит целое число, меньшее, чем значения, хранящиеся в поддеревьях этого узла. Таким образом, минимальное число всегда находится в корне дерева. Такое дерево иногда называют «пирамидой».

Высотой бинарного дерева назовем максимальное число узлов, которое может встретиться на пути из корня дерева в некоторый другой узел, при условии, то этот путь проходит только по связанным между собой узлам и никогда не проходит дважды через один и тот же узел.

Будем для удобства считать, что пустой указатель представляет вырожденное пустое» дерево, высота которого равна нулю. В этом случае высота бинарного дерева может быть выражена следующей формулой:

$$h(t) = \begin{cases} 0, npu(t == null) \\ \max(h(t_{left}, h(t_{right})), npu(t != null) \end{cases}$$

Тогда определение функции height, реализующим вычисление высоты дерева, может выгледеть так:

```
int height(node *root){
        int hl = 0;
        int hr = 0;
        if (root->left != NULL)
                hl = height(root->left);
        if (root->right != NULL)
                hr = height(root->right);
        return ((hl > hr)? hl : hr ) + 1; }
```

Обход деревьев

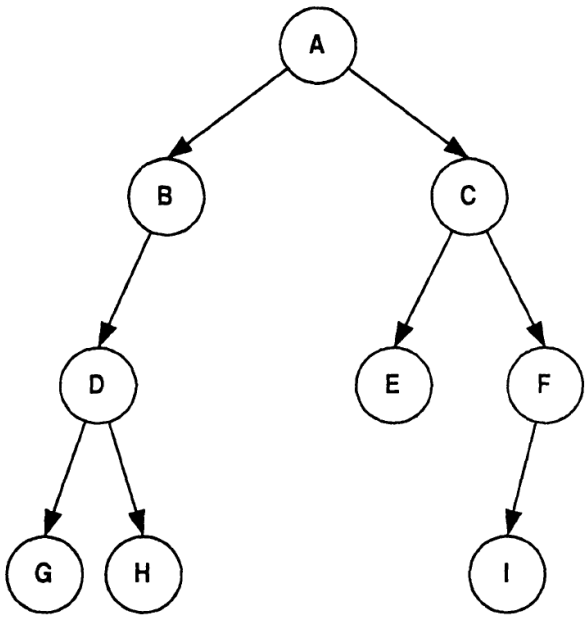



Рис. 2. Пример бинарного дерева

Рассмотрим двоичное дерево, содержащее в узлах символы, например, такое, как на рис. 2. Говорят, что дерево проходится сверху вниз, если каждый узел всегда проходится раньше, чем поддеревья этого узла. Для изображенного дерева следующие порядки прохождения узлов будут порядками обхода «сверху вниз»:

A B C D E F G H I

A B D H G C E F I

A C B F I E D G H

В этом случае также говорят, что дерево обходится в нисходящем порядке. Аналогично, при обходе снизу вверх (восходящий порядок обхода) узел проходится после того, как пройдены его поддеревья, например:

I H G F E D C B A

I F E C G H D B A

Очевидно, что для каждого обхода «сверху вниз» обход тех же узлов в противоположном порядке будет обходом «снизу вверх».

Порядок обхода бинарного дерева называется инфиксным, если каждый узел проходится после того, как пройдены все узлы одного поддерева, и до того, как пройдены узлы другого поддерева. Обычно используют либо левосторонний инфиксный обход, при котором сначала обходится левое поддерево, потом корневой узел, потом правое поддерево, либо правосторонний инфиксный обход, при котором наоборот, сначала обходится правое поддерево, затем корень, и, наконец, левое поддерево.

Вот как, например, располагаются узлы дерева в порядке левостороннего инфиксного обхода:

G D H B A E C I F

Для нисходящих и восходящих обходов обычно также определяются некоторые «стандартные» порядки обхода, например, при нисходящем левостороннем обходе порядок прохождения узлов определен следующим образом: непосредственно после обхода каждого узла проходятся узлы его поддеревьев в порядке слева направо (для бинарного дерева — сначала левого поддерева, затем правого). Для дерева, приведенного на Рис. 2, узлы в порядке нисходящего левостороннего обхода располагаются следующим образом:

A B D G H C E F I

Еще два термина, часто использующихся для определения порядка обхода узлов, — «обход в глубину» (depth first) и «обход в ширину» (breadth first).

Обход является обходом «в ширину», если узлы, расположенные ближе к корню дерева, обходятся раньше, чем узлы, отстоящие дальше от корня. При этом расстояние от узла до корня измеряется количеством ребер на кратчайшем пути из этого узла до корня. Так, например, в дереве на рис. 2.7 узлы D, E и /'находятся на одном и том же расстоянии от корня — 2, а узел B располагается ближе к корню, т. к. расстояние от него до корня равно 1. Обычно полагают, что корень находится на расстоянии 0от себя, так что при обходе «в ширину» он всегда обходится первым.

Обход является обходом «в глубину», если в любом поддереве исходного дерева узлы обходятся «подряд», т. е. если обход некоторого поддерева начат, то он продолжается до тех пор, пока все поддерево не будет обойдено. Например, нисходящий обход «в глубину» может быть представлен такой последовательностью узлов того же дерева:

A B D G H C E F I

Если в этой последовательности узлов расставить скобки для обозначения поддеревьев, то можно увидеть, что действительно, каждое поддерево занимает свой «участок» в этой последовательности, не пересекающийся с участками, занятыми другими поддеревьями:

( A ( B ( D (G) (H) ) ) ( C ( E ( F ( I ) ) ) ) )

Обходы «в глубину» особенно удобно реализовывать в виде рекурсивных процедур, поскольку такая процедура, получив в качестве аргумента некоторое поддерево для обхода, будет обходить его до тех пор, пока все узлы этого поддерева не будут пройдены. Напротив, реализовать обход «в ширину» с помощью рекурсивной процедуры практически невозможно, во всяком случае, такая реализация обхода будет неестественной и неэффективной.

Для примера приведем один способ обхода дерева — левосторонний:

```
void left_walk(node *n){
    if(n == NULL) return;
    left_walk(n->left);
    cout<< n->key;
    left_walk(n->right);
}
```

Обходы «в глубину» особенно удобно реализовывать в виде рекурсивных процедур, поскольку такая процедура, получив в качестве аргумента некоторое поддерево для обхода, будет обходить его до тех пор, пока все узлы этого поддерева не будут пройдены. Напротив, реализовать обход «в ширину» с помощью рекурсивной процедуры практически невозможно, во всяком случае, такая реализация обхода будет неестественной и неэффективной.

Начнем с обхода в глубину. Будем использовать следующий алгоритм. Для того чтобы обойти некоторое поддерево, сначала пройдем корень этого поддерева, затем спустимся в его левое поддерево, а правое поддерево запомним в стеке. Если левое поддерево пусто, то можно сразу спускаться в правое поддерево вместо того, чтобы запоминать его в стеке. Если оба поддерева пусты, то очередное поддерево надо извлечь из стека.

```
void deep_walk(node *root){
    stack = create_stack();
    stack_push(stack, root)
    // Пока в стеке есть узлы
    while((node *n = stack_pop(stack)) != NULL){
        do{
            //Выводим текущую вершину
            cout<< n->key;
            if (n->left != NULL){
                // Помещаем в стек левое поддерево
                if(n->right != NULL)
                    stack_push(stack, n->right);
                n = n->left;
            } else
                N = n->right;
        } while(n!=NULL)
    }
}
```

Данный фрагмент напечатает строку:

ABDGHCEFI

Рассмотрим процедуру для нисходящего обхода «в ширину», использующий очередь для хранения вершин в порядке обхода.

Очередь идеально походит для организации обхода «в ширину». Действительно, по мере продвижения от корня дерева вниз в очередь будут попадать все более удаленные от корня вершины дерева. При этом, чем раньше вершина попала в дерево, тем раньше она будет обработана — это основной принцип хранения информации в структуре очереди. Рассмотрим опять дерево, приведенное на рис. 2.

```
void deep_walk(node *root){
    queue = create_queue();
    enqueue(queue, root);
    while((node *n = dequeue(queue)) != NULL){
        //Ставим в очередь поддеревья этого узла
        if (n->left != NULL) enqueue(queue, n->left);
```

```
                if (n->right != NULL) enqueue(queue, n->right);
cout<< n->key;
        }
}
```

До сих пор мы все время говорили только о бинарных деревьях. В некоторой степени это оправдано, поскольку d деревьях общего вида узлы могут содержать два указателя точно так же, как и в бинарных деревьях. Соответственно, все те способы обхода, которые применимы к бинарным деревьям, могут быть применены и к деревьям общего вида. Тем не менее, смысл, вкладываемый нами в эти обходы, может меняться, поскольку логическая структура дерева может не совпадать с его физической структурой.

Поясним сказанное на примере. Пусть имеется дерево, логическая структура которого показана на рис. 3.

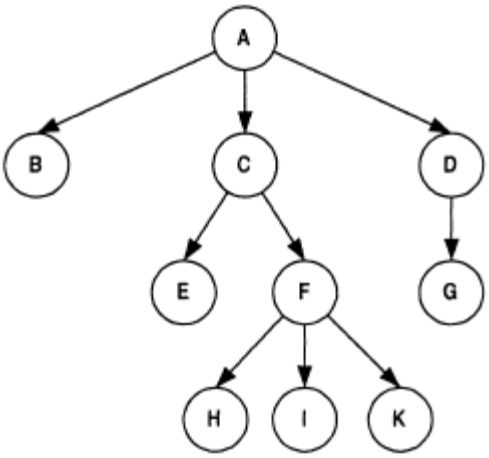



Рис. 3. Пример дерева общего вида для обхода

В памяти связи между узлами будут выглядеть несколько по-другому: каждый узел будет содержать указатели на первого непосредственного потомка («старшего сына») и на соседний узел, находящийся на том же уровне иерархии («брата»), так что физическая структура связей между теми же узлами будет выглядеть так, как на рис. 4 (логические связи, отсутствующие в физической структуре дерева, показаны пунктиром).

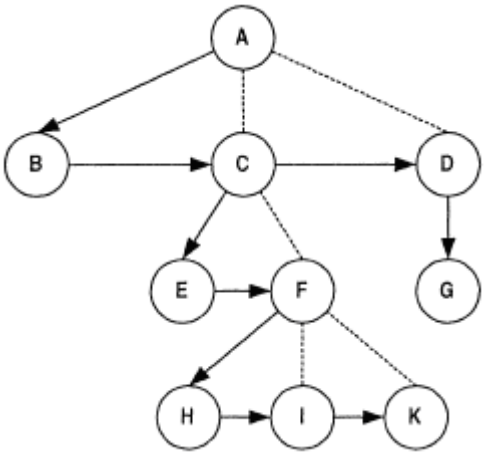



Рис. 4. Представление дерева общего вида с помощью бинарного

Если теперь рассмотреть какой-либо из порядков обхода узлов для исходного дерева, скажем, левосторонний обход «в ширину», то для второго дерева тот же порядок обхода уже не будет обходом «в ширину». Для исходного дерева узлы в порядке левостороннего обхода «в ширину» будут располагаться так:

A, B, C, D, E, F, G, H, I, K,

но для второго дерева узлы в порядке обхода «в ширину» будут расположены уже по-другому:

A, B, C, E, D, F, G, H, I, K.

Конечно, нетрудно написать функцию, которая будет представлять обход «в ширину» узлов исходного дерева, получая в качестве аргумента его физическое представление, однако, реализация этой функции будет, разумеется, отличаться от реализации обхода «в ширину» для бинарного дерева. Тем не менее, для некоторых порядков обхода функции будут одинаковы в обоих случаях. Например, нисходящий левосторонний обход «в глубину» для обоих деревьев будет одним и тем же:

A, B, C, E, F, H, I, K, D, G.

Иногда очень важным оказывается вопрос о количестве дополнительной памяти, требующейся для организации обхода дерева. Так, например, в системах, реализующих язык программирования LISP, практически вся память, участвующая в работе программы, организована в виде деревьев, которые иногда требуется обходить в условиях недостатка памяти (например, для организации «уборки мусора»). Во всех приводимых ранее алгоритмах для выполнения обхода явно или неявно использовалась некоторая достаточно сложно организованная структура памяти — стек или очередь. Такая структура нужна для того, чтобы в процессе обхода помнить, какие элементы дерева еще не пройдены.

Конечно, можно каждый раз искать очередной непройденный элемент, исходя из того, какой узел дерева обходится в настоящий момент, но такой поиск очередного элемента «от корня» приводит к резкому замедлению скорости работы алгоритма. Компромиссом являются варианты, в которых в той или иной степени используются элементы памяти самого дерева. При этом обычно приходится отводить некоторую дополнительную память непосредственно в узлах дерева, так что какие-то потери памяти все равно остаются, однако их можно свести к минимуму: часто достаточно всего одного дополнительного бита памяти в каждом узле дерева для того, чтобы организовать обход дерева без использования дополнительных структур.

Иногда такой дополнительный бит памяти можно получить «даром», если при хранении информации в дереве имеется некоторая избыточность. Например, если информация, хранимая в узле дерева, содержит неотрицательное целое число, то часто можно использовать знаковый разряд этого числа для хранения нужной информации. Можно также задействовать неиспользуемые пустые указатели на несуществующие поддеревья и узлы для хранения дополнительной информации.

Рассмотрим два способа обхода деревьев на основе внутренней информации узлов дерева для организации обхода. В первом способе основная идея состоит в том, чтобы задействовать пустые указатели в исходном дереве для организации перехода от «потомков» к «предкам». Во втором способе указатели трансформируются уже в процессе обхода: при движении по дереву «вниз» пройденные указатели заменяются указателями «наверх», к корню дерева; при движении по ним обратно происходит восстановление направленности указателей. В обоих случаях требуется наличие дополнительного бита, который будет представлен в программах дополнительным полем типа bool в структуре узла дерева.

Сначала рассмотрим обход «сверху вниз» для дерева произвольной структуры (вообще говоря, не бинарного), представленного так, как показано на рис. 4.

Будем считать, что в узлах, представляющих самого младшего «брата», содержится ссылка на «родительский» узел. Для этого можно использовать свободную ссылку на «брата», а чтобы различить эти два способа применения

ссылки, введем дополнительный признак в каждый узел. Получившееся представление дерева изображено на рис. 5. На нем «младшие братья» выделены штриховкой, а ссылки на родительские узлы представлены стрелками иного вида, чем основные ссылки.

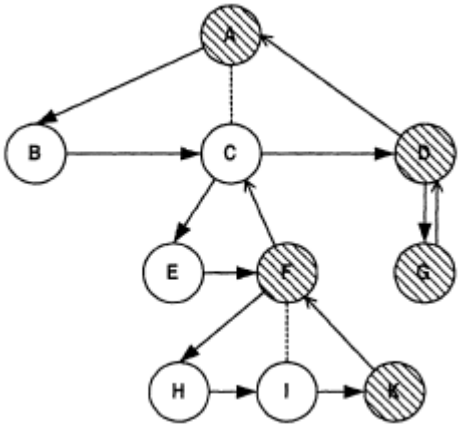



Рис. 5. Представление дерева с "обратными" ссылками

В этой процедуре переход к следующему элементу в порядке обхода осуществляется предельно просто: если у узла имеется «сын», то происходит переход к этому «сыну»; если имеется «брат», то выполняется переход к «брату», наконец, если нет и «брата», то происходит поиск «брата» у ближайшего предка. Обратите внимание, что поиск следующего узла осуществляется не в тот момент, когда этот узел становится действительно нужен, а «заранее», т. е. в тот момент, когда запрашивается предыдущий узел. Это дает возможность легко определить конец итерации.

```
walk_tree(node *root){
node *current = root;
while (current != NULL){
        cout<<current->key; //Выдача результата
// теперь ищем следующий элемент
if (current->son != NULL)
current = current->son;
// Далее youngest признак того, что указатель
// ссылается на отца. Изначально все FALSE
while (current != NULL && current->youngest) {
current = current->brother; // переход вверх
}
if (current != NULL)
current = current->brother;
}
}
```

Если дерево, построено правильно, то его узлы будут пройдены в следующем порядке:

A, B, C, E, F, H, I, K, D, G,

что для этого дерева является естественным порядком обхода «сверху вниз».

Недостатком такого способа обхода является то, что необходимо аккуратно поддерживать правильную структуру ссылок в дереве. Если, например, некоторый узел удаляется, то необходимо проверить, не содержит ли удаляемый узел ссылку вверх по дереву на предка (то есть не является ли он «самым младшим братом»). Если это действительно так, то необходимо найти его ближайшего «старшего брата» (разумеется, с использованием той же самой ссылки вверх по дереву), и модифицировать представление этого «брата». Аналогично, необходимо аккуратно следить за корректностью представления и при добавлении узлов.

Второй способ обхода с использованием внутренней структуры дерева, свободен от этого недостатка. Нет необходимости как-то по-особому обрабатывать или представлять

узлы дерева при этом способе обхода. Все, что необходимо иметь, — это свободный бит информации для использования его уже во время обхода.

Недостаток этого способа обхода в другом: структура дерева динамически меняется во время его обхода.

На рис. 6 представлен промежуточный этап при обходе бинарного дерева, изображенного на рис. 2. При работе алгоритма используются следующие указатели:

processed — указатель на последнюю обработанную вершину, содержащую указатель «вверх» по дереву;

current — указатель на первую вершину еще не обработанной части дерева.

При движении вниз по дереву (от корня к листьям) некоторые указатели в дереве используются для того, чтобы запомнить обратный путь. При движении обратно значения указателей восстанавливаются. Дополнительный бит информации служитдля того, чтобы запомнить, какой из двух указателей — левый или правый — используется для временного сохранения ссылки на родительский узел.

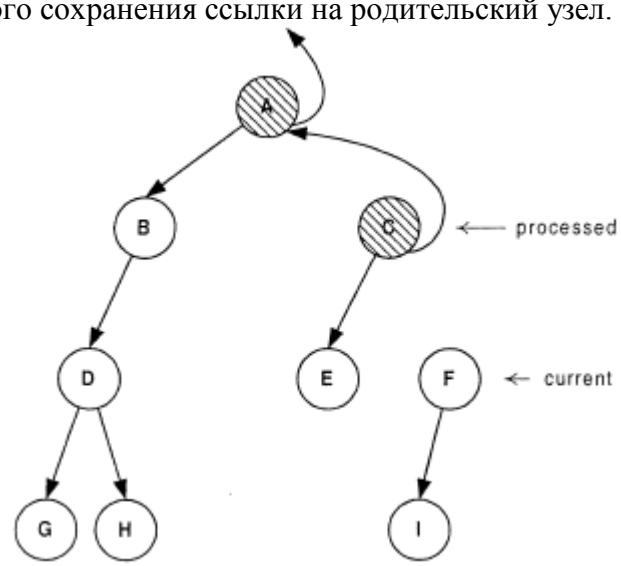



Рис. 6. Обход дерева с использованием внутренних указателей

На рис. 6 изображена ситуация, когда в двух узлах дерева - A и C - поля, в которых в нормальном состоянии записаны указатели на правое поддерево, используются для хранения ссылок на родительский узел.

```
public class Tree {
* Определение узла включает дополнительный бит flag
s t a t i c class Node {
public Object item; // содержимое узла
public boolean flag = false; // флажок для обхода
public Node left = null; // указатель на левое поддерево
public Node right = null; // указатель на правое поддерево
/ / Конструктор узла
public Node(Object item) {
this, item = item;
}
Node root = null; // корень дерева
void traverseWithlnversion (Visitor visitor) {
node processed = null; // указатель вверх по дереву
node current = root; // указатель на текущую вершину
bool down = true; // направление движения
// Цикл обхода узлов закончится,
// когда при движении вверх
// окажется, что уже все узлы пройдены
while (down || processed != NULL) {
```

```
if (down) {
if (current == null) {
// Меняем направление движения
down = false;
} else {
// Спускаемся вниз по дереву на один шаг
node *W = current->left;
current->left = processed;
processed = current;
current = w;
}
} else {
if (processed.flag) {
// Восстанавливаем указатель и
//продвигаемся вверх по дереву
Processed->flag = false;
node *w = processed->right;
processed->right = current;
current = processed;
processed = w;
} else {
// Посещаем вершину при переходе
// из левого поддерева в правое
printf("%c",processed->key);
// Переходим к обработке правого поддерева
node *W = processed->right;
processed->flag = true;
processed->right = processed->left;
processed->left = current;
current = w;
// Снова двигаемся вниз
down = true;
}
}
}
```

**Порядок выполнения работы**

1. Ознакомиться с теоретическими сведениями.
2. Получить вариант задания у преподавателя.
3. Выполнить задание: написать программу на языке C++.
4. Продемонстрировать выполнение работы преподавателю.
5. Оформить отчет.
6. Защитить лабораторную работу.

**Требования к оформлению отчета**

Отчет по лабораторной работе должен содержать следующие разделы:
титульный лист;
цель работы;
задание на лабораторную работу;
ход работы;
ответы на контрольные вопросы;
выводы по проделанной работе.

**Варианты заданий**

1. Реализуйте на языке программирования C++ дерево целых чисел, добавление и удаление листьев. Создайте процедуру обхода дерева «в глубину».
2. Реализуйте на языке программирования C++ дерево целых чисел, добавление и удаление листьев. Создайте процедуру обхода дерева «в ширину».
3. Реализуйте на языке программирования C++ дерево целых чисел, добавление и удаление листьев. Создайте процедуру поиска значения.
4. Создайте процедуру построения бинарного дерева на основе не бинарного. Язык программирования C++.
5. Реализуйте на языке программирования C++ дерево целых чисел, добавление и удаление листьев. Создайте процедуру поиска значения.
6. Реализуйте на языке программирования C++ дерево целых чисел, добавление и удаление листьев. Создайте процедуру обхода дерева «в ширину».
7. Реализуйте на языке программирования C++ дерево целых чисел, добавление и удаление листьев. Создайте процедуру обхода дерева «в глубину».
8. Реализуйте на языке программирования C++ дерево целых чисел, добавление и удаление листьев. Создайте процедуру обхода дерева «в ширину».
9. Реализуйте на языке программирования C++ дерево целых чисел, добавление и удаление листьев. Создайте процедуру поиска значения.
10. Создайте процедуру построения бинарного дерева на основе не бинарного. Реализуйте на языке программирования C++.

**Контрольные вопросы**

1. Что такое дерево?
2. Какие есть способы задания бинарных деревьев?
3. Какие есть способы задания не бинарных деревьев?
4. Какие виды обхода вы знаете? В чем разница между обходом «сверху вниз» и «снизу вверх»?
5. Опишите основные принципы обхода «в глубину».
6. Опишите основные принципы обхода «в ширину».

# Лабораторная работа № 5

# СТРУКТУРА ДАННЫХ МНОЖЕСТВО

**Цель работы:** Ознакомится с представлением данных в виде множеств. Изучить основные алгоритмы работы со множествами..

**Краткие теоретические сведения**

**Множество –** это составной объект, который содержит компоненты одного и того же класса. Над множеством, в отличие от массива, определены совсем другие операции. Во множестве порядок элементов несуществен. Основная операция – проверка принадлежности элемента множеству. Другие операции – объединение, пересечение, добавление элемента, мощность множества. Эффективная реализация множества обычно подразумевает такое его представление, при котором элементы множества не хранятся, вместо этого хранится информация о том, содержится ли элемент во множестве. Это возможно, когда элементы множества достаточно просты, а максимальное число элементов множества не велико. Тогда можно присутствие каждого из возможных элементов множества кодировать одним битом. Например, 1 – элемент принадлежит множеству, 0 – не принадлежит. Наиболее часто используется множество целых чисел из некоторого диапазона и множество символов из некоторого набора символов. Для представления множества можно сформировать битовую шкалу, каждый элемент которой определяет, содержится ли во множестве некоторый элемент. Такая битовая шкала будет содержать столько битов, сколько элементов содержится в универсальном множестве для данного проекта или задачи.

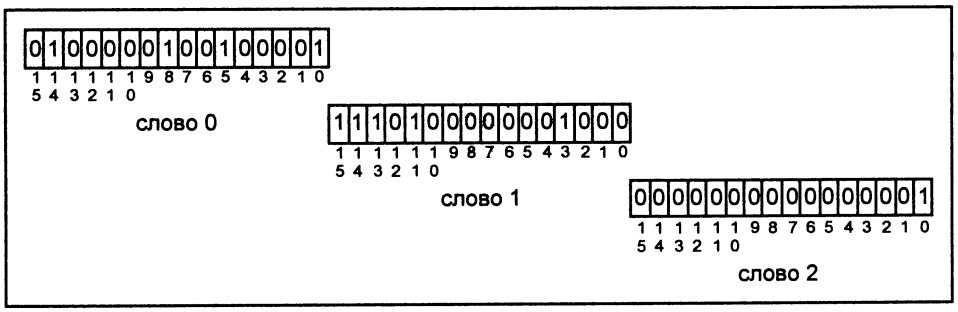



Рис 1. Представление множества в ввиде битовой шкалы.

Листинг 1. Определение класса Set/

```
//---------- файл set.h ----------------------------------------
class Set {

  int minElem;     // минимальный элемент диапазона
  int maxElem;     // максимальный элемент диапазона
  WORD *elems;     // битовая шкала
  int numWords;    // длина шкалы (в словах)

  friend const Set & operator | (const Set & s1, const Set & s2);
  friend const Set & operator & (const Set & s1, const Set & s2);
  friend const Set & operator - (const Set & s1, const Set & s2);
  friend const Set & operator - (const Set & s);

public:

  // Конструктор
  Set(int min = 0, int max = 255);
```

```
  // Конструктор копирования
  Set(const Set & s);

  // Деструктор
  ~Set() { delete[] elems; }

  // Проверка принадлежности элемента множеству
  bool has(int n) const;

  // Добавление элемента в множество
  Set & operator |= (int n);

  // Добавление множества в множество (объединение)
  Set & operator |= (const Set & other);

  // Пересечение множества с множеством
  Set & operator &= (const Set & other);

  // Удаление элемента из множества
  Set & operator -= (int n);

  // Удаление множества из множества (вычитание множеств)
  Set & operator -= (const Set & other);

  // Обращение (нахождение дополнения) множества
  Set & inverse();
};
//---------- файл set.cpp ---------------------------------
#include <stdexcept>
#include "set.h"

// Конструктор
Set::Set(int min, int max) {
  // обеспечим min < max
  if (min > max) {
    minElem = max;
    maxElem = min;
  } else {
    minElem = min;
    maxElem = max;
  }
  int num = maxElem - minElem + 1;      // количество битов
  numWords = (num + 15) >> 4;           // количество слов
  elems = new WORD[numWords];

  // Инициализация пустого множества
  for (int i = 0; i < numWords; i++) elems[i] = 0;
}

// Конструктор копирования
Set::Set(const Set & s) {
  minElem = s.minElem;
  maxElem = s.maxElem;
  elems = new WORD[numWords = s.numWords];
  for (int i = 0;  i < numWords;  i++) {
    elems[i] = s.elems[i];
  }
}
```

```
  // Проверка принадлежности элемента множеству
bool Set::has(int n) const {
  if (n > maxElem || n < minElem)
    return false;    // элемент находится за пределами границ множества
  int bit = n - minElem;
  return (elems[bit >> 4] & (1 << (bit & 15))) != 0;
}


  // Добавление элемента в множество
Set & Set::operator |= (int n) {
  if (n <= maxElem && n >= minElem) {
    int bit = n - minElem;
    elems[bit >> 4] |= (1 << (bit & 15));
  } else {
    throw out_of_range("Cannot add an element: it is out of range");
  }
  return *this;
}

  // Добавление элементов другого множества в данное (объединение)
Set & Set::operator |= (const Set & other) {
  if (other.minElem != minElem || other.maxElem != maxElem) {
    throw out_of_range("Sets incomparable");
  }
  for (int i = 0;  i < numWords;  i++) {
    elems[i] |= other.elems[i];
  }
  return *this;
}
// Пересечение множества с другим множеством
Set & Set::operator &= (const Set & other) {
  if (other.minElem != minElem || other.maxElem != maxElem) {
    throw out_of_range("Sets incomparable");
  }
  for (int i = 0;  i < numWords;  i++) {
    elems[i] &= other.elems[i];
  }
  return *this;
}

// Удаление элемента из множества
Set & Set::operator -= (int n) {
  if (n <= maxElem && n >= minElem) {
    int bit = n - minElem;
    elems[bit >> 4] &= ~(1 << (bit & 15));
  }
  return *this;
}

// Удаление элементов другого множества из данного (вычитание)
Set & Set::operator -= (const Set & other) {
  if (other.minElem != minElem || other.maxElem != maxElem) {
    throw out_of_range("Sets incomparable");
  }
  for (int i = 0;  i < numWords;  i++) {
    elems[i] &= ~other.elems[i];
  }
  return *this;
}
```

```
// Обращение (нахождение дополнения) множества
Set & Set::inverse() {
  for (int i = 0;  i < numWords;  i++) {
    elems[i] = ~elems[i];
  }
  return *this;
}

// Объединение множеств
const Set & operator | (const Set & s1, const Set & s2) {
  return Set(s1) |= s2;
}

// Пересечение множеств
const Set & operator & (const Set & s1, const Set & s2) {
  return Set(s1) &= s2;
}

// Вычитание множеств
const Set & operator - (const Set & s1, const Set & s2) {
  return Set(s1) -= s2;
}

// Дополнение множества до универсального
const Set & operator - (const Set & s) {
  return Set(s).inverse();
}
```

**Порядок выполнения работы**

1. Ознакомиться с теоретическими сведениями.
2. Получить вариант задания у преподавателя.
3. Выполнить задание: написать программу на языке C++.
4. Продемонстрировать выполнение работы преподавателю.
5. Оформить отчет.
6. Защитить лабораторную работу.

**Требования к оформлению отчета**

Отчет по лабораторной работе должен содержать следующие разделы:
титульный лист;
цель работы;
задание на лабораторную работу;
ответы на контрольные вопросы;
выводы по проделанной работе.

**Варианты заданий**

Реализуйте класс множество. Реализуйте все необходимые методы для варианта задания.

1. Известны сорта роз, выращиваемых тремя цветоводами: «Анжелика», «Виктория», «Гагарин», «Ave Maria», «Катарина», «Юбилейная». Определить те сорта, которые имеются у каждого из цветоводов, которые есть хотя бы у одного из цветоводов, которых нет ни у одного из цветоводов.
2. Заданы имена девочек. Определить, какие из этих имен встречаются во всех классах данной параллели, которые есть только в некоторых классах и какие из этих имен не встречаются ни в одном классе.

3. Имеется список класса (все имена различны). Определить, есть ли в классе человек, который побывал в гостях у всех. (Для каждого ученика составить множество побывавших у него в гостях друзей, сам ученик в это множество не входит.)
4. Имеется множество, содержащее натуральные числа из некоторого диапазона. Сформировать два множества, первое из которых содержит все простые числа из данного множества, а второе — все составные.
5. Известны марки машин, изготовленные в данной стране и импортируемые за рубеж. Даны некоторые N стран. Определить для каждой из марок, какие из них были:
   - доставлены во все страны;
   - доставлены в некоторые из стран;
   - не доставлены ни в одну страну.
6. В озере водится несколько видов рыб. Три рыбака поймали рыб, представляющих некоторые из имеющихся видов. Определить:
   - какие виды рыб есть у каждого рыбака;
   - какие рыбы есть в озере, но нет ни у одного из рыбаков.
7. Составить программу, которая вычисляет сумму тех элементов двумерного массива, номера строк и столбцов которых приуадлежат соответственно непустым множествам $S_1$ и $S_2$.
8. Задано некоторое множество M и множество T того же типа. Подсчитать, сколько элементов из множеств T и M совпадает.
9. Из диапазона целых чисел m ... n выделить:
   - множество чисел, делящихся без остатка или на k;
   - множество чисел, делящихся на k-1 без остатка.
10. Дан текст из цифр и строчных латинских букв, за которыми следует точка. Определить, каких букв — гласных или согласных — больше в этом тексте.

**Контрольные вопросы**

1. Что такое множество?
2. Какие есть способы задания множества?
3. Какие основные операции над множествами Вы знаете? Приведите пример реализации одной из них.

# Лабораторная работа № 6

# КРИПТОГРАФИЯ

**Цель работы:** Ознакомиться с методами криптографии.

**Краткие теоретические сведения**

Криптография — это исследование методов безопасной передачи информации, которая потенциально может быть перехвачена злоумышленниками. На ранних этапах развития криптография представляла собой обычную письменность; она работала по той простой причине, что немногие тогда, на заре цивилизации, умели читать. Позже в обиход вошли специальные алфавиты, известные только отправителю и получателю. В одном из самых ранних образцов этого вида криптографии использовались нестандартные иероглифы, высеченные в гробницах (Египет, 1900 г. до н. э.).

Задача шифрования — сделать так, чтобы отправитель мог послать сообщение получателю, а третья сторона, которую обычно зовут злоумышленником или перехватчиком, не могла данное сообщение понять. Предполагается, что злоумышленник в любом случае перехватит данные, поэтому помешать ему их прочитать может только шифрование.

Расшифрованное сообщение называют открытым текстом, а зашифрованное — криптограммой. Процесс превращения открытого текста в криптограмму именуется шифрованием, или кодированием. Восстановление открытого текста из криптограммы называется дешифрованием, или декодированием.

Формально шифр — пара алгоритмов для шифрования и дешифрования сообщений. Криптоанализ — это исследование злоумышленником методов взлома шифрования.

Перестановочные шифры. В перестановочном шифре буквы открытого текста меняются местами каким-то определенным образом, результатом чего является криптограмма. Чтобы прочитать сообщение, получатель возвращает буквы на их исходные позиции. Такие шифры частично полагаются на безопасность через неясность, если злоумышленник не знает, какой именно метод перестановки используется. Большинство из этих методик, помимо прочего, предоставляют ключ, который несет в себе определенную информацию о перестановке. Например, перестановочный шифр по строкам/столбцам, описанный в следующем подразделе, использует в качестве ключа количество столбцов. Однако такие ключи обычно допускают достаточно ограниченное число значений, поэтому их несложно подобрать и взломать тем самым шифрование, особенно если использовать компьютер. С такими шифрами довольно легко работать с помощью карандаша и бумаги - они могут играть роль занятных упражнений (если они кажутся вам слишком простыми, попробуйте проводить вычисления в голове).

Маршрутные шифры. В маршрутном шифре открытый текст записывается в массив (или упорядочивается каким-то другим способом) и затем считывается из него в порядке, который определяется конкретным маршрутом.

Шифры подстановки. В шифрах подстановки буквы открытого текста заменяются другими.

Схема одноразовых блокнотов — это некая разновидность шифра Виженера, в которой ключ имеет ту же длину, что и сообщение. Сдвиг присваивается каждой букве отдельно, поэтому для его поиска нельзя использовать частотный анализ криптограммы.

Блочные шифры. В блочном шифре сообщение разбивается на блоки, которые затем отдельно шифруются и объединяются в криптограмму. Во многих блочных шифрах кодирование выполняется путем многократной трансформации данных. Трансформация должна быть обратимой, чтобы позже криптограмму можно было расшифровать. Разбиение текста на одинаковые блоки позволяет предусмотреть изменения фрагментов определенной длины. Полезное свойство блочных шифров заключается в том, что они

позволяют криптографическому программному обеспечению работать с относительно небольшими фрагментами текста. Представьте, например, что вам нужно зашифровать очень длинное сообщение размером, скажем, в несколько гигабайтов. Если использовать шифр с перестановкой по столбцам, программе придется переходить в разные участки сообщения в памяти. Это может привести к сбрасыванию данных в файл подкачки, что существенно замедлит работу. Блочный шифр ведет себя иначе. Он может анализировать сообщение по частям, которые легко помещаются в памяти. Даже если программе придется использовать файл подкачки, каждый фрагмент сообщения будет загружаться в память всего один раз.

Шифр Фейстеля. В шифре Фейстеля, названном в честь криптографа Хорста Фейстеля, сообщение разбивается пополам на левую и правую части — $L_0$ и $R_0$. К правой части применяется функция, и полученный результат совмещается с левой частью с помощью операции XOR. Затем эти два фрагмента меняются местами и процесс повторяется определенное количество раундов.

Шифрование с открытым ключом и RSA. В шифровании с открытым ключом используется два отдельных ключа — открытый и закрытый. Открытый ключ обычно публикуется, поэтому его может узнать кто угодно (включая злоумышленника). Закрытый ключ известен только получателю. Отправитель использует открытый ключ для шифрования сообщения и передает результат получателю. Однако расшифровать его можно только с помощью закрытого ключа.

В других типах шифрования для кодирования и декодирования сообщения используется один и тот же ключ, поэтому их называют симметричными. Одним из самых известных алгоритмов шифрования с открытым ключом является RSA, названный в честь тех, кто впервые его описал: Рональда Ривеста, Ади Шамира и Леонарда Адлемана.

Цифровая подпись — это криптографический инструмент, чем-то напоминающий криптографическое хеширование. Если вы хотите доказать, что какой-то конкретный документ написан вами, его нужно подписать. Позже можно будет проверить подлинность вашей подписи. Если кто-то другой изменит документ, он не сможет подписать его от вашего имени.

Обычно система цифровых подписей состоит из трех частей:

_ алгоритм генерирования закрытых и открытых ключей;

_ алгоритм, который подписывает документ с помощью закрытого ключа;

_ алгоритм, проверяющий подлинность вашей подписи с помощью открытого ключа.

Цифровая подпись в некотором смысле является противоположной по отношению к системе шифрования с закрытым ключом. Сообщение может быть зашифровано с помощью открытого ключа любым числом отправителей, но расшифровать его сумеет только один из получателей, обладающий закрытым ключом. В системе цифровых подписей сообщение подписывается отправителем с помощью закрытого ключа, а открытый ключ, позволяющий проверить его подлинность, доступен любому количеству получателей.

**Порядок выполнения работы**

1. Ознакомиться с теоретическими сведениями.
2. Получить вариант задания у преподавателя.
3. Выполнить задание: написать программу на языке C++.
4. Продемонстрировать выполнение работы преподавателю.
5. Оформить отчет.
6. Защитить лабораторную работу.

**Требования к оформлению отчета**

Отчет по лабораторной работе должен содержать следующие разделы:

титульный лист;
цель работы;
задание на лабораторную работу;
ход работы;
ответы на контрольные вопросы (или ответить устно);
выводы по проделанной работе.

**Варианты заданий**

1. Напишите программу, которая шифрует и расшифровывает сообщения с помощью перестановочного шифра по строкам/столбцам.
2. Напишите программу, которая шифрует и расшифровывает сообщения с помощью перестановочного шифра по столбцам.
3. Напишите программу, которая подобна шифру перестановки столбцов, что меняет местами не только столбцы, но и строки.
4. Напишите программу, которая использует шифр Виженера для шифрования и расшифровывания сообщений. Расшифруйте с ее помощью криптограмму VDOKR RVVZK OTUII MNUUV RGFQK TOGNX VHOPG RPEVW VZYYO WKMOC ZMBR, применив ключ VIGENERE.
5. Напишите программу, которая использует псевдослучайный одноразовый блокнот. При запуске она должна генерировать блокнот с помощью генератора псевдослучайных чисел (не путать с генератором случайных чисел). Программа должна следить за тем, какие символы используются при шифровании и расшифровывании сообщений.
6. Напишите программу, что выводит частоту, с которой буквы встречаются в сообщении. Отсортируйте результат по количеству вхождений и выведите для каждой буквы сдвиг относительно E.
7. Напишите программу для обмена сообщениями с другим человеком с помощью большого одноразового блокнота.
8. Напишите программу, которая шифрует и расшифровывает сообщения с помощью блочного шифра.
9. Напишите программу, которая шифрует и расшифровывает сообщения с помощью маршрутного шифра.
10. Напишите программу, которая шифрует и расшифровывает сообщения с помощью перестановочного шифра по строкам/столбцам.

**Контрольные вопросы**

1. Что такое криптография?
2. Что такое блочный шифр?
3. Что такое подстановочные шифры, перестановочные шифры?
4. Что такое шифр Цезаря?
5. Что такое маршрутный шифр?
6. Что такое шифр Виженера?
7. Что такое одноразовый блокнот?
8. Что такое алгоритм RSA?
9. Области применения криптографии?

# Лабораторная работа № 7

# РЕКУРСИВНЫЕ АЛГОРИТМЫ

**Цель работы:** Ознакомиться с рекурсивными алгоритмами.

**Краткие теоретические сведения**

В ходе *рекурсии* метод вызывает сам себя. Если он делает это непосредственно, то рекурсия называется *прямой*, если через другой метод — *косвенной*. Она также может быть *единичной* (однократный вызов) либо *множественной* (вызов осуществляется несколько раз). На первый взгляд понятие кажется несколько сложным, поскольку человек не мыслит рекурсивно. Тем не менее есть задачи, которые рекурсивны по своей природе, а их структура

и решение легко отслеживаются с помощью алгоритма. Таковы, например, программы, выстраивающие деревья и проводящие по ним поиск. Однако не всегда это наилучший способ решения задач, в некоторых случаях он снижает производительность программы, в связи с чем необходимо рассмотреть еще и способы удаления рекурсии.

В следующих подразделах приводятся три рекурсивных алгоритма для расчета факториала, чисел Фибоначчи, а также решения задачи о ханойской башне. Они относительно просты, но раскрывают важные понятия. Некоторые алгоритмы используют рекурсию, чтобы создавать сложные графические изображения. И хотя код у них очень компактный, на самом деле они ничуть не проще базовых алгоритмов.

Кривая Коха — хороший пример *самоподобного фрактала*, где часть кривой напоминает ее общую форму. Построение таких фракталов начинается с *инициатора* — участка, определяющего основные очертания будущей фигуры. В ходе рекурсии все инициаторы или некоторые из них преобразуются (правильно масштабируются, поворачиваются и т. д.), превращаясь в своего рода *генераторы*, которые затем замещаются новыми версиями генераторов. Простейшая кривая Коха в качестве инициатора использует линию, которая на каждом уровне рекурсии разбивается на четыре части. Из них формируются три сегмента длиной в 1/3 от исходного отрезка. Первая и четвертая части находятся в направлении базовой линии, вторая поворачивается на –60°, третья — на 120° (рис. 9.4). На следующем уровне рекурсии программа заменяет каждый сегмент в генераторе новой копией генератора. Взглянув на рисунок 9.5, вы поймете, почему кривая называется самоподобной: ее часть выглядит как уменьшенная копия целой фигуры. Пусть pt1, pt2, pt3, pt4 и pt5 — точки, связанные с сегментами в генераторе. Начертим кривую Коха с помощью приведенного ниже псевдокода.

```
// Чертим кривую Коха заданной сложности, начиная с точки p1
// и прокладывая расстояние length в направлении angle.
DrawKoch(Integer: depth, Point: pt1, Float: angle, Float: length)
If (depth == 0) Then
<Чертим сегмент.>
Else
<Находим точки pt2, pt3, and pt4.>
// Рекурсивно чертим части кривой.
DrawKoch(depth - 1, pt1, angle, length / 3);
DrawKoch(depth - 1, pt2, angle - 60, length / 3);
DrawKoch(depth - 1, pt3, angle + 60, length / 3);
DrawKoch(depth - 1, pt4, angle, length / 3);
End If
End DrawKoch
```

Если depth = 0, то алгоритм чертит сегмент от точки p1 длиной length, следуя в направлении angle. (То как осуществляется рисование, зависит от используемой вами среды программирования.) При depth > 0 алгоритм определяет точки pt2, pt3 и pt4, а затем

выстраивает четыре сегмента длиною 1/3 от исходного отрезка: вначале он следует из точки pt1 в направлении angle до точки pt2, потом разворачивается на 60° влево и идет до точки pt3, совершает еще один поворот на 120° вправо (на угол, превышающий исходный на 60°) и двигается до точки pt4, и наконец, из нее, придерживаясь начального угла, преодолевает последнюю часть пути. Если глубина больше 0, то алгоритм рекурсивно вызывает себя четыре раза. Предположим, что $T(N)$ — количество шагов, которые он совершает для глубины n, тогда $T(N) = 4 \_ T(N - 1) + C$, где $C$ — константа. Если проигнорировать последнюю, то $T(N) = 4 \_ T(N - 1)$, отсюда время работы — $O(4N)$. Максимальная глубина рекурсии, необходимая для построения кривой Коха сложностью $N$, определяется только $N$ и не должна вызывать проблем, ведь, как и в предыдущих алгоритмах (для нахождения чисел Фибоначчи и решения головоломки «Ханойская башня»), время работы возрастает невероятно быстро.

Если соединить края трех кривых Коха таким образом, чтобы их инициаторы образовывали треугольник, то получится так называемая снежинка Коха.

**Порядок выполнения работы**

1. Ознакомиться с теоретическими сведениями.
2. Получить вариант задания у преподавателя.
3. Выполнить задание: написать программу на языке C++.
4. Продемонстрировать выполнение работы преподавателю.
5. Оформить отчет.
6. Защитить лабораторную работу.

**Требования к оформлению отчета**

Отчет по лабораторной работе должен содержать следующие разделы:
титульный лист;
цель работы;
задание на лабораторную работу;
ход работы;
ответы на контрольные вопросы;
выводы по проделанной работе.

**Варианты заданий**

1. Напишите программу для решения головоломки «Ханойская башня» и покажите ходы, изобразив передвижения дисков между колышками.
2. Создайте программу для рисования снежинок Коха.
3. В стандартной снежинке Коха генератор создает углы, равные 60. P но можно использовать и другие значения для получения интересных результатов. Напишите программу, которая разрешит пользователю задавать угол в качестве входного параметра.
4. Реализуйте программу для черчения кривых Гильберта.
5. Создайте программу, которая чертит кривые Серпинского.
6. Создайте программу, которая рисует ковер Серпинского.
7. Реализуйте программу для решения задачи о восьми ферзях.
8. Усовершенствуйте задачу о восьми ферзях: отследите, сколько ферзей может атаковать определенную позицию на доске.
9. Напишите программу, которая решает задачу о ходе коня методом возврата и в которой пользователю разрешается задавать размеры шахматной доски. Каковы будут наименьшие размеры доски, чтобы ход конем оказался возможен?
10. Используйте любимый язык программирования, чтобы решить задачу о ходе коня с использованием эвристического алгоритма Варнсдорфа.

**Контрольные вопросы**

1. Что такое рекурсия?
2. Какие есть способы удаления рекурсии, зачем это применяется?
3. Что такое алгоритмы с возвратом, приведите примеры?

# Лабораторная работа № 8

# ОБОБЩЕННОЕ ПРОГРАММИРОВАНИЕ В ЯЗЫКЕ С++

**Цель работы:** изучить основные понятия обобщенного программирования.

***Краткие теоретические сведения***

***Назначение шаблонов.*** Алгоритм выполнения какого-либо действия можно записывать независимо от того, какого типа данные обрабатываются. Простейшим примером служит определение минимума из двух величин.

```
if (a < b)
    x = a;
else
    x = b;
```

Независимо от того, к какому именно типу принадлежат переменные a, b и x, если это один и тот же тип, для которого определена операция "меньше", запись будет одна и та же. Было бы естественно определить функцию min, возвращающую минимум из двух своих аргументов. Возникает вопрос, как описать аргументы этой функции? Конечно, можно определить min для всех известных типов, однако, во-первых, пришлось бы повторять одну и ту же запись многократно, а во-вторых, с добавлением новых классов добавлять новые функции.

Аналогичная ситуация встречается и в случае со многими сложными структурами данных. В классе, реализующем связанный список целых чисел, алгоритмы добавления нового атрибута списка, поиска нужного атрибута и так далее не зависят от того, что атрибуты списка – целые числа. Точно такие же алгоритмы нужно будет реализовать для списка вещественных чисел или *указателей* на класс Book.

Механизм *шаблонов* в языке Си++ позволяет эффективно решать многие подобные задачи.

***Функции-шаблоны.***

Запишем алгоритм поиска минимума двух величин, где в качестве параметра используется тип этих величин.

```
template <class T>
const T&  min(const T&  a, const T&  b)
{
    if (a < b)
        return a;
    else
        return b;
}
```

Данная запись еще не создала ни одной функции, это лишь *шаблон* для определенной функции. Только тогда, когда происходит обращение к функции с аргументами конкретного типа, будет выполнена генерация конкретной функции.

```
int x, y, z;
String s1, s2, s3;
. . .
// генерация функции min для класса String
s1 = min(s2, s3);
. . .
// генерация функции min для типа int
x = min(y, z);
```

Первое обращение к функции min генерирует функцию

```
const String& min(const String& a,
                  const String& b);
```
Второе обращение генерирует функцию
```
const int& min(const int& a,
               const int& b);
```
Объявление *шаблона функции* min говорит о том, что конкретная функция зависит от одного параметра – типа T. Первое обращение к min в программе использует аргументы типа String. В *шаблон функции* подставляется тип String вместо T. Получается функция:
```
const String& min(const String& a,
                  const String& b)
{
  if (a < b)
    return a;
  else
    return b;
}
```
Эта функция компилируется и используется в программе. Аналогичные действия выполняются и при втором обращении, только теперь вместо параметра T подставляется тип int. Как видно из приведенных примеров, компилятор сам определяет, какую функцию надо использовать, и автоматически генерирует необходимое определение.

У *функции-шаблона* может быть несколько параметров. Так, например, функция find библиотеки STL (стандартной библиотеки *шаблонов*), которая ищет первый элемент, равный заданному, в интервале значений, имеет вид:
```
template <class InIterator, class T>
InIterator
find(InIterator first, InIterator last,
     const T& val);
```
Класс T – это тип элементов интервала. Тип InIterator – тип *указателя* на его начало и конец.

***Шаблоны классов***

*Шаблон класса* имеет вид:
```
template <список параметров>
class объявление_класса
```
Список параметров *класса-шаблона* аналогичен списку параметров *функции-шаблона*: список классов и переменных, которые подставляются в объявление класса при генерации конкретного класса.

Очень часто *шаблоны* используются для создания коллекций, т.е. классов, которые представляют собой набор объектов одного и того же типа. Простейшим примером коллекции может служить массив. Массив, несомненно, очень удобная структура данных, однако у него имеется ряд существенных недостатков, к которым, например, относятся необходимость задавать размер массива при его определении и отсутствие контроля использования значений индексов при обращении к атрибутам массива.

Попробуем при помощи *шаблонов* устранить два отмеченных недостатка у одномерного массива. При этом по возможности попытаемся сохранить синтаксис обращения к атрибутам массива. Назовем новую структуру данных *вектор* vector.
```
template <class T>
class vector
{
public:
  vector() : nItem(0), items(0) {};
  ~vector() { delete [] items; };
```

```
  void insert(const T& t)
   { T* tmp = items;
   items = new T[nItem + 1];
   memcpy(items, tmp, sizeof(T)* nItem);
   items[nItem++] = t;
   delete tmp; }
  void remove(void)
   { T* tmp = items;
   items = new T[--nItem];
   memcpy(items, tmp, sizeof(T) * nItem);
   delete tmp;
   }
  const T& operator[](int index) const
   {
   if ((index < 0) || (index >= nItem))
     throw IndexOutOfRange;
   return items[index];
   }
  T& operator[](int index)
   {
   if ((index < 0) || (index >= nItem))
      throw IndexOutOfRange;
   return items[index];
   }
private:
  T* items;
  int nItem;
};
```

Кроме конструктора и деструктора, у нашего *вектора* есть только три метода: метод insert добавляет в конец *вектора* новый элемент, увеличивая длину *вектора* на единицу, метод remove удаляет последний элемент *вектора*, уменьшая его длину на единицу, и операция [] обращается к n-ому элементу *вектора*.

```
vector<int> IntVector;
IntVector.insert(2);
IntVector.insert(3);
IntVector.insert(25);
// получили вектор из трех атрибутов:
// 2, 3 и 25
// переменная x получает значение 3
int x = IntVector[1];
// произойдет исключительная ситуация
int y = IntVector[4];
// изменить значение второго атрибута вектора.
IntVector[1] = 5;
```

Обратите внимание, что операция [] определена в двух вариантах – как константный метод и как неконстантный. Если операция [] используется справа от операции присваивания (в первых двух присваиваниях), то используется ее константный вариант, если слева (в последнем присваивании) – неконстантный. Использование операции индексирования [] слева от операции присваивания означает, что значение объекта изменяется, соответственно, нужна неконстантная операция.

Параметр *шаблона* vector – любой тип, у которого определены операция присваивания и стандартный конструктор. (Стандартный конструктор необходим при операции new для массива.)

Так же, как и с *функциями-шаблонами*, при задании первого объекта типа vector<int> автоматически происходит генерация конкретного класса из *шаблона*. Если далее в программе будет использоваться *вектор* вещественных чисел или строк, значит, будут сгенерированы конкретные классы и для них. Генерация конкретного класса означает, что генерируются все его методы, соответственно, размер исходного кода растет. Поэтому из небольшого *шаблона* может получиться большая программа. Ниже мы рассмотрим одну возможность сокращения размера программы, использующей почти однотипные *шаблоны*.

Сгенерировать конкретный класс из *шаблона* можно явно, записав:

template vector<int>;

Этот оператор не создаст никаких объектов типа vector<int>, но, тем не менее, вызовет генерацию класса со всеми его методами.

***"Интеллигентный указатель"***

Рассмотрим еще один пример использования *класса-шаблона*. С его помощью мы попытаемся " усовершенствовать" *указатели* языка Си++. Если *указатель* указывает на объект, выделенный с помощью операции new, необходимо явно вызывать операцию delete тогда, когда объект становится не нужен. Однако далеко не всегда просто определить, нужен объект или нет, особенно если на него могут ссылаться несколько разных *указателей*. Разработаем класс, который ведет себя очень похоже на *указатель*, но автоматически уничтожает объект, когда уничтожается последняя ссылка на него. Назовем этот класс "***интеллигентный указатель***" (Smart Pointer). Идея заключается в том, что настоящий *указатель* мы окружим специальной оболочкой. Вместе со значением *указателя* мы будем хранить счетчик – сколько других объектов на него ссылается. Как только значение этого счетчика станет равным нулю, объект, на который *указатель* указывает, пора уничтожать.

Структура Ref хранит исходный *указатель* и счетчик ссылок.

```
template <class T>
struct Ref
{
  T* realPtr;
  int counter;
};
```

Теперь определим интерфейс *"интеллигентного указателя"*:

```
template <class T>
class SmartPtr
{
public:
  // конструктор из обычного указателя
  SmartPtr(T* ptr = 0);
  // копирующий конструктор
  SmartPtr(const SmartPtr& s);
  ~SmartPtr();
  SmartPtr& operator=(const SmartPtr& s);
  SmartPtr& operator=(T* ptr);
  T* operator->() const;
  T& operator*() const; private:
  Ref<T>* refPtr;
};
```

У класса SmartPtr определены операции обращения к элементу ->, взятия по адресу "*" и операции присваивания. С объектом класса SmartPtr можно обращаться практически так же, как с обычным *указателем*.

```
struct A
{
    int x;
    int y;
};
SmartPtr<A> aPtr(new A);
int x1 = aPtr->x;
(*aPtr).y = 3;
 // создать новый указатель
 // обратиться к элементу A
 // обратиться по адресу
```

Рассмотрим реализацию методов класса SmartPtr. Конструктор инициализирует объект *указателем*. Если *указатель* равен нулю, то refPtr устанавливается в ноль. Если же конструктору передается ненулевой *указатель*, то создается структура Ref, счетчик обращений в которой устанавливается в 1, а *указатель* – в переданный *указатель*:

```
template <class T>
SmartPtr<T>::SmartPtr(T* ptr)
{
 if (ptr == 0)
   refPtr = 0;
 else {
   refPtr = new Ref<T>;
   refPtr->realPtr = ptr;
   refPtr->counter = 1;
 }
}
```

Деструктор уменьшает количество ссылок на 1 и, если оно достигло 0, уничтожает объект

```
template <class T>
SmartPtr <T>::~SmartPtr()
{
 if (refPtr != 0) {
   refPtr->counter--;
   if (refPtr->counter <= 0) {
     delete refPtr->realPtr;
     delete refPtr;
   }
 }
}
```

Реализация операций -> и * довольно проста:

```
template <class T>
T*
SmartPtr<T>::operator->() const
{
 if (refPtr != 0)
   return refPtr->realPtr;
 else
   return 0;
}
```

```
template <class T>
T&
SmartPtr<T>::operator*() const
{
  if (refPtr != 0)
    return *refPtr->realPtr;
  else
    throw bad_pointer;
}
```

Самые сложные для реализации – копирующий конструктор и операции присваивания. При создании объекта SmartPtr – копии имеющегося – мы не будем копировать сам исходный объект. Новый *"интеллигентный указатель"* будет ссылаться на тот же объект, мы лишь увеличим счетчик ссылок.

```
template <class T>
SmartPtr<T>::SmartPtr(const
          SmartPtr& s):refPtr(s.refPtr)
{
  if (refPtr != 0)
    refPtr->counter++;
}
```

При выполнении присваивания, прежде всего, нужно отсоединиться от имеющегося объекта, а затем присоединиться к новому, подобно тому, как это сделано в копирующем конструкторе.

```
template <class T>
SmartPtr&
SmartPtr<T>::operator=(const SmartPtr& s)
{
  // отсоединиться от имеющегося указателя
  if (refPtr != 0) {
    refPtr->counter--;
    if (refPtr->counter <= 0) {
      delete refPtr->realPtr;
      delete refPtr;
    }
  }
  // присоединиться к новому указателю
  refPtr = s.refPtr;
  if (refPtr != 0)
    refPtr->counter++;
}
```

В следующей функции при ее завершении объект класса Complex будет уничтожен:

```
void foo(void)
{
  SmartPtr<Complex> complex(new Complex);
  SmartPtr<Complex> ptr = complex;
  return;
}
```

***Задание свойств класса***

Одним из методов использования *шаблонов* является уточнение поведения с помощью дополнительных параметров *шаблона*. Предположим, мы пишем функцию сортировки *вектора*:

```
template <class T>
void sort_vector(vector<T>& vec)
{
  for (int i = 0; i < vec.size() -1; i++)
    for (int j = i; j < vec.size(); j++) {
      if (vec[i] < vec[j]) {
        T tmp = vec[i];
        vec[i] = vec[j];
        vec[j] = tmp;
      }
    }
}
```

Эта функция будет хорошо работать с числами, но если мы захотим использовать ее для массива *указателей* на строки (char*), то результат будет несколько неожиданный. Сортировка будет выполняться не по значению строк, а по их адресам (операция "меньше" для двух *указателей* – это сравнение значений этих *указателей*, т.е. адресов величин, на которые они указывают, а не самих величин). Чтобы исправить данный недостаток, добавим к *шаблону* второй параметр:

```
template <class T, class Compare>
void sort_vector(vector<T>& vec)
{
  for (int i = 0; i < vec.size() -1; i++)
    for (int j = i; j < vec.size(); j++) {
      if (Compare::less(vec[i], vec[j])) {
        T tmp = vec[i];
        vec[i] = vec[j];
        vec[j] = tmp;
      }
    }
}
```

Класс Compare должен реализовывать статическую функцию less, сравнивающую два значения типа T. Для целых чисел этот класс может выглядеть следующим образом:

```
class CompareInt
{
  static bool less(int a, int b)
    { return a < b; };
};
```

Сортировка *вектора* будет выглядеть так:

```
vector<int> vec;
sort_vector<int, CompareInt>(vec);
```

Для *указателей* на байт (строк) можно создать класс

```
class CompareCharStr
{
  static bool less(char* a, char* b)
    { return strcmp(a,b) >= 0; }
};
```

и, соответственно, сортировать с помощью вызова

```
vector<char*> svec;
sort_vector<char*, CompareCharStr>(svec);
```

Как легко заметить, для всех типов, для которых операция "меньше" имеет нужный нам смысл, можно написать *шаблон класса* сравнения:

```
template<class T> Compare
```

```
{
  static bool less(T a, T b)
    { return a < b; };
};
```

и использовать его в сортировке (обратите внимание на пробел между закрывающимися угловыми скобками в параметрах *шаблона*; если его не поставить, компилятор спутает две скобки с операцией сдвига):

```
vector<double> dvec;
sort_vector<double, Compare<double> >(dvec);
```

Чтобы не загромождать запись, воспользуемся возможностью задать значение параметра по умолчанию. Так же, как и для аргументов функций и методов, для параметров *шаблона* можно определить значения по умолчанию. Окончательный вид функции сортировки будет следующий:

```
template <class T, class C = Compare<T> >
void sort_vector(vector<T>& vec)
{
  for (int i = 0; i < vec.size() -1; i++)
    for (int j = i; j < vec.size(); j++) {
      if (C::less(vec[i], vec[j])) {
        T tmp = vec[i];
        vec[i] = vec[j];
        vec[j] = tmp;
      }
    }
}
```

Второй параметр *шаблона* иногда называют *параметром-штрих*, поскольку он лишь модифицирует поведение класса, который манипулирует типом, определяемым первым параметром.

**Порядок выполнения работы.**

1. При домашней подготовке необходимо изучить литературу по теме лабораторной работы.
2. Получить задание у преподавателя.
3. Разработать алгоритм решения задачи и написать программу на языке C++, реализующую задание.
4. Проверить правильность ее работы.
5. Составить отчет и защитить работу.

**Требования к отчету.**

Отчет по лабораторной работе должен содержать титульный лист, цели и задачи работы, текст задания, алгоритм и программную реализацию решения, контрольные примеры, ответы на контрольные вопросы.

**Варианты заданий.**

1. Спроектировать шаблон класса complex
2. Спроектировать шаблон класса vector
3. Спроектировать шаблон класса matrix
4. Спроектировать шаблон класса list
5. Спроектировать шаблон класса qeue
6. Спроектировать шаблон класса tree
7. Спроектировать шаблон класса graph
8. Спроектировать шаблон класса set
9. Спроектировать шаблон класса line
10. Спроектировать шаблон класса triangl

**Контрольные вопросы.**

1. Назначение шаблонов?
2. Понятие функции шаблона?
3. Шаблон класса?
4. Что такое smart pointer?

# Лабораторная работа № 9

# ЖАДНЫЕ АЛГОРИТМЫ

**Цель работы:** Ознакомиться с методами решения задач, используя жадные алгоритмы.

**Порядок выполнения работы**

1. Ознакомиться с теоретическими сведениями.
2. Получить вариант задания у преподавателя.
3. Выполнить задание: написать программу на языке C++.
4. Продемонстрировать выполнение работы преподавателю.
5. Оформить отчет.
6. Защитить лабораторную работу.

**Требования к оформлению отчета**

Отчет по лабораторной работе должен содержать следующие разделы: титульный лист; цель работы; задание на лабораторную работу; ход работы; ответы на контрольные вопросы; выводы по проделанной работе.

**Варианты заданий**

1. Написать программу для решения задачи, используя жадные алгоритмы. Толик придумал новую технологию программирования. Он хочет уговорить друзей использовать ее. Однако все не так просто. *i*-й друг согласится использовать технологию Толика, если его авторитет будет не меньше *ai* (авторитет выражается целым числом). Как только *i*-й друг начнет ее использовать, к авторитету Толика прибавится число *bi* (попадаются люди, у которых $bi < 0$). Помогите Толику наставить на путь истинный как можно больше своих друзей. Входные данные: на первой строке содержатся два целых числа: Количество друзей у Толика $n$ и первоначальный авторитет Толика $a_0$ ($-10^9 \le a_0 \le 10^9$). Следующие $n$ строк содержат пары целых чисел $a_i$ и $b_i$ ($-10^9 \le a_i, b_i \le 10^9$). Выходные данные: на первой строке выведите число $m$ − максимальное число друзей, которых может увлечь Толик. На второй строке выведите $m$ чисел − номера друзей в том порядке, в котором их нужно агитировать.
2. Написать программу для решения задачи, используя жадные алгоритмы. Мальчик Костя очень любит конфеты, но мама не разрешает ему брать их слишком много. Поэтому каждый раз, когда Костя хочет съесть конфету, мама предлагает ему сыграть в игру. Изначально у Кости нет конфет, а у мамы их $N$ (они пронумерованы от 1 до $N$). На каждой конфете мама написала два числа $Ai$ и $Ci$. Мама очень следит за *уровнем вредности* конфет, который получает ее сын. Изначально этот уровень равен 0. На каждом ходу игры Костя может взять одну конфету. Если Костя возьмет конфету с номером $i$, то *уровень вредности* увеличивается на $Ai$. Если сразу после этого *уровень вредности* становится большей $Ci$, то брать эту конфету запрещается. Брать конфеты можно в произвольном порядке, но одну и ту же можно брать не более одного раза. Помогите Косте взять как можно больше конфет (вне зависимости от финального *уровня вредности*). Входные данные: в первой сроке входных данных записано целое число $N$ ($1 \le N \le 1000$) − количество видов конфет. Во второй строке записаны $N$ целых чисел $Ai$ ($1 \le Ai \le 10^6$). В третей строке записаны $N$ целых чисел $Ci$ ($1 \le Ci \le 10^9$). Выходные данные: в единственной

строке выведите целое число, равное максимальному количеству конфет, которые может взять Костя.

3. Написать программу для решения задачи, используя жадные алгоритмы. Теплым весенним днем группа из $N$ школьников-программистов гуляла в окрестностях города Тулы. К несчастью, они набрели на большую и довольно глубокую яму. Как это случилось — непонятно, но вся компания оказалась в этой яме. Глубина ямы равна $H$. Каждый школьник знает свой рост по плечи $hi$ и длину своих рук $li$. Таким образом, если он, стоя на дне ямы, поднимет руки, то его ладони окажутся на высоте $hi+li$ от уровня дна ямы. Школьники могут, вставая друг другу на плечи, образовывать вертикальную колонну. При этом любой школьник может встать на плечи любого другого школьника. Если под школьником $i$ стоят школьники $j1$,$j2$,...,$jk$, то он может дотянуться до уровня $hj1+hj2+...+hjk+hi+li$. Если школьник может дотянуться до края ямы (то есть $hj1+hj2+...+hjk+hi+li \geq H$), то он может выбраться из нее. Выбравшиеся из ямы школьники не могут помочь оставшимся. Найдите наибольшее количество школьников, которые смогут выбраться из ямы до прибытия помощи, и перечислите их номера. Входные данные: В первой строке записано натуральное число $N$ — количество школьников, попавших в яму. Далее в $N$ строках указаны по два целых числа: рост $i$-го школьника по плечи $hi$ и длина его рук $li$. В последней строке указано целое число — глубина ямы $H$. Выходные данные: В первой строке выведите $K$ — максимальное количество школьников, которые смогут выбраться из ямы. Если $K>0$, то во второй строке выведите их номера в том порядке, в котором они вылезают из ямы. Школьники нумеруются с единицы в том порядке, в котором они заданы во входном файле. Если существует несколько решений, выведите любое.
4. Написать программу для решения задачи, используя жадные алгоритмы. У Васи в комнате очень много коробок, которые валяются в разных местах. Васина мама хочет, чтобы он прибрался. Свободного места в комнате мало и поэтому Вася решил собрать все коробки и составить их одну на другую. К сожалению, это может быть невозможно. Например, если на картонную коробку с елочными украшениями положить что-то железное и тяжелое, то вероятно следующий Новый год придется встречать с новыми игрушками. Вася взвесил каждую коробку и оценил максимальный вес который она может выдержать. Помогите ему определить какое наибольшее количество коробок $m$ он сможет составить одну на другую так, чтобы для каждой коробки было верно, что суммарный вес коробок сверху не превышает максимальный вес, который она может выдержать. Входные данные: первая строка содержит целое число $n$ ($1 \leq n \leq 10^5$) - количество коробок в комнате. Каждая следующая из $n$ строк содержит два целых числа $wi$ и $ci$ ($1 \leq wi \leq 10^5, 1 \leq ci \leq 10^9$), где $wi$ − это вес коробки с номером $i$, а $ci$ − это вес который она может выдержать. Выходные данные:

одно число − ответ на задачу.

5. Написать программу для решения задачи, используя жадные алгоритмы. Дана лекционная аудитория, в которой несколько профессоров хотят прочесть свои лекции. Для составления расписания профессора подали заявки, вида [$si$, $fi$) – время начала и конца лекции. Лекция считается открытым полуинтервалом, то есть какая-то лекция может начаться в момент окончания другой, без перерыва. Составьте расписание занятий так, чтобы выполнить максимальное количество заявок. Входные данные: в первой строке вводится натуральное число $N$, не более 1000 – общее количество заявок. Затем вводится $N$ строк с описаниями заявок - по два числа в каждом $si$ и $fi$. Гарантируется, что $si < fi$. Время начала и окончания лекции – натуральное число, не превышает 1440 (в минутах с начала

суток). Выходные данные: выведите одно число – максимальное количество заявок, которые можно выполнить.

6. Написать программу для решения задачи, используя жадные алгоритмы. В некоей воинской части есть сапожник. Рабочий день сапожника длится *N* минут. Заведующий складом оценивает работу сапожника по количеству починенной обуви, независимо от того, насколько сложный ремонт требовался в каждом случае. Дано *k* сапог, нуждающихся в починке. Определите какое максимальное количество из них сапожник сможет починить за один рабочий день. Входные данные: в первой строке вводятся число *N* (натуральное, не превышает 1000), и число *k* (натуральное, не превышает 500). Затем идет *k* чисел – количество минут, которые требуются чтобы починить *i*-й сапог (времена – натуральные числа, не превосходят 100). Выходные данные: выведите единственное число – максимальное количество сапог, которые можно починить за один рабочий день.
7. Написать программу для решения задачи, используя жадные алгоритмы. Андрей едет из пункта A в пункт B на автомобиле. Расстояние между этими пунктами равно *N* километров. Известно, что с полным баком автомобиль способен проехать k километров. Дана карта, на которой отмечены координаты бензоколонок, относительно пункта A. Определите минимальное число заправок, которые придется сделать Андрею чтобы успешно достичь пункта B. Известно, что при выезде из пункта A бак был полон. Входные данные: в первой строке вводятся числа *N* и *k* (натуральные, не превосходят 1000). В следующей строке вводится количество бензоколонок *S*, потом следует *S* натуральных чисел, не превосходящих *N* – расстояния от пункта A до каждой заправки. *Заправки упорядочены по удаленности от пункта A*. Выходные данные: если при данных условиях пункта B достичь невозможно, то вывести число -1. Если решение существует, то вывести минимальное количество остановок на дозаправку, которое нужно чтобы достичь пункта B.
8. Написать программу для решения задачи, используя методами жадные алгоритмы. В некотором королевстве есть *N* провинций. Король пожелал объединить все их под своей самодержавной властью. Естественно, чтобы никто не догадался об этих планах, он будет это делать поэтапно, а именно: раз в год он будет объединять какие-то две провинции в одну. Чтобы жителям обеих провинций не было обидно, новому территориальному образованию будет присвоено новое название, которое будет отличаться от обоих старых названий. Естественно, это потребует выпуска новых паспортов для жителей обеих провинций. Очевидно, что если в первой провинции *pi* жителей, а во второй – *pj* жителей, то для них надо выпустить *pi*+*pj* новых паспортов. На следующий год король объединяет еще какие-то две провинции. И так далее, до тех пор пока вся территория королевства не будет объединена в одну большую «провинцию». Определите, какое наименьшее количество новых паспортов придется выпустить, если король будет объединять провинции оптимально с этой точки зрения. Входные данные: в первой строке вводится число *N* (натуральное, не превышает 105) – количество провинций. Затем вводится *N* чисел – количество жителей каждой провинции (натуральное, не превосходит $10^9$). *Гарантируется, что изначально в королевстве хотя бы две провинции.* Выходные данные: выведите единственное число – количество новых паспортов, которые придется выпустить.
9. Написать программу для решения задачи, используя жадные алгоритмы. Толик придумал новую технологию программирования. Он хочет уговорить друзей использовать ее. Однако все не так просто. *i*-й друг согласится использовать технологию Толика, если его авторитет будет не меньше *ai* (авторитет

выражается целым числом). Как только $i$-й друг начнет ее использовать, к авторитету Толика прибавится число $bi$ (попадаются люди, у которых $bi < 0$). Помогите Толику наставить на путь истинный как можно больше своих друзей. Входные данные: на первой строке содержатся два целых числа: Количество друзей у Толика $n$ и первоначальный авторитет Толика $a_0$ ($-10^9 \le a_0 \le 10^9$). Следующие $n$ строк содержат пары целых чисел $a_i$ и $b_i$ ($-10^9 \le a_i, b_i \le 10^9$). Выходные данные: на первой строке выведите число $m$ − максимальное число друзей, которых может увлечь Толик. На второй строке выведите $m$ чисел − номера друзей в том порядке, в котором их нужно агитировать.

10. Написать программу для решения задачи, используя жадные алгоритмы. Мальчик Костя очень любит конфеты, но мама не разрешает ему брать их слишком много. Поэтому каждый раз, когда Костя хочет съесть конфету, мама предлагает ему сыграть в игру. Изначально у Кости нет конфет, а у мамы их $N$ (они пронумерованы от 1 до $N$). На каждой конфете мама написала два числа $Ai$ и $Ci$. Мама очень следит за *уровнем вредности* конфет, который получает ее сын. Изначально этот уровень равен 0. На каждом ходу игры Костя может взять одну конфету. Если Костя возьмет конфету с номером $i$, то *уровень вредности* увеличивается на $Ai$. Если сразу после этого *уровень вредности* становится большей $Ci$, то брать эту конфету запрещается. Брать конфеты можно в произвольном порядке, но одну и ту же можно брать не более одного раза. Помогите Косте взять как можно больше конфет (вне зависимости от финального *уровня вредности*). Входные данные: в первой сроке входных данных записано целое число $N$ ($1 \le N \le 1000$) − количество видов конфет. Во второй строке записаны $N$ целых чисел $Ai$ ($1 \le Ai \le 10^6$). В третей строке записаны $N$ целых чисел $Ci$ ($1 \le Ci \le 10^9$). Выходные данные: в единственной строке выведите целое число, равное максимальному количеству конфет, которые может взять Костя.

**Контрольные вопросы**

1. Что такое жадные алгоритмы?
2. Какие классы задач можно решать с помощью жадных алгоритмов? Пример.

# Лабораторная работа № 10

# ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

**Цель работы:** Ознакомиться с методами динамического программирования.

**Порядок выполнения работы**

1. Ознакомиться с теоретическими сведениями.
2. Получить вариант задания у преподавателя.
3. Выполнить задание: написать программу на языке C++.
4. Продемонстрировать выполнение работы преподавателю.
5. Оформить отчет.
6. Защитить лабораторную работу.

**Требования к оформлению отчета**

Отчет по лабораторной работе должен содержать следующие разделы: титульный лист; цель работы; задание на лабораторную работу; ход работы; ответы на контрольные вопросы; выводы по проделанной работе.

**Варианты заданий**

1. Написать программу для решения задачи методами динамического программирования. Требуется подсчитать, сколько различных текстовых сообщений можно написать используя не более k нажатий на клавиатуре кнопочного телефона.
2. Написать программу для решения задачи методами динамического программирования. На вершине лесенки, содержащей $N$ ступенек, находится мячик, который начинает прыгать по ним вниз, к основанию. Мячик может прыгнуть на следующую ступеньку, на ступеньку через одну или через 2. (То есть, если мячик лежит на 8-ой ступеньке, то он может переместиться на 5-ую, 6-ую или 7-ую.) Определить число всевозможных "маршрутов" мячика с вершины на землю. Входные данные: вводится одно число $0 < N < 31$. Выходные данные: выведите одно число — количество маршрутов.
3. Написать программу для решения задачи методами динамического программирования. Вычислите n-й член последовательности, заданной формулами: $a_{2n} = a_n + 1$ (n>0), $a_{2n+2} = a_{2n+1} - a_n$ (n>0), $a_0=1$, $a_1=1$. Входные данные: вводится натуральное число n, не превосходящее 1000. Выходные данные: выведите ответ к задаче.
4. Написать программу для решения задачи методами динамического программирования. Требуется подсчитать количество последовательностей длины $N$, состоящих из 0 и 1, в которых никакие две единицы не стоят рядом. Входные данные: на вход программы поступает целое число $N$ ($1 \leq N \leq 100$). Выходные данные: выведите количество искомых последовательностей.
5. Написать программу для решения задачи методами динамического программирования. Мальчик подошел к платной лестнице. Чтобы наступить на любую ступеньку, нужно заплатить указанную на ней сумму. Мальчик умеет перешагивать на следующую ступеньку, либо перепрыгивать через ступеньку. Требуется узнать, какая наименьшая сумма понадобится мальчику, чтобы добраться до верхней ступеньки. Входные данные: в первой строке вводится одно натуральное число $N \leq 100$ — количество ступенек. В следующей строке вводятся $N$ натуральных чисел, не превосходящих 100 — стоимость каждой ступеньки (снизу вверх). Выходные данные: выведите одно число — наименьшую возможную стоимость прохода по лесенке.
6. Написать программу для решения задачи методами динамического программирования. В прямоугольной таблице $N$x$M$ в начале игрок находится в левой

верхней клетке. За один ход ему разрешается перемещаться в соседнюю клетку либо вправо, либо вниз (влево и вверх перемещаться запрещено). Посчитайте, сколько есть способов у игрока попасть в правую нижнюю клетку. Входные данные: вводятся два числа $N$ и $M$ - размеры таблицы ($1 \le N \le 10$, $1 \le M \le 10$). Выходные данные: выведите искомое количество способов.

7. Написать программу для решения задачи методами динамического программирования. Даны две последовательности, требуется найти длину их наибольшей общей подпоследовательности. Входные данные: в первой строке входных данных содержится число $N$ – длина первой последовательности ($1 \le N \le 1000$). Во второй строке заданы члены первой последовательности (через пробел) – целые числа, не превосходящие 10000 по модулю. В третьей строке записано число $M$ – длина второй последовательности ($1 \le M \le 1000$). В четвертой строке задаются члены второй последовательности (через пробел) – целые числа, не превосходящие 10000 по модулю. Выходные данные: требуется вывести одно число – длину наибольшей общей подпоследовательности двух данных последовательностей или 0, если такой подпоследовательности нет.
8. Написать программу для решения задачи методами динамического программирования. В государстве в обращении находятся банкноты определенных номиналов. Национальный банк хочет, чтобы банкомат выдавал любую запрошенную сумму при помощи минимального числа банкнот, считая, что запас банкнот каждого номинала неограничен. Помогите Национальному банку решить эту задачу. Входные данные: первая строка входных данных содержит натуральное число N не превосходящее 100 — количество номиналов банкнот в обращении. Вторая строка входных данных содержит N различных натуральных чисел $x_1$, $x_2$, ..., $x_N$, не превосходящих $10^6$ — номиналы банкнот. Третья строчка содержит натуральное число S, не превосходящее $10^6$ —сумму, которую необходимо выдать. Выходные данные: программа должна найти представление числа S виде суммы слагаемых из множества $x_i$, содержащее минимальное число слагаемых и вывести это представление на экран (в виде последовательности чисел, разделенных пробелами). Если таких представлений существует несколько, то программа должна вывести любое (одно) из них. Если такое представление не существует, то программа должна вывести строку No solution.
9. Написать программу для решения задачи методами динамического программирования. Вы решили заказать пиццу с доставкой на дом. Известно, что для клиентов, сделавших заказ на сумму более $C$ рублей, доставка является бесплатной, при заказе на $C$ рублей и меньше доставка стоит B рублей.
10. Вы уже выбрали товар, стоимостью $A$ рублей. В наличии имеются еще $N$ товаров стоимостью $d_1, ..., d_N$ рублей, каждый в единственном экземпляре. Их также можно включить в заказ. Как потратить меньше всего денег и получить на дом уже выбранный товар в $A$ рублей? Входные данные: сначала вводятся числа $A, B, C, N$, а затем $N$ чисел $d_1, ..., d_N$. Все числа целые, $1 \le A \le 1000$, $1 \le B \le 1000$, $1 \le C \le 1000$, $0 \le N \le 1000$, $1 \le di \le 1\ 000\ 000$. Выходные данные: выведите сначала суммарное количество денег, которое придется потратить. Если при этом вы планируете сделать дополнительный заказ с расчетом на бесплатную доставку, то далее выведите количество этих товаров и их номера в возрастающем порядке. Если же Вы будете оплачивать доставку сами, то далее выведите одно число –1 (минус один).

**Контрольные вопросы**

1. Что такое динамическое программирование?
2. Приведите основные принципы динамического программирования?
3. Какие классы задач можно решать с помощью динамического программирования? Примеры.