МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

А.Г.ГЛУЩЕНКО

ОСНОВЫ АЛГОРИТМИТИЗАЦИИ И ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++

Методические указания к практическим работам

Санкт-Петербург
2020

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире каждый человек погружен в информационную среду. Практически каждая сфера нашей жизни сопряжена с IT-сферой. Причем непомерно растущие потребности человека заставляют развиваться информационные технологии с бешенной скоростью. От того и рождается большой спрос на специалистов в данной области.

Практически каждый человек, имеющий достаточно опыта работы с персональным компьютером, может написать простейшую программу. К сожалению, программистом себя он после такого назвать не может. На изучение всех алгоритмов, методов и особенностей определенного языка программирования может уйти несколько лет, особенно, если это язык программирования C++.

Данное учебно-методическое пособие предназначено для выполнения практических работ по дисциплине «Программирование» и содержит всю необходимую справочную информацию, необходимую для выполнения практических работ. Практические работы построены таким образом, чтобы поэтапно изучить основы алгоритмизации и программирования на языке C++ и получить структурированное представление о программировании в целом.

Изложение материала сопровождается пояснительными примерами, которые рассматривают различные алгоритмы, методы, структуры данных и примеры написания программ.

В первом разделе содержаться общие сведения о языке программирования C++. Углубленное изучение языка происходит при выполнении практических работ. Поэтому некоторая информация в первом разделе опускается. Прочтение и изучение раздела об общих сведениях о языке C++ обеспечит минимальные навыки программирования, которые пригодятся при выполнении практических работ.

ПРАКТИЧЕСКАЯ РАБОТА №1
ОБЩИЕ СВЕДЕНИЯ О ЯЗЫКЕ ПРОГРАММИРОВАНИЯ С++

Цель работы: ознакомиться с синтаксисом и основными понятиями языка программирования языка С++. Ознакомиться с основными структурами и реализовать простейшую программу, чтобы закрепить теоретический материал.

1.1. Алфавит языка

Изучение С++ лучше начать с самого простого – с алфавита. Затем стоит затрагивать новые элементы, постепенно углубляясь в самые сложные методы и структуры данных.

Алфавит С++ включает:

- прописные и строчные латинские буквы, и знак подчеркивания;
- арабские цифры от 0 до 9, шестнадцатеричные цифры от A до F;
- специальные знаки:
  “ { } , | [ ] ( ) + - / % * . \ ‘ : ? < = > ! & # _ ; ^
- пробельные символы: пробел, символ табуляции, символ перехода на новую строку.

Символы алфавита формируют лексемы языка, которая является минимальной единицей языка, имеющая самостоятельный смысл:

- идентификаторы;
- ключевые (зарезервированные слова);
- знаки операций. Служат для указания действий над операндами. В зависимости от количества операндов в С++ различают унарные, бинарные и одна тернарная операции;
- константы – неизменяемые величины;
- разделители (скобки, точка, запятая, пробельные символы);
- комментарии. Комментарии используются для пояснения функций, структур, переменных и т.д. Обычно для пояснения строчки кода используют комментарий, начинающийся с двух символов «прямая косая черта» (//) и заканчивается символом перехода на новую строку. Скобки

/* */ позволяют закомментировать любой блок программы (набор строк) и обычно используется для временного исключения блоков кода программы при отладке. При этом сам комментарий может состоять из любых символов.

Границы лексем определяются другими лексемами, такими, как разделители или знаки операций.

1.2. Идентификаторы

Идентификаторы – это имя различных программных объектов. В качестве идентификатора могут выступать имена переменных, констант, функций, структур, классов и т.д.

Синтаксически правильный идентификатор – это последовательность латинских букв, цифр и знака нижнего подчеркивания, которая не начинается с цифры. Можно использоваться верхний и нижний регистр при написании идентификатора. При этом `example`, `Example` и `EXAMPLE` – три различных имени. Пробел внутри имени не допускается.

Осмысленные имена – основное требование к идентификаторам, которые будут использоваться при выполнении практических работах, представленных в данной методичке.

Идентификатор может быть любой длины, верхний предел не установлен. Хотя существуют компиляторы, ограничивающие длину идентификатора. Идентификатор впервые используется при объявлении переменной, функции и т.п., после чего он может быть использован в последующих операторах программы. При этом идентификатор не должен совпадать с ключевым словом.

1.3. Ключевые слова

Ключевые слова – это зарезервированные идентификаторы, имеющие специальное значение для компилятора. У них имеется предопределенный смысл. Всего в C++ 63 ключевых слова, список которых приведен в табл. 1.1.

Таблица 1.1. – Список ключевых слов C++

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

Использование ключевого слова в качестве имени переменной приведет к возникновению синтаксической ошибки.

1.4. Структура и основные элементы программы

Любая программа, написанная на C++, является одной или набором связанных функций. Простейшая программа представлена в листинге 1.1

Листинг 1.1 – Пример простейшей программы на C++

```
/* theFirstProgram.cpp: Этот файл содержит функцию "main". Здесь начинается и заканчивается
выполнение программы. */

#include <iostream> /* Директива препроцессору включить в текст заголовочный файл
библиотеки классов C++ для управления вводом – выводом */

int main() // Основная функция программы – начало выполнения программы
{
       using namespace std; // Директива компилятору использовать пространство имен std
      cout << "This is my first program!" << endl;  // Вывод сообщения
       cout << "Это моя первая программа!" << endl; // Вывод сообщения на русском языке
      setlocale(LC_ALL, "Russian"); /* Установка локальных настроек на вывод русского
текста */
      cout << "Это моя первая программа!" << endl;
      return 0; // завершение функции main()
}
```

Ниже представлен результат выполнения скомпилированной программы из листинга 1.1:

```
This is my first program!
Єю ью  яхЁтр  яЁюуЁрььр!
Это моя первая программа!
```

Видно, что вторая строка – это всего лишь набор символов. После установки локальных настроек удалось вывести корректный текст. Это самый простой пример программы на языке C++.

Когда код не такой объемный и легко читаемый, то функции не востребованы. Но даже при небольшом размере кода может возникнуть ситуация, когда одно и то же действие выполняется много раз. Часто оптимизация достигается использованием циклов, но в определенных случаях уместно разбивать код на функции.

1.5. Переменные

Переменная – это именованная область памяти, в которой хранятся данные определенного типа. У любой переменной имеется имя и значение. Имя необходимо для обращения к области памяти, где хранится значение переменной. Значение переменной можно менять во время выполнения программы неоднократно. При этом любая переменная, перед ее использованием, должна быть описана. Примеры объявления переменных:

```
int a = 26; // Переменная a инициализирована значением 26
double pi = 3.14; // Переменная pi инициализирована значением 3.14
int a (26); // Переменная a инициализирована значением 26
bool m, w; // Объявлены переменные m и w
```

Переменные также можно инициализировать выражениями других переменных:

```
int a = 5;
int b = a * 5;
```

Общее правило определения переменной можно сформулировать так:

```
[класс памяти] <тип данных> <идентификатор – имя> [инициализатор];
```

Если переменная определена внутри блока, то она называется локальной, и область её действия – от момента объявления до конца блока, включая все вложения.

Если переменная определена вне любого блока, то она называется глобальной и её область действия – файл, в котором она определена, от момента объявления до конца файла.

Класс памяти определяет важные характеристики: время жизни и область видимости переменных. Если класс памяти не указан явно, он определяется компилятором по умолчанию исходя из контекста объявления.

Время жизни может быть постоянным (в течении выполнения всей программы) и временным (в течении выполнения блока).

Область видимости – та часть программы, из которой допустим обычный доступ к связанной с идентификатором областью памяти.

Класс памяти может принимать одно из значений:

- `auto` – автоматическая переменная. Время жизни – с момента объявления до конца блока. Память выделяется в стеке, освобождение памяти происходит по завершению времени жизни. Для локальных переменных спецификатор `auto` задается автоматически, а для глобальных переменных он не используется.
- `extern` – означает, что переменная объявляется в другом месте программы (в другом блоке или файле).
- `static` – статическая переменная. Время ее жизни постоянное. Статическая переменная может глобальной и локальной и инициализируется один раз.
- `register` – аналогичный auto спецификатор, но память выделяется по возможности в регистрах процессора. Доступ к таким переменным осуществляется непорядок быстрее. Если возможности выделить память нет, то переменная обрабатывается просто как `auto`.

Типы данных будут подробно рассмотрены в практической работе №1.

1.6. Операции и выражения

Операция – это действие, осуществляемое над операндами. При помощи операций происходит обработка данных.

В зависимости от количества операндов в языке C++ имеются унарные (с одни операндом), бинарные (с двумя операндами) и одна тернарная (с тремя операндами) операции.

Из знаков операций, операндов и круглых скобок строятся выражения. В качестве операндов могут выступать константы, переменные, функции и другие выражения. В результате вычисления выражения получается некоторое значение определенного типа. Тип данных значения выражения зависит от выполняемой операции и типов данных операндов.

Одной из особенностей языка программирования C++ является то, что некоторые операции имеют разное назначение, зависящее от контекста. В табл. 1.2 приведены основные операции.

Таблица 1.2 – Основные операции языка C++

Унарные операции	
Операция	**Описание**
++	увеличение на 1 (инкремент)
--	уменьшение на 1 (декремент)
sizeof	размер объекта или тапа данных в байтах
~	поразрядное отрицание
!	логическое отрицание
-	арифметическое отрицание (унарный минус)
+	унарный плюс
(тип)	явное преобразование типов
Бинарные операции	
*	умножение
/	деление
%	остаток от деления
+	сложение
-	вычитание
<<	сдвиг влево
>>	сдвиг вправо
<	меньше
<=	меньше или равно

Окончание таблицы 1.2

>	больше
>=	больше или равно
==	равно
!=	не равно
&	поразрядная конъюнкция (И)
^	поразрядное исключающее ИЛИ
|	поразрядная дизъюнкция (ИЛИ)
&&	логическое И
||	логическое ИЛИ
Тернарная операция	
?:	условная операция

Особое значение имеет операция присвоения и ее модификации (табл. 1.3).

Таблица 1.3 – Операция присвоения и её модификации.

Операция	**Краткое описание**
=	присваивание
*=	умножение с присваиванием
/=	деление с присваиванием
%=	остаток отделения с присваиванием
+=	сложение с присваиванием
-=	вычитание с присваиванием
<<=	сдвиг влево с присваиванием
>>=	сдвиг вправо с присваиванием
&=	поразрядное И с присваиванием
|=	поразрядное ИЛИ с присваиванием
^=	поразрядное исключающее ИЛИ с присваиванием

У описанных операций имеются свои особенности. К примеру, операция увеличения (инкремент) и операция уменьшения (декремент) имеют две формы записи – префиксную, когда операция записывается перед операндом, и постфиксную, когда операция записывается после операнда:

```
int x = 0, y = 0;
cout << x++; // Вывод 0, увеличение x на 1
cout << ++y; // Увеличение y на 1, вывод 1
cout << x << “ “ << y; // Вывод 1 1
```

В префиксной форме сначала изменяется операнд, а затем его значение становится результирующим значением выражения, а в постфиксной форме значением выражения является исходное значение операнда, после чего он изменяется.

1.7. Управляющие структуры и инструкции языка C++

При хаотическом программировании возникают очень серьезные отрицательные последствия:

- затруднено восприятие программы или алгоритма;
- сложно производить отладку программы и ее тестирование;
- практически невозможна модификация программы;
- сложно организовать коллективную работу над общей задачей.

Все это привело к созданию определенных методологий в программировании. И, пожалуй, первым таким методом является структурное программирование:

- программа должна реализовываться поэтапно, при этом реализация каждого этапа не должна вызывать затруднений;
- сложная задача должна разбиваться на более простые, легко реализуемые части;
- логика программы должна опираться на минимальное число достаточно простых базовых управляющих структур.

Управляющие структуры необходимы для управления хода выполнения программы. В языке программирования C++ имеются три категории управляющих инструкций:

1. Инструкция ветвления (выбора):
    - `if` – условная конструкция;
    - `switch` – инструкция множественного выбора.
2. Итерационные (циклические) инструкции:
    - `while` – цикл с предусловием;
    - `do while` – цикл с постусловием;
    - `for` – итерационный цикл.

3. Инструкции перехода:
    - `break` – прекращение выполнения циклических инструкций и инструкции `switch`;
    - `continue` – переход к следующей итерации цикла;
    - `return` – прекращение выполнения функции;
    - `goto` – переход по метке.

В некоторых случаях структуризация алгоритмов может привести к появлению в них определенной избыточности, но это полностью оправдывается достоинствами структурированных алгоритмов.

1.8. Условная инструкция

Условная инструкция `if` позволяет выбрать одно из двух направлений выполнения программы.

Имеются две формы записи этой инструкции. Формы записи инструкции `if` представлены на рис. 1.1-1.2.

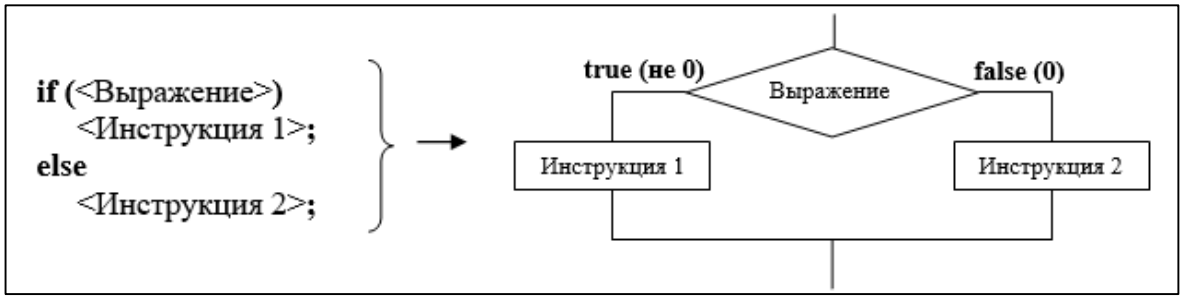


Рисунок 1.1 – первая форма записи инструкции `if`

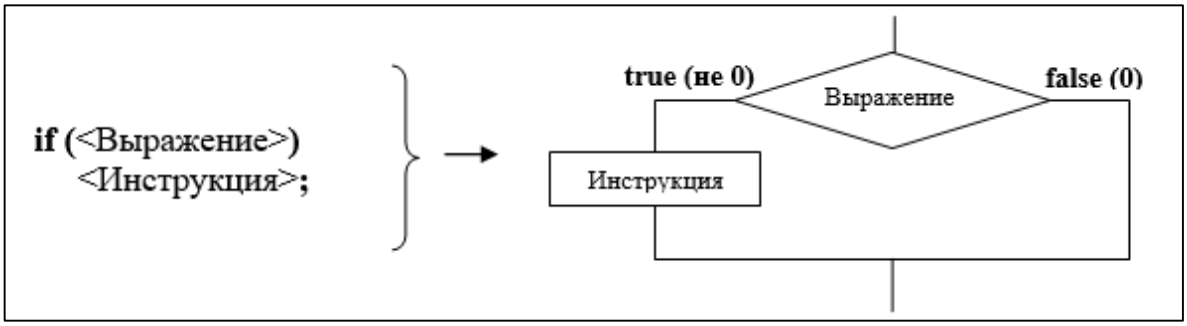


Рисунок 1.2 – Вторая форма записи инструкции `if`

Под термином `<Инструкция>` может пониматься несколько последовательных инструкций. Инструкция может быть пустой.

Может быть ситуация, когда в блоке одной уловной конструкции, имеется еще одна или несколько условных конструкций. Такие инструкции называются вложениями. Количество уровней вложения `if`-инструкций в языке C++ ограничено 256 уровнями.

Подобные схемы вложения можно использовать для множественного выбора, однако это сильно ухудшает читабельность кода. Для реализации схемы множественного выбора больше подходит инструкция множественного выбора.

1.9. Инструкция множественного выбора

Инструкция множественного выбора `switch` служит для ветвления программы во многих направлениях. Форма записи этой инструкции:

```
switch (<Выражение>)
    {
         case <Константа 1>:
              <Последовательность инструкций 1>
              break;
         case <Константа 2>:
              <Последовательность инструкций 2>
              break;
         …….
         case <Константа N>:
              <Последовательность инструкций N>
              break;
         default:
              <Последовательность инструкций>
    }
```

При совпадении значения выражения со значением одной из констант будет выполнена соответствующая этой ветви последовательность инструкций. Инструкция `break` осуществляет прерывание выполнения инструкции switch и управление передается следующему за `switch`-инструкцией оператору. Если значение выражения не совпадает ни с одной из констант, то будут выполнены инструкции ветви `default`, сама же эта ветвь не обязательна.

Значение выражения в инструкции `switch` обязательно должно быть либо целого, либо символьного типа, вещественные значения не допускаются.

1.10. Цикл с предусловием и постусловием

Цикл с предусловием реализует структурную схему, приведенную на рис. 1.3, и имеет вид:

```
while (выражение) тело цикла;
```

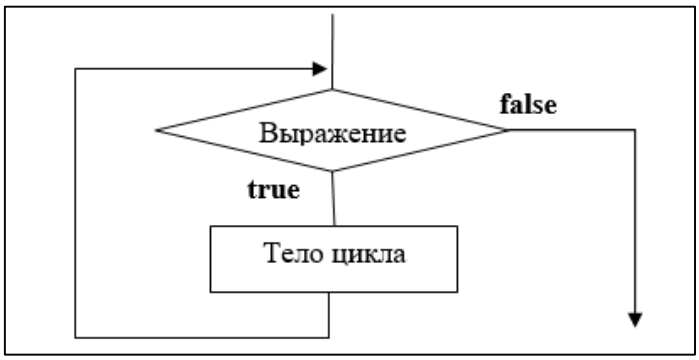


Рисунок 1.3 – Схема цикла с предусловием

Выражение в этой инструкции может быть любого типа, значение которого можно трактовать как значение логического типа данных. Оно определяет условие повторения тела цикла, представленного простым или составным оператором. Выполнение оператора начинается с вычисления выражения. Если выражение истинно (**true** или не равно 0), то тело цикла выполняется вновь, если же ложно (**false** или 0), то цикл заканчивается и управление передается следующей за циклом инструкции.

Для того чтобы цикл начал выполняться, необходимо перед началом цикла выполнить инициализацию его параметров так, чтобы значение выражения соответствовало значению **true** или не было равно 0.

Неправильное использование этой инструкции может привести к образованию бесконечного цикла. Такая ситуация может возникнуть, если значение выражение не меняется в процессе выполнения цикла, или не возможна ситуация остановки цикла. Чтобы избежать подобного, нужно предусмотреть ситуации с зацикливанием, либо использовать принудительное завершение цикла с помощью инструкции `break`.

Пример, в котором используется цикл с предусловием:

```
int n = 0;
while (true) // Пока true
{
      cin >> n; // Ввод n
      if (cin.good() == false) { // Если ввод некорректный
```

```
            cin.clear();
            while (cin.get() != '\n');
      }
      else break; // Если код корректный – завершить выполнение цикла
}
cout << n << endl;
```

Цикл с постусловием реализует схему, приведенную на рис. 1.4, и имеет вид:

```
do тело цикла while (выражение)
```

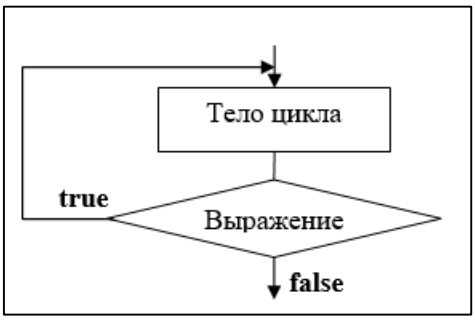


Рисунок 1.4 – Схема цикла с постусловием

Сначала выполняется простой или составной оператор, составляющий тело цикла, а затем вычисляется выражение. Если оно истинно (не равно **false**), тело цикла выполняется еще раз. Цикл завершается, когда выражение станет равным **false** или в теле цикла будет выполнен какой-либо оператор передачи управления. Тип выражения должен быть арифметическим или приводимым к нему.

Цикл с постусловием следует использовать в случае, если необходимо выполнить тело цикла один или более раз. Ниже приведен пример с использованием цикла с постусловием:

```
int n;
do
{
      cin >> n;
} while (n != 0);
      cout << n << endl;
```

1.11. Итерационный цикл

Цикл с параметром реализует схему, приведенную на рис. 1.5, и имеет следующий формат:

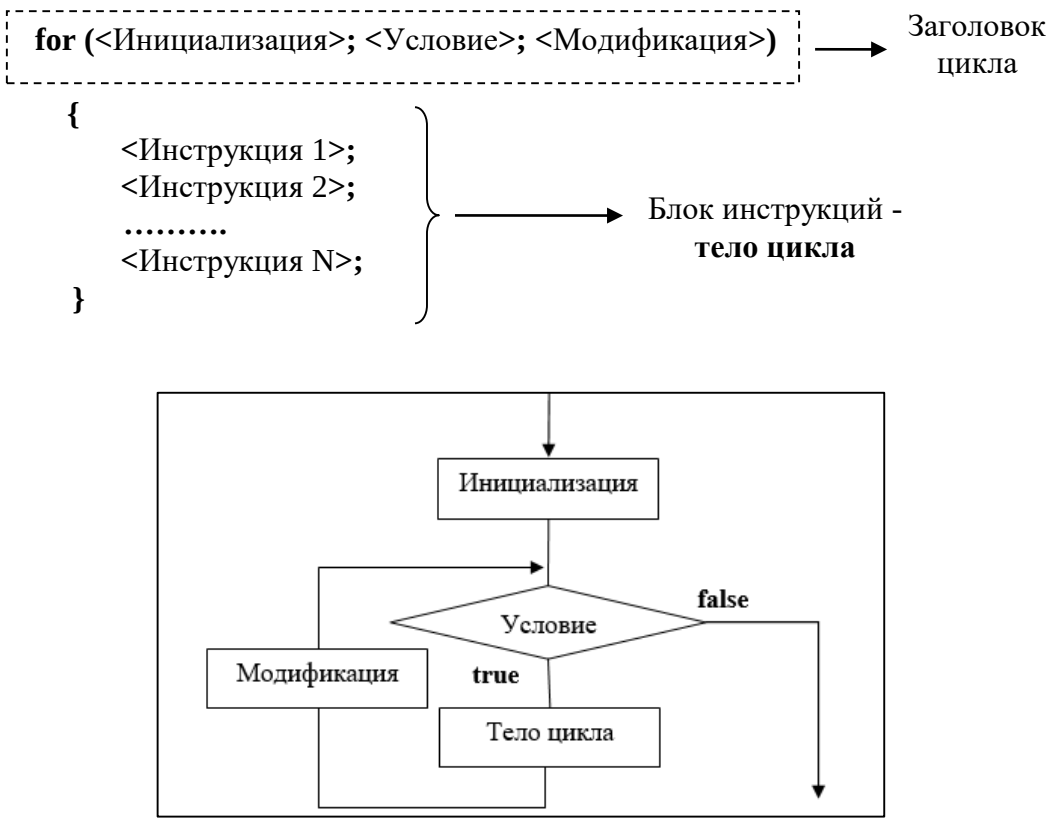


Рисунок 1.5 – Схема цикла с параметром

При запуске цикла однократно выполняется `<Инициализация>` параметра или параметров цикла, после чего осуществляется проверка `<Условия>`, которое определяет необходимость выполнения тела цикла. После окончания выполнения инструкций тела цикла, на каждой итерации выполняется `<Модификация>` параметра (параметров) цикла и снова проверяется `<Условие>`. Так продолжается до тех пор, пока `<Условие>` не станет ложным.

Разделы в заголовке цикла разделяются символом ‘;’. Причем любая из частей оператора может быть опущена. При этом точки с запятой должны остаться. Отдельные элементы разделов отделяются друг от друга символом ‘,’.

Пример использования цикла с параметром представлен в листинге 1.3.

Тела циклов могут содержать любые инструкции языка C++, в том числе и другие циклы. Подобные конструкции называются вложенными циклами. Использование вложенных циклов является весьма распространенным приемом программирования при решении очень многих задач.

Листинг 1.3 – Простейшая пузырьковая сортировка

```
/* bubbleSort.cpp : Этот файл содержит функцию "main". Здесь начинается и заканчивается
выполнение программы */

#include <iostream>
#include <ctime>

using namespace std;

int main()
{
	setlocale(LC_ALL, "Russian");
	const int N = 10;
	srand(time(0));
	int arr[N], inter;
	cout << "Исходный массив данных: ";
	for (int i = 0; i < N; i++) // Цикл, заполняющий массив случайными числами
	{
		arr[i] = rand() % 9;
		cout << arr[i] << " "; }
    for (int j = N - 1; j >= 0; j--) // Цикл, реализующий пузырьковую соритровку
	{
		for (int i = 0; i < j; i++) // Вложенный цикл
		{
			if (arr[i] > arr[i + 1])
			{
				inter = arr[i + 1];
				arr[i + 1] = arr[i];
				arr[i] = inter;
			}
		}
	}
	cout << endl << "Отсортированный массив данных: ";
	for (int i = 0; i < N; i++)
	{
		cout << arr[i] << " ";
	}
	cout << endl;
	return 0;
}
```

1.12. Задание к практической работе

Необходимо разработать алгоритм и написать программу, которая позволяет:

1) Сравнить два числа, не используя условный оператор `if` (пользователь сам вводит числа);

2) Определить чётное ли число ввел пользователь, не используя условный оператор `if`;
3) Вводит дату через N дней. Подразумевается, что год не високосный, а отчет начинается с 1 января.

Далее необходимо выполнить индивидуальное задание (табл. 1.4) и сделать вывод по проделанной работе.

Таблица 1.4 – Индивидуальные задания

Номер варианта	Задание
1	Пользователь вводит год (например, 1886). Необходимо преобразовать его в текст и вывести на экран (одна тысяча восемьсот восемьдесят шестой год).
2	Пользователь вводит год рождения. Необходимо преобразовать его в текст и вывести на экран.
3	Пользователь вводит два числа. Необходимо подсчитать разницу этих чисел и вывести на экран. Причем выражение вместе с ответом необходимо вывести в числовом и текстовом видах.
4	Пользователь вводит два числа. Необходимо подсчитать сумму этих чисел и вывести на экран. Причем выражение вместе с ответом необходимо вывести в числовом и текстовом видах.
5	Пользователь вводит два числа. Необходимо подсчитать произведение этих чисел и вывести на экран. Причем выражение вместе с ответом необходимо вывести в числовом и текстовом видах.

ПРАКТИЧЕСКАЯ РАБОТА №2
ТИПЫ ДАНЫХ И ИХ ВНУТРЕНЕЕ ПРЕДСТАВЛЕНИЕ В ПАМЯТИ

Цель работы: знакомство с внутренним представление различных типов данных, используемых компьютером при их обработке. Научиться работать с побитовыми операциями.

2.1. Понятие типов данных

Любая программа предназначена для обработки данных. Данные бывают различного типа, причем хранятся и обрабатываются они по-разному. Данные бывают числовыми, текстовыми, аудио и видео и т.д. Однако, несмотря на многообразие видов информации (данных), внутреннее машинное представление её едино. Любые данные хранятся в памяти компьютера в виде двоичных кодов.

Память компьютера представляет собой непрерывную последовательность двоичных ячеек (рис. 2.1), каждая из которых может находится в двух состояний, которые можно условно обозначить 0 и 1. Каждая такая двоичная ячейка называется битом. Последовательность, условно разбитая по 8 бит, называется байтами. Из этого следует, что 1 байт = 8 бит. Байт в свою очередь является основной единицей измерения объема памяти.

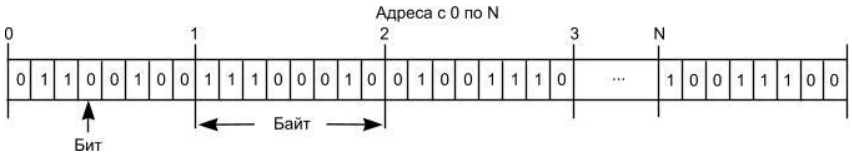


Рисунок 2.1 – Условное представление памяти компьютера

В любом алгоритмическом языке каждая константа, переменная, результата вычисления выражения или функции должны иметь определённый тип, который определяет:

- характер данных (знаковое или число без знака, целое или с дробной частью, последовательность символов или одиночный символ и т.д.);
- объем памяти, который занимают в памяти эти данные;
- диапазон или множество возможных значений;
- операции и функции, которые можно применять к величинам этого типа.

Опираясь на данные характеристики, программист выбирает тип каждой величины, которая используется в программе для представление реальных объектов. В разных языках программирования определены разные наборы типов данных, но, в целом, типы данных можно разделить на две группы: простые и структурированные. Простые типы данных представляют неразделимые данные, которые не имеют внутренней структуры (числа, символы и т.д.). Структурированные типы данных имеют иногда достаточно сложную внутреннюю структуру. Структурированные типы строятся на основе простых типов данных.

Также все типы данных можно разделить на предопределенные и пользовательские. Предопределенные изначально встроены в язык программирования. Пользовательские типы данных определяются программистом.

В языке C++ определено шесть основных типов данных для представления целых, вещественных, символьных и логических величин. На основании этих типов программист может вводить описание составных типов. К ним относятся массивы, перечисления, функции, структуры, ссылки, указатели, объединения и классы.

Основные типы данных часто называют арифметическими, т.к. их можно использовать в арифметических операциях. Для описания основных типов определены следующие ключевые слова:

- `int` (целый);
- `char` (символьный);
- `bool` (логический);
- `float` (вещественный);
- `double` (вещественный тип с двойной точностью).

Первые три типа называют целочисленными, последние два – типами с плавающей точкой. Код, который формирует компилятор для обработки целых величин, отличается от кода для величин с плавающей точкой.

Существует четыре спецификатора типа, уточняющих внутреннее представление и диапазон значений стандартных типов:

- `short` (короткий);

- long (длинный);
- signed (знаковый);
- unsigned (беззнаковый).

Спецификаторы добавляются слева к названию типа. Сами же спецификаторы могут идти в произвольном порядке: unsigned short int эквивалентно short unsigned int. При этом допустимы не все сочетания спецификаторов и типов данных.

2.2. Целый тип (int)

Размер типа int зависит от компьютера и компилятора. Для 16-разрядного процессора под величины этого типа отводится 2 байта. Для 32-разрядного – 4 байта. Узнать, сколько отводиться под тот или иной тип данных или объект памяти, можно при помощи операции sizeof. Пример использования этой операции представлен в листинге 2.1. Операция sizeof вычисляет размер в байтах и имеет две формы:

```
sizeof выражение;
sizeof ( тип );
```

Листинг 2.1 – Использование операции sizeof

```
/* sizeofInt.cpp: Этот файл содержит функцию "main". Здесь начинается и заканчивается
выполнение программы */

#include <iostream>

using namespace std;

int main()
{
	int a = 26, b = 35;
	short int c = 5;
	cout << sizeof(int) << endl; // 4
	cout << sizeof a << " " << sizeof b << " " << sizeof (a + b) << endl; // 4 4 4
	cout << sizeof c << endl; // 2
	cout << sizeof(long int) << endl; // 4
	return 0;
}
```

Спецификатор short перед именем указывает компилятору, что под число требуется отвести 2 байта. Спецификатор long означает, что целая величина будет занимать 4 байта.

Спецификатор `unsigned` позволяет представлять только положительные числа, при этом старший разряд рассматривается как часть кода числа. Таким образом, диапазон значений типа int зависит от спецификаторов. По умолчанию все целочисленные типы считаются знаковыми, то есть спецификатор `signed` можно опускать.

Целые константы литералы можно задавать в трех форматах: в десятичном, восьмеричном и в шестнадцатеричном. В восьмеричном формате сначала записывается 0, а за ним восьмеричные разряды самого числа. В шестнадцатеричном формате значащим разрядам числа должны предшествовать символы 0x ил 0X.

Внутреннее представление величин целого типа – целое число в двоичном коде. При использовании спецификатора `signed` старший бит числа интерпретируется как знаковый (0 – положительное число, 1 – отрицательное). Для кодирования целых чисел со знаком применяется прямой, обратный и дополнительный коды. Представление положительных и отрицательных чисел в прямом, обратном и дополнительном кодах отличается.

В прямом коде в знаковый разряд помещается цифра 1, а в разряды цифровой части числа – двоичный код его абсолютной величины. Прямой код числа −3 (для 16-разрядного процессора) представлен на рис 2.2.

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Знак числа

Рисунок 2.2 – Прямой код числа −3

Обратный код получается инвертированием всех цифр двоичного кода абсолютной величины, включай разряд знака: нули заменяются единицами, единицы – нулями. Прямой код можно преобразовать в обратный, инвертировав все значения всех битов (кроме знакового). Обратный код числа −3 представлен на рис 2.3.

16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0

Знак числа

Рисунок 2.3 – Обратный код числа −3

Дополнительный код получается образованием обратного кода с последующим прибавлением единицы к его младшему разряду. Дополнительный код числа −3 представлен на рис 2.4.

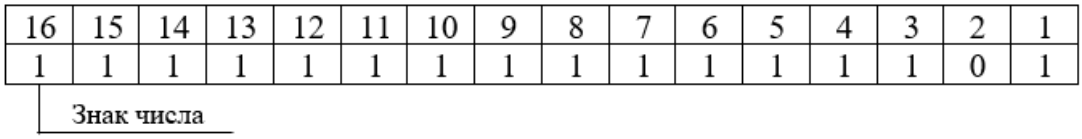


Рисунок 2.4 – Обратный код числа −3

Увидеть, каким образом тип данных представляется на компьютере, можно при помощи логических операций: побитового сдвига (<<) и поразрядной конъюнкции (&). Пример использования данных операций представлен в листинге 2.2.

Листинг 2.2 – Представление int (32 бита) в памяти

```
/* int.cpp: Этот файл содержит функцию "main". Здесь начинается и заканчивается выполнение
программы */

#include <iostream>

using namespace std;

int main()
{
	int value = -127; // Значение числа
	unsigned int order = 32; // Количество разрядов
	unsigned int mask = 1 << order - 1; // Маска побитового сравнения
	for (int i = 1; i <= order; i++)
	{
		putchar(value & mask ? '1' : '0');
		value <<= 1; // Побитовый сдвиг числа
		if (i % 8 == 0)
		{
			putchar(' ');
		}
		if (i % order - 1 == 0)
		{
			putchar(' ');
		}
	}
	return 0;
}
```

В консоль будет выведено: `1 1111111 11111111 11111111 10000001`.

При сдвиге вправо для чисел без знака позиции битов, освобожденные при операции сдвига, заполняются нулями. Для чисел со знаком бит знака используется для заполнения освобожденных позиций битов. Другими словами, если число

является положительным, используется 0, если число является отрицательным, используется 1.

При сдвиге влево позиции битов, освобожденных при операции сдвига, заполняются нулями. Сдвиг влево является логическим сдвигом (биты, сдвигаемые с конца, отбрасываются, включая бит знака).

2.3. Вещественные типы данных

Стандарт C++ определяет три типа данных для хранения вещественных значений: `float`, `double` и `long double`. Все эти типы предназначены для представления отрицательных и положительных значений (спецификатор `unsigned` к ним не применим) в различных диапазонах. В листинге 2.3 представлен пример использования `sizeof` применительно к вещественным типам данных.

Листинг 2.3 – Операция `sizeof` с вещественными типами данных

```
/* sizeofFloatAndDouble.cpp: Этот файл содержит функцию "main". Здесь начинается и
заканчивается выполнение программы */

#include "pch.h"
#include <iostream>

using namespace std;

int main()
{
	float a = 3.14;
	double b = 3.14;
	long double c = 3.14;
	cout << sizeof a << endl; // 4
	cout << sizeof b << endl; // 8
	cout << sizeof c << endl; // 8
	return 0;
}
```

В консольных приложениях Windows тип данных `long double` занимает в памяти 8 байт, то есть ничем не отличается от типа `double`. В иных случаях спецификатор `long` перед именем типа `double` указывает, что под величину отводится 10 байт.

Дробная часть отделяется от целой части точкой, а не запятой. По умолчанию вещественные константы трактуются как константы типа double. Для явного указания

другого типа вещественной константы можно использовать суффиксы `F(f)` – тип `float` или `L(l)` – тип `long double`.

Вещественные типы данных хранятся в памяти компьютера иначе, чем целочисленные. Внутреннее представление вещественного числа состоит из двух частей – мантиссы и порядка. На рис. 2.5 представлен пример вещественного числа.

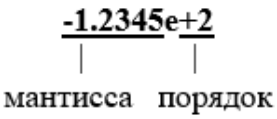


Рисунок 2.5 – Пример вещественного числа

Для 32-разрядного процессора для `float` (рис. 2.6) под мантиссу отводится 23 бита, под экспоненту – 8, под знак – 1. Для `double` под мантиссу отводится 52 бита, под экспоненту – 11, под знак – 1.

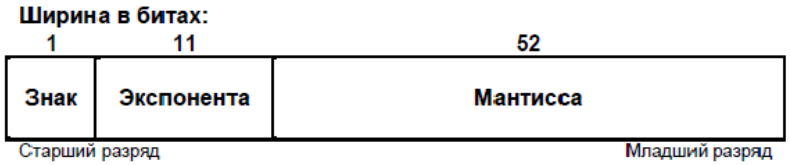


Рисунок 2.6 – Представление `float` в памяти

Увидеть, каким образом вещественные типы данных представляются в компьютере немного сложнее. Логические операции, которые использовались с int, для вещественных типов данных не подходят. Но это ограничение можно легко обойти, использовав объединения.

Объединения – это две или более переменных расположенных по одному адресу (они разделяют одну и ту же память). Объединения определяются с использованием ключевого слова union. Пример использования объединения представлен в листинге 2.4.

Объединения не могут хранить одновременно несколько различных значений, они позволяют интерпретировать несколькими различными способами содержимое одной и той же области памяти.

С объединениями нужно быть острожным. Вся работа с памятью требует грамотного подхода. Более подробно с объединениями можно будет ознакомиться при изучении структур. Пока что объедения будут служить инструментом для работы с `float` и `double`.

Листинг 2.4 – Пример использования union

```
/* union.cpp: Этот файл содержит функцию "main". Здесь начинается и заканчивается
выполнение программы */

#include <iostream>

using namespace std;

int main()
{
	union {
		int tool;
		float numb_f = 3.14;
	};
	cout << tool << endl;    // 1078523331
	cout << numb_f << endl;  // 3.14
	tool = tool >> 1;        // побитовый сдвиг вправо
	cout << tool << endl;    // 5392261665
	cout << numb_f;          // 1.3932e-19
	return 0;
}
```

Подобные манипуляции возможны благодаря тому, что `int` и `float` занимают 4 байта. Проводя манипуляции над `tool`, мы изменяем значение `numb_f`. Таким образом, алгоритм, который использовался для представления в памяти int может использоваться и для `float`.

Алгоритма представления `double` немного отличается. Под вещественное число с двойной точностью отводиться 8 байт, в то время как под int всего 4 байта. Но и это ограничение можно легко обойти. Так как данные любой линейной структуры в память записываются последовательно (друг за другом), можно использовать массив из двух `int`, под который будет отведено 8 байт.

2.4. Логический тип данных

Величины логического типа могут принимать значения **true** и **false**, которые являются зарезервированными словами. Внутренняя форма представления значения **false** – 0. Любое другое значение интерпретируется как true. При преобразовании к целому типу true имеет значение 1.

В памяти переменные этого типа занимают 1 байт.

2.5 Символьный тип данных

Для обозначения этого типа используется ключевое слово `char`.

Под величину символьного типа отводится количество байт, достаточное для размещения любого символа из набора символов для данного компьютера, что и обусловило название типа. Как правило, это 1 байт. Тип char, как и другие целые типы, может быть со знаком и без знака. Спецификаторы `short` и `long` к этому типу данных не применяются. В табл. 2.1 представлены диапазоны значений простых типов и их размер в байтах.

Таблица 2.1 – Диапазоны значений простых типов данных

Тип	Диапазон значений	Размер в байтах
`bool`	**true** и **false**	1
`signed char`	−128 … 127	1
`unsigned char`	0 … 255	1
`signed short int`	−32 768 … 32 767	2
`unsigned short int`	0 … 65535	2
`signed long int`	−2 147 483 648 … 2 147 483 647	4
`unsigned long int`	0 … 4 294 967 295	4
`float`	3.4e−38 … 3.4e+38	4
`double`	1.7e−308 … 1.7e+308	8
`long double`	3.4e−4932 … 3.4e+4932	10

Более подробно символьный тип данных будет рассмотрен в 4 практической работе.

2.6 Задание к практической работе

Разработать алгоритм и написать программу, которая позволяет:

1) Вывести, сколько памяти (в байтах) на вашем компьютере отводится под различные типы данных со спецификаторами и без: `int`, `short int`, `long int`, `float`, `double`, `long double`, `char` и `bool`.
2) Вывести на экран двоичное представление в памяти (все разряды) целого числа (`int`, `short int`, `unsigned int`). При выводе необходимо визуально обозначить знаковый разряд и значащие разряды отступами ли цветом.

3) Вывести на экран двоичное представление в памяти (все разряды) типа `float`. При выводе необходимо визуально обозначить знаковый разряд мантиссы, знаковый разряд порядка (если есть), мантиссу и порядок.
4) Вывести на экран двоичное представление в памяти (все разряды) типа double. При выводе необходимо визуально обозначить знаковый разряд мантиссы, знаковый разряд порядка (если есть), мантиссу и порядок.
5) Выполнить индивидуальное задание, варианты которого представлены в табл. 2.2.
6) Вывести значения, после изменения его двоичного представления в памяти.

Сделать вывод по проделанной работе.

Таблица 2.2 – Индивидуальное задание

Номер варианта	Задание
1	Возможность изменить произвольные биты на 1 или 0 по указанию пользователя (номер бита и значения вводятся с клавиатуры) для всех затронутых типов данных.
2	Возможность менять местами заданные пользователем биты, номера которых вводятся пользователем с клавиатуры для всех затронутых типов данных.
3	Возможность неоднократно инвертировать все биты для всех затронутых типов данных.
4	Возможность инвертировать все биты, кроме битов, номера которых вводятся пользователем с клавиатуры, для всех затронутых типов данных.
5	Возможность изменить случайным образом состояние бит, номера которых вводятся пользователем с клавиатур, для всех затронутых типов данных.
6	Возможность зеркально отобразить все биты, начиная с бита, номер которого вводится пользователем с клавиатуры, для всех затронутых типов данных.
7	Возможность сдвинуть все биты, равные единице, вправо для всех затронутых типов данных.
8	Возможность сдвинуть все биты, равные единице, влево для всех затронутых типов данных.

Продолжение таблицы 2.2

9	Возможность для всех затронутых типов данных попарно поменять местами группы битов, размерность группы битов вводит пользователь с клавиатуры,
10	Если число отрицательное, возможность для всех затронутых типов данных случайным образом поменять значение четных битов, если положительное, то случайным образом поменять значения нечетных битов.
11	Возможность сдвинуть все биты влево на введенное число пользователем для всех затронутых типов данных.
12	Возможность сдвинуть все биты вправо на введенное число пользователем для всех затронутых типов данных.
13	Если число отрицательно, то возможность поменять значение четных битов на 0, если положительное, то поменять значение нечетных битов на 1.
14	Если число отрицательное, то возможность поменять значения четных битов на 1, если положительное, то поменять значения нечетных битов на 0.
15	Возможность присвоить биту, номер которого вводит пользователь, значение случайного бита введенного пользователем числа, для всех затронутых типов данных.

ПРАКТИЧЕСКАЯ РАБОТА №3
ОДНОМЕРНЫЕ СТАТИЧЕСКИЕ МАССИВЫ

Цель работы: изучение структуры одномерных массивов, обработка данных одномерных массивов. Изучение различных видов сортировок. Проведение временной оценки различных действий с массивами.

3.1. Понятие массива

При использовании простых переменных каждой области памяти для хранения данных соответствует свое имя. Если с группой величин одинакового типа требуется выполнить однообразные действия, им дают одно имя, а различают по порядковому номеру (индексу). Это дает возможность компактно записать множество операций с использованием циклов.

Массив представляет собой индексированную последовательность однотипных элементов с заранее определенным количеством элементов. Наглядно одномерный массив можно представить, как набор пронумерованных ячеек, в каждой из которых содержится определенное значение.

Все массивы можно разделить на две группы: одномерные и многомерные. Описание массива в программе отличается от объявления обычной переменной наличием размерности массива, которая задается в квадратных скобках после имени. В листинге 3.1 представлен пример работы с одномерным массивом.

Элементы массива нумеруются с нуля. При описании массива используются те же модификаторы (класс памяти, `const` и инициализатор), что и для простых переменных.

Инициализировать массив можно и другим более простым способом: инициализирующие значения записываются в фигурных скобках. Значения элементам присваиваются по порядку. Если элементов в массиве больше, чем инициализаторов, элементы, для которых значения не указаны, обнуляются:

```
int arr[4] = {3, 2, 1}; // arr[0] = 3, arr[1] = 2, arr[2] = 1, arr[3] = 0
```

Размерность массива вместе с типом его элементов определяет объем памяти, необходимый для размещения массива, которое выполняется на этапе компиляции,

поэтому размерность должна быть задана целой положительной константой или константным выражением.

Листинг 3.1 – Пример заполнения одномерного массива

```
/* arr.cpp: Этот файл содержит функцию "main". Здесь начинается и заканчивается выполнение
программы */

#include <iostream>
#include <ctime>

using namespace std;

int main()
{
	setlocale(LC_ALL, "Russian");
	const int N = 10; // Задаём размерность массива
	srand(time(0));
	int arr[N]; // Объявляем массив arr размерности N
	cout << "Исходный массив данных: ";
	for (int i = 0; i < N; i++) // Цикл, заполняющий массив случайными числами
	{
		arr[i] = rand() % 9; // Заполняем массив случайными числами от 0 до 9
		cout << arr[i] << " "; // Выводим элемент массива на экран
	}
	return 0; }
```

Если при описании массива не указана размерность, массив обязательно должен быть инициализирован. Компилятор сам определит размерность массива по количеству элементов:

```
int arr[] = {7, 6, 5, 4, 3, 2, 1}; // Всего 7 элементов
```

Для доступа к элементу массива после его имени указывается номер элемента (индекс) в квадратных скобках. Важно помнить, что при обращении к элементам массива автоматический контроль выхода индекса за границу массива не производится, это может привести к ошибкам.

Обмен элементов массива осуществляется через буферную переменную (рис. 3.1.) либо через функцию `swap(a, b)`.

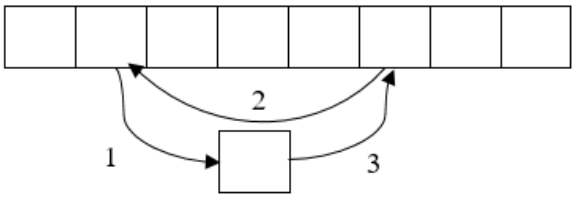


Рисунок 3.1 – Обмен местами элементов массива

Определить, сколько памяти выделено под массив, можно с помощью операции `sizeof`. Необходимо определить, сколько выделяется памяти на 1 элемент массива и умножить на количество элементов массива.

Принцип действия схож с задачей про переливание воды из одного сосуда в другой. Чтобы поменять жидкость сосудов местами, нужно использовать еще один сосуд, который будет временно содержать жидкость одного из сосудов.

3.2. Пузырьковая сортировка массива (bubble sort)

Сортировка – процесс размещения элементов заданного множества объектов в определенном порядке. Когда элементы отсортированы, их проще найти, производить с ними различные операции. Сортировка напрямую влияет на скорость алгоритма, в котором нужно обратиться к определенному элементу массива.

Простейшая из сортировок – сортировка обменом (пузырьковая сортировка). Вся суть метода заключается в сравнивании попарном сравнении элементов и последующем обмене. Таким образом, если следующий элемент меньше текущего, то они меняются местами, максимальный элемент массива постепенно смещается в конец массива, а минимальный – в начало. Один полный проход по массиву может гарантировать, что в конце массива находится максимальный элемент.

Затем процесс повторяется до тех пор, пока вся последовательность не будет упорядочена. Важно заметить, что после первого прохода по массиву, уже имеется один упорядоченный элемент, он стоит на своем месте, и менять его не надо. Таким образом на следующем шаге будут сравниваться N-1 элемент. Пример работы алгоритма bubble sort представлен на рис. 3.1.

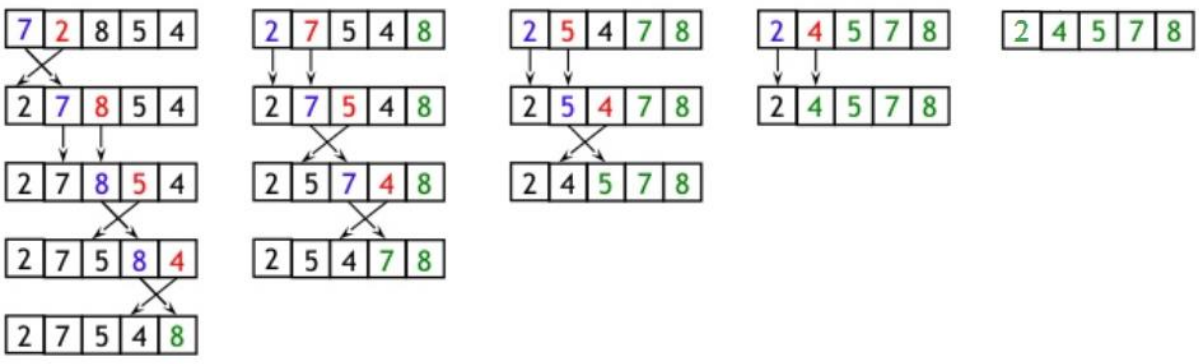


Рисунок 3.2 – Пример работы алгоритма bubble sort

Очевидно, что хуже всего алгоритм будет работать, когда на вход подается массив, отсортированный в обратную сторону (от большего меньшего). Быстрее же всего алгоритм работает с уже отсортированным массивом.

Но стандартный алгоритм пузырьковой сортировки предполагает полный циклический проход по массиву. Если изначально подается упорядоченная последовательность, то работа алгоритма все равно продолжиться. Исправить это можно, добавив условие проверки: если на текущей итерации ни один элемент не изменил свой индекс, то работа алгоритма прекращается.

3.3. Шейкер-сортировка массива (shaker sort)

Shaker sort – модификация пузырьковой сортировки. Принцип работы этой сортировки аналогичен bubble sort: попарное сравнение элементов и последующий обмен местами. Но имеется существенное отличие. Как только максимальный элемент становится на свое место, алгоритм не начинает новую итерацию с первого элемента, а запускает сортировку в обратную сторону. Алгоритм гарантирует, что после выполнения первой итерации, минимальный и максимальный элемент будут в начале и конце массива соответственно.

Затем процесс повторяется до тех пор, пока массив не будет отсортирован. За счет того, что сортировка работает в обе стороны, массив сортируется непорядок быстрее. Очевидным примером этого был бы случай, когда в начале массива стоит максимальный элемент, а в конце массива – минимальный. Shaker sort справится с этим за 1 итерацию, при условии, что другие элементы стоят на правильном месте.

Пример работы shaker sort представлен на рис. 3.2.

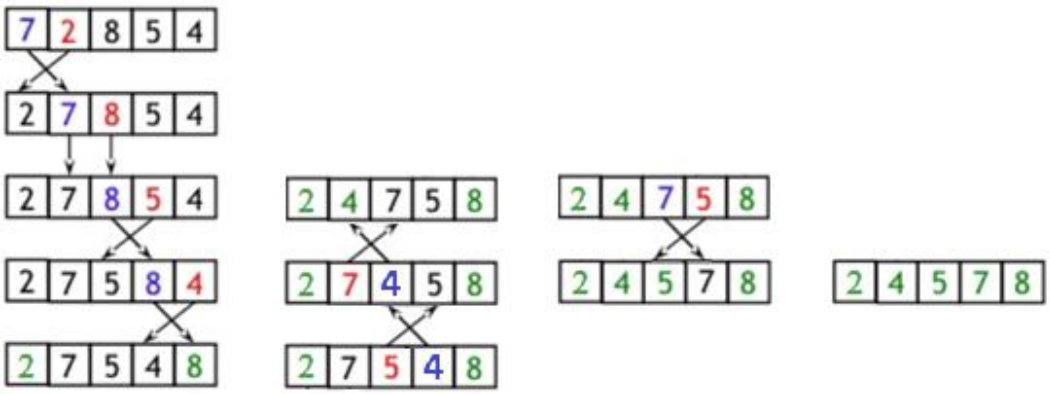


Рисунок 3.3 – Пример работы алгоритма shaker sort

Кажется, что bubble sort теряет свою эффективность по сравнению с shaker sort. Сортировка проходит в массиве в обоих направлениях, а не только от его начала к концу. Но в работе с большими массивами преимущество шейкер-сортировки уменьшается как раз из-за использования двух циклов.

3.4. Сортировка массива расчёской (comb sort)

Очевидный недостаток bubble и shaker sort заключается в том, элементы переставляются максимум на одну позицию.

Comb sort (сортировка расческой) – ещё одна модификация сортировки пузырьком. Алгоритм был разработан для специально для случаев, когда минимальные элементы стоят слишком далеко, или максимальные – слишком близко к началу массива. В сортировке расческой переставляются элементы, стоящие на расстоянии.

Оптимально изначально взять расстояние равным длине массива $N - 1$, а далее уменьшать его на определенный коэффициент, который примерно равен 1.247. Когда расстояние станет равно 1, выполняется обычная сортировка пузырьком. Пример работы comb sort представлен на рис. 3.4.

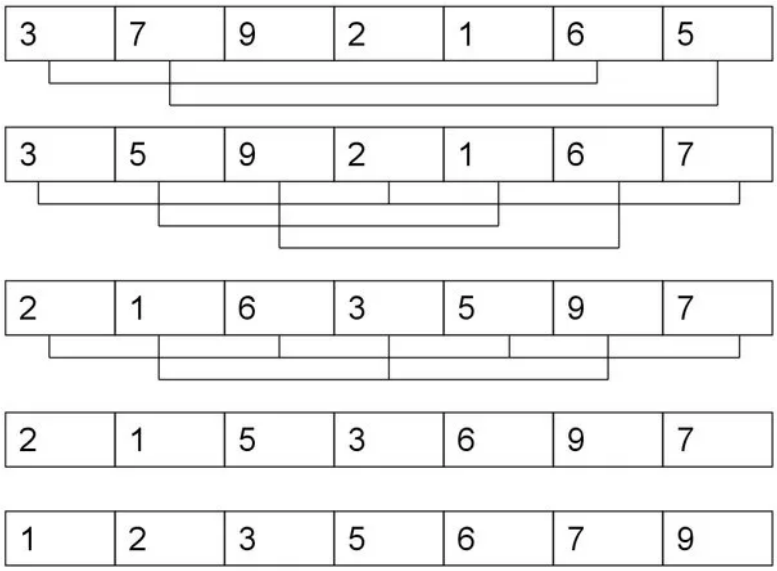


Рисунок 3.4 – Пример работы алгоритма comb sort

Простая идея и несложная идея порождающее впечатляющий результат. Сортировка расческой работает намного быстрее, чем bubble или shaker sort, в

некоторых ситуациях comb sort работает быстрее quick sort. Но данная сортировка обладает одним очевидным минусом – неустойчивость.

3.5. Сортировка массива вставками (insert sort)

Сортировка вставками (insert sort) – алгоритм сортировки, в котором элементы массива просматриваются по одному, и каждый новый элемент размещается в подходящее место среди ранее упорядоченных элементов.

Общая суть сортировки вставками такова:

1) Перебираются элементы в неотсортированной части массива.
2) Каждый элемент вставляется в отсортированную часть массива на то место, где он должен находится.

Сортировка вставками делить массив на 2 части – отсортированную и неотсортированную. С каждым новым элементом отсортированная часть будет увеличиваться, а неотсортированная уменьшаться. Причем найти нужное место для очередного элемента в отсортированном массиве достаточно легко.

Рассмотрим самый простой способ (рис. 3.5). Необходимо пройти массив слева направо и обработать каждый элемент. Слева будет наращиваться отсортированная часть массива, а справа – уменьшаться неотсортированная. В отсортированной части массива ищется точка вставки для очередного элемента. Сам элемент отправляется в буфер, что освобождает место в массиве и позволяет сдвинуть элементы и освободить точку вставки.

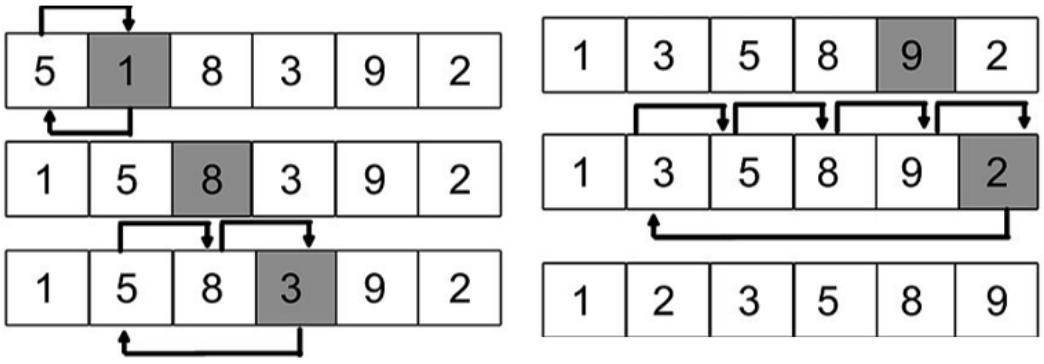


Рисунок 3.4 – Пример работы алгоритма comb sort

Существует множество модификаций сортировки вставками, некоторые из них затрагивают именно способ вставки элемента в отсортированную часть. Одна из

самых лучших модификаций – сортировка простыми вставками с бинарным поиском. Бинарный поиск будет описан позже.

Лучше всего сортировка вставками работает при обработке почти отсортированных массивов. В таком случае insert sort работает быстрее других сортировок.

3.6. Быстрая сортировка массива (quick sort)

Быстрая сортировка (quick sort) – одна из самых быстрых сортировок. Эта сортировка по сути является существенно улучшенной версией алгоритма пузырьковой сортировки.

Общая идея алгоритма состоит в том, что сначала выбирается из массива элемент, который называется опорным. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность. Затем необходимо сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: меньше опорного, раны опорному и больше опорного. Для меньших и больших значений необходимо выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы. Пример работы алгоритма представлен на рис. 3.5.

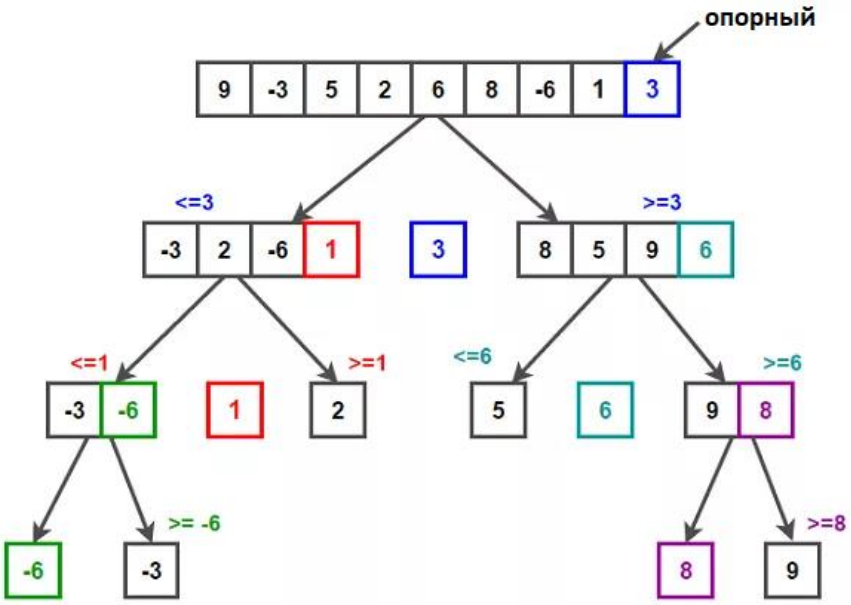


Рисунок 3.5. – Пример работы алгоритма quick sort

На практике массив обычно делят на две части: «меньше опорного» и «равные и большие» или «меньше опорного или равные» и «большие». Такой поход в общем случае эффективнее, ведь упрощается алгоритм разделения.

При том, что это один из самых быстродействующих из алгоритмов, данные алгоритм сортировки неустойчив, а прямая реализация в виде функции с двумя рекурсивными вызовами может привести к ошибке переполнения стека.

3.7. Алгоритм бинарного поиска

Алгоритм бинарного поиска – классический алгоритм поиска в отсортированном массиве, который использует дробление массива на половины. Если элемент, который необходимо найти, присутствует в списке, то бинарный поиск возвращает ту позицию, в которой он был найден.

Рассмотрим простой пример: имеется массив из 100 элементов, упорядоченных по возрастанию от 1 до 100. Было загадано какое-то число, необходимо его назвать. Компьютер имеет три ответа на ваше предположение: верно, число больше, число меньше. Сколько попыток нужно, чтобы ответить правильно? Обычный перебор – наихудшая стратегия. Можно назвать правильный ответ лишь с 100-ой попытки.

Но если начать спрашивать с середины, то ситуация кардинально меняется. Если число больше 50, то необходимо делить правую половину, и следующее предположение – 75, если меньше – 25. Так необходимо продолжать до тех пор, пока не будет названо правильное число. Наибольшее число предположений равняется:

$$k = \lceil \log_2 100 \rceil = 7.$$

Этот результат явно лучше простого перебора. Схожий принцип работы имеет алгоритм бинарного поиска. Пример его работы представлен на рис. 3.6.

Бинарный поиск работает только в топ случае, если список отсортирован. Например, если бы искомое минимальное значение стояло не на своем положенном месте, а на месте максимального элемента, то мы бы откинули его на первой же итерации. Сам алгоритм имеет вид:

1) Определение значения в середине массива (или иной структуры данных). Полученное значение сравнивается с ключом (значением, которое необходимо найти).
2) Если ключ меньше значения середины, то необходимо осуществлять поиск в первой половине элементов, иначе – во второй.
3) Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.
4) Процесс продолжается до тех пор, пока не будет определен элемент, равный значению ключа или не станет пустым интервал для поиска.

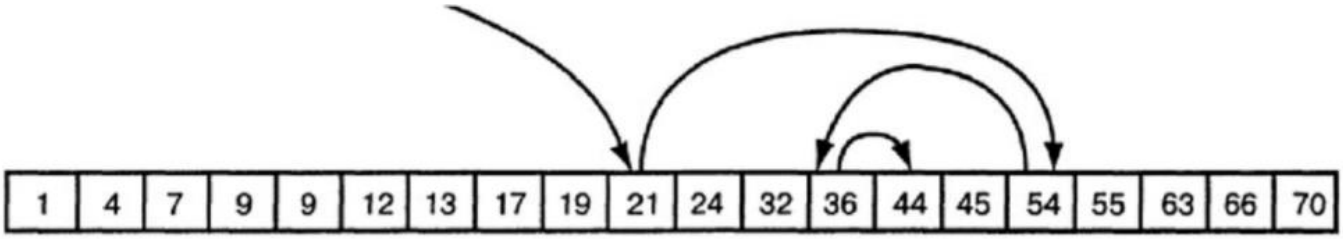


Рисунок 3.6 – Бинарный поиск числа 44

Чтобы уменьшить количество шагов поиска, можно сразу смещать границы поиска на элемент, следующий за серединой отрезка.

3.8. Задание к практической работе

Необходимо написать программу, которая:

1) Создает целочисленный массив размерности $N = 100$. Элементы массивы должны принимать случайное значение в диапазоне от -99 до 99.
2) Отсортировать заданный в пункте 1 элементы массива […] сортировкой. Определить время, затраченное на сортировку, используя библиотеку `chrono`.
3) Найти максимальный и минимальный элемент массива. Подсчитайте время поиска этих элементов в отсортированном массиве и неотсортированном, используя библиотеку `chrono`.
4) Выводит среднее значение (если необходимо, число нужно округлить) максимального и минимального значения. Выводит все числа, которые равны этому значению.

5) Выводит количество элементов в отсортированном массиве, которые меньше числа a, которое инициализируется пользователем.
6) Выводит количество элементов в отсортированном массиве, которые больше числа b, которое инициализируется пользователем.
7) Выводит информацию о том, есть ли введенное пользователем число в отсортированном массиве. Реализуйте алгоритм бинарного поиска. Сравните скорость его работы с обычным перебором.
8) Меняет местами элементы массива, индексы которых вводит пользователь. Выведите скорость обмена, используя библиотеку `chrono`.

Должна присутствовать возможность запуска каждого пункта многократно.

Затем выполните задания по вариантам и сделайте вывод по проделанной работе.

Задание по вариантам: варианты сортировок представлены табл. 3.1, а индивидуальные задания в табл. 3.2.

Таблица 3.1 – Варианты сортировок

Номер варианта	Название сортировки
1	Bubble sort (пузырьковая сортировка)
2	Shaker sort (шейкер-сортировка)
3	Comb sort (сортировка расчёской)
4	Insert sort (сортировка вставками)
5	Quick sort (быстрая сортировка)

Таблица 3.2 – Индивидуальные задания

Номер варианта	Задание
1	Разделите массив из первого задания на 2 массива. Поменяйте значения нечетных элементов 1 массива с четными элементами массива 2. Сделайте все значения 2 массива четными прибавлением или вычитанием единицы. Сравните попарно значения элементов 1 и 2 массивов. Выведите количество элементов, которые больше в 1 массиве, чем во 2.

Продолжение таблицы 3.2

2	Уменьшите каждый четный элемент на значение, введенное пользователем. Каждый четный элемент умножьте на случайную величину в диапазоне от 1 до 9. Выведите на экран сколько чётных элементов имеют четные значения, сколько нечетных элементов имеют нечетные значения. Выведите количество элементов, которые нацело делятся на 1, 2, 3, 4, 5, 6, 7, 8, 9.
3	Самостоятельно изучите и реализуйте сортировку Шелла. Сравните её с сортировкой, которую вы использовали в вашей программе. Выделите достоинства и недостатки данной сортировки.
4	Замените каждый элемент массива суммой значений текущего элементы и следующего (последний = последний + первый). Присвойте каждому элементу случайный номер в массиве. Выведите количество элементов, которые нацело делятся на 1, 2, 3, 4, 5, 6, 7, 8, 9.
5	Самостоятельно изучите и реализуйте сортировку слиянием. Сравните её с сортировкой, которую вы использовали в вашей программе. Выделите достоинства и недостатки данной сортировки.
6	Самостоятельно изучите и реализуйте сортировку простыми вставками с бинарным поиском. Сравните её с сортировкой, которую вы использовали в вашей программе. Выделите достоинства и недостатки данной сортировки.
7	Реализуйте еще две сортировки из табл. 3.1. Проанализируйте их работу, выделите достоинства и недостатки каждой. Определите самую лучшую сортировку.
8	Самостоятельно изучите и реализуйте парную сортировку простыми вставками. Сравните её с сортировкой, которую вы использовали в вашей программе. Проанализируйте её работу, выделите достоинства и недостатки этой сортировки.
9	Уменьшите каждый нечетный элемент на значение, введенное пользователем. Каждый нечетный элемент умножьте на случайную величину в диапазоне от 1 до 9. Выведите количество элементов, которые нацело делятся на 1, 2, 3, 4, 5, 6, 7, 8, 9.

Окончание таблицы 3.2

10	Составьте проверочную работу по сортировкам. Задания проверочной работы: - Отсортируйте предложенный массив. - Определите, сколько итераций, нужно, чтобы отсортировать массив (сортировка по вашему варианту). Обучающий, выполняющий проверочную работу, должен поэтапно объяснить ход действий при вставке и удалении. Нужно вывести ответы к проверочной работе с объяснением каждого действия. Пример: Отсортируйте массив 2 3 1 сортировкой bubble sort. Элемент 2. Сравниваем элемент 2 и 3. Элемент 2 < 3. Обмен не нужен. Массив имеет вид 2 3 1. Массив не отсортирован. Элемент 3. Сравниваем элемент 3 и 1. Элемент 3 > 1. Меняем местами 3 и 1. Элемент 3 является отсортированным. Массив имеет вид 2 1 3. Массив не отсортирован. … Массив имеет вид 1 2 3. Массив отсортирован.

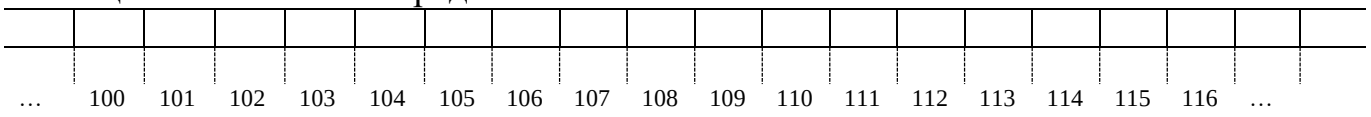
ПРАКТИЧЕСКАЯ РАБОТА №4
УКАЗАТЕЛИ И МНОГОМЕРНЫЕ СТАТИЧЕСКИЕ МАССИВЫ

Цель работы: изучение структуры многомерных статических массивов, обработка данных многомерных массивов. Получение практических навыков работы с указателями. Изучение простейшей арифметики указателей.

4.1. Понятие указателя

Компилятор, обрабатывая оператор определения переменной, выделяет память в соответствии с типом переменной и инициализирует её указанным значением. Все обращения по имени переменной заменяются компилятором на адрес области памяти, в которой хранится значение переменной. Возможно создание собственных переменных, которые будут хранить какой-то адрес памяти. Такие переменные называются указателями.

Таблица 4.1 – Условное представление памяти



Указатели предназначены для хранения адресов областей памяти (табл. 4.1.). В C++ существует три вида указателей:

1) Указатели на объект, который содержит адрес области памяти, хранящей данные определенного типа.
2) Указатели на функцию. Указатель на функцию содержит адрес сегмента кода, по которому располагается исполняемый код функции. Указатели на функции используются для косвенного вызова функции (через обращение к переменной, хранящей её адрес), а также для передачи имени функции в другую функцию в качестве параметра. Указатель функции должен иметь тип «указатель функции, возвращающей значение заданного типа и имеющей аргументы заданного типа.
3) Указатели на `void`. Такой указатель применяется в тех случаях, когда тип объекта, адрес которого нужно хранить, не определен. Указателю типа `void`

можно присвоить значение любого типа, но перед выполнением каких-либо действий его нужно явным образом преобразовать к этому типу.

Для получения адреса какого-либо программного объекта используют оператор &. Предположим, что у нас имеется две переменные: `float A = 3.14` и `double = 3.14` (табл. 4.2). Видно, что обе переменные занимают по 4 байта памяти. Они имеют адреса 101 и 105 соответственно.

Таблица 4.2 – Представление в памяти некоторых переменных

		float A = 3.14				double B = 3.14												
…	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	…

Если использовать оператор &, то он вернет адреса этих переменных. Правда, обычно значение адреса памяти выводится в шестнадцатеричном коде.

Указатели на объекты определяются в программе следующим образом:

```
<тип данных> *<имя переменной>;
```

Тип данных – базовый тип указателя. Имя переменной – идентификатор переменной-указателя. Признаком того, что переменная – указатель, является символ ‘*’, располагающийся перед именем переменной.

Определим две переменных-указателя:

```
float *p1;
 int *p2;
```

Указатель $p1$ предназначен для хранения адресов участков памяти, размер которого соответствует типу `int`, а переменная $p2$ - для хранения адресов участков памяти, размер которых соответствует типу `double`.

Указатели представляют собой обычные целые числа значения типа `int` и занимают в памяти 4 байта не зависимо от базового типа указателя. Значения указателей при их выводе на экран представляются как целые значения в шестнадцатеричном формате.

4.2. Инициализация указателей

Присвоить указателю адрес некоторой переменной можно инструкцией присваивания и операции &. Пример инициализации указателя представлен в

листинге 4.1. Получить значение объекта, на который ссылается некоторый указатель можно с помощью операции * (разыменовывание указателя).

Листинг 4.1 – Пример использования указателя

```
/* pointer.cpp: Этот файл содержит функцию "main". Здесь начинается и заканчивается
выполнение программы */

#include <iostream>

using namespace std;

int main()
{
	setlocale(LC_ALL, "Russian");
	float pi = 3.14, *p;
	p = &pi;
	cout << "Значение переменной pi: " << pi << endl;
	cout << "Значение переменной pi: " << *p << endl;
	cout << "Адрес переменной pi: " << p << endl;
	return 0;
}
```

Результат выполнения этой программы:

```
Значение переменной pi: 3.14
Значение переменной pi: 3.14
Адрес переменной pi: 012FF77C
```

Из памяти по адресу, который хранится в указатели, берется столько байт памяти, сколько требуется базовому типу указателя. Далее с этими байтами работают, как со значением базового типа указателя.

С помощью указателей можно не только получать значения, расположенные по адресам, хранящимся в указателях, но и записывать нужные значения по этим адресам:

```
*p = 1.618;
```

Указатели могут использоваться в различных выражениях наравне с обычными переменными и константами. При использовании указателей в выражениях важно помнить, что операция * имеет наивысший приоритет по отношению к другим операциям (за исключением операции унарный минус).

Указателю можно присвоить значение другого указателю, если совпадают их базовые типы.

Хотя указатели представляют собой целые значения, присваивать им произвольные целые значения нельзя. Единственным исключением является присвоение указателю нулевого значения. Нулевое значение указателя означает то, что указатель не на что не ссылается.

4.3. Арифметика указателей

К указателям можно применять некоторые арифметические операции, одни из них +, −, ++, − −. Результаты выполнения этих операций по отношению к указателям существенно отличаются от результатов соответствующих арифметических операций, выполняющихся с обычными числовыми данными.

Таблица 4.3 – Представление переменные в памяти

		int A = 20				**int B = 30**				**int p1 = 100**								
…	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	…

В табл. 4.3 представлено, как переменные A и B могут расположиться в памяти. Указатель p1 содержит адрес переменной A. Если выполнить операцию

```
p1 = p1 + 1; (или p1++;)
```

Значение указателя изменится, и станет равно 104. Теперь указатель $p1$ ссылается на переменную B. Таким образом, добавление или вычитание 1 из указателя приводит к изменению его значения на размер базового типа указателя. При этом добавлять или вычитать из указателей можно только целые значения.

Не стоит забывать о приоритете операций:

```
*p1 + 1; и *(p1 + 1);
```

имеют совершенно разный смысл.

4.4. Указатели и массивы

Между массивами и указателями имеется очень тесная связь. Когда в программе определяется некоторый массив. Имя переменной без индексов представляет собой указатель на первый элемент массива:

```
int Arr[5];
```

Если вывести на экран значение переменной `Arr`, то будет отображено некоторое целое значение в шестнадцатеричном формате, которое соответствует адресу первого элемента этого массива.

Именно из-за этого операция присвоения сразу всех значений одного массива в другому запрещена в C++. Попытка присвоения

```
Arr1 = Arr2;
```

привела бы к тому, что переменная `Arr1`, стала бы указывать на ту же область памяти, что и переменная `Arr2`. Адрес, который ранее хранился в переменной `Arr1`, был бы утерян, что привело бы к утечке памяти. Более того, по этой причине запрещены любые изменения значения переменной массива.

Указателю, который имеет такой же базовый тип, как и элемент массива, можно присвоить указатель на массив. Но обратное присвоение выполнить невозможно, так как переменная массива – это константа, изменение которой запрещено.

Так как переменная массива является указателем на первый элемент массива, появляются дополнительные возможности по работе с массивами на основе использования арифметики указателей.

Например, `Arr[4]` эквивалентно `*(Arr + 4)`. Первое выражение – это пример обычной индексации элементов массива. Во втором выражении использовалась арифметика указателей, и с помощи операции + был получен адрес пятого элемента массива. Здесь нельзя забывать о приоритете операций.

Использование арифметики указателей при работе с массивами приводит обычно к уменьшению объема генерируемого кода программы и к уменьшению времени ее выполнения, то есть к увеличению быстродействия.

Поскольку указатель и имя массива взаимосвязаны, указатели можно индексировать, как обычные массивы, а также создавать массивы указателей.

4.5. Объявление многомерных массивов

Многомерные массивы определяются аналогично одномерным массивам. Количество элементов по каждому измерению указывается отдельно в квадратичных

скобках. Элементы многомерных массивов располагаются друг за другом в непрерывном участке памяти.

При инициализации многомерного массива он представляется либо как массив из массивов, при этом каждый массив заключается в свои фигурные скобки (в этом случае левую размерность при описании можно не указывать), либо задается общий список элементов в том порядке, в котором элементы располагаются в памяти:

```
int arr[][] = { {0,0}, {1,1}, {1,0} };
int arr[3][3] = { 0, 0, 0, 1, 1, 1, 2, 2, 2 };
```

Определить, сколько памяти выделено под многомерный массив, можно аналогично одномерному массиву с помощью операции `sizeof`. Необходимо определить, сколько выделяется памяти на 1 элемент массива и умножить на количество элементов массива. В листинге 4.2 представлен пример заполнения многомерного массива.

Листинг 4.2 – Пример инициализации двумерного массива

```
/* arr[NxM].cpp: Этот файл содержит функцию "main". Здесь начинается и заканчивается
выполнение программы */

#include <iostream>
#include <ctime>

using namespace std;

int main()
{
	setlocale(LC_ALL, "Russian");
	const int N = 10, M = 10; // Количество строк матрицы N = 10, столбцов – M = 10
	srand(time(0));
	int arr[N][M]; // Объявляем массив arr размерности NxM
	cout << "Исходный массив данных: ";
	for (int i = 0; i < N; i++)
	{
		for (int j = 0; j < M; j++)
		{
			arr[i][j] = rand() % 9;
			cout << arr[i][j] << " ";
		}
		cout << endl;
	}
	return 0;
}
```

В приведенном выше примере сначала объявляется двумерный массив, затем при помощи двух циклов (один из которых вложенный) происходит заполнение массива случайными числами. Первый цикл нужен, чтобы пройти все строчки

матрицы, а второй цикл – пройти все строчки, заполнить их значениями и вывести на экран.

4.6. Задание к практической работе

Необходимо написать программу, которая:

1) Использую арифметику указателей, заполняет квадратичную целочисленную матрицу порядка N (6,8,10) случайными числами от 1 до N^2 согласно схемам приведенным на рис. 4.1, *а*, и на рис. 4.1, *б*. Пользователь должен видеть процесс заполнения квадратичной матрицы.

Рисунок 4.1 – Схема заполнения квадратичной матрицы: *а* – спиральная схема; *б* – схема заполнения зигзагом

2) Получает новую матрицу, из матрицы п. 1, переставляя ее блоки в соответствии со схемами, которые представлены на рис. 4.2.

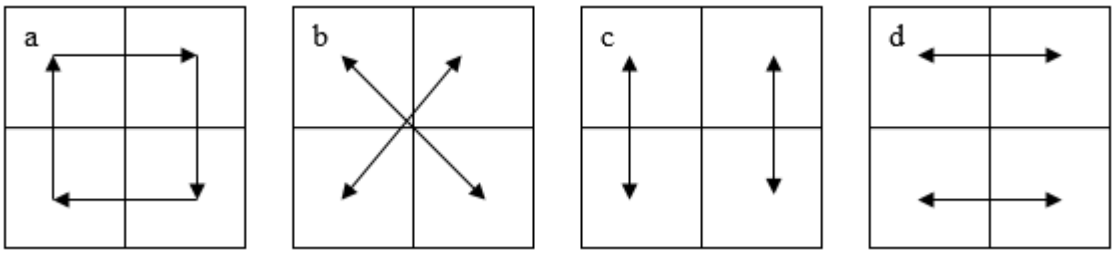


Рисунок 4.2 – Схемы перестановки блоков матрицы

3) Используя арифметику указателей, сортирует элементы любой сортировкой.
4) Уменьшает, увеличивает, умножает или делит все элементы матрицы на введенное пользователем число.

Далее необходимо выполнить индивидуальное задание, которое представлено табл. 4.4 и сделать вывод по проделанной работе.

Таблица 4.4 – Индивидуальные задания

Номер варианта	Задание
1	Заполните матрицу случайными числами (от 1 до 100). Напишите алгоритм, который генерирует ответы к проверочной работе с матрицами. Суть работы заключается в том, что учащийся должен, переставляя строки и столбцы числовой квадратичной матрицы порядка N, добиться того, чтобы наибольший элемент оказался в верхнем левом углу. Вам необходимо вывести решение поставленной задачи с подробным объяснением каждого шага. Пример: 1 3 4 2 Поиск максимального элемента… Максимальный элемент: Arr[2,1] = 4 Меняем строки 2 и 1 местами, получаем 4 2 1 3 Максимальный элемент стоит элемент находится в верхнем левом углу.
2	Заполните матрицу случайными числами от 1 до N^2. Создайте еще одну матрицу порядка N. Заполните её случайными числами от 1 до N^2. Напишите алгоритм, который генерирует ответы к проверочной работе с матрицами. Суть работы заключается в том, что учащийся должен сложить две матрицы (из основного и индивидуального задания). Вам необходимо вывести решение поставленной задачи с подробным объяснением каждого шага.
3	Заполните матрицу случайными числами от 1 до N^2. Создайте еще одну матрицу порядка N. Заполните её случайными числами от 1 до N^2. Напишите алгоритм, который генерирует ответы к проверочной работе с матрицами. Суть работы заключается в том, что учащийся должен вычесть из первой матрицы (из основного задания) вторую матрицу (из индивидуального задания). Вам необходимо вывести решение поставленной задачи с подробным объяснением каждого шага.

Продолжение таблицы 4.4

4	Заполните матрицу случайными числами от 1 до N^2. Создайте еще одну матрицу порядка N. Заполните её случайными числами от 1 до N^2. Напишите алгоритм, который генерирует ответы к проверочной работе с матрицами. Суть работы заключается в том, что учащийся должен умножить две матрицы (из основного и индивидуального задания). Вам необходимо вывести решение поставленной задачи с подробным объяснением каждого шага.
5	Заполните матрицу случайными числами от 1 до N^2. Создайте еще одну матрицу порядка N. Заполните её случайными числами от 1 до N^2. Напишите алгоритм, который генерирует ответы к проверочной работе с матрицами. Суть работы заключается в том, что учащийся должен вычесть из первой матрицы (из основного задания) вторую матрицу (из индивидуального задания). Вам необходимо вывести решение поставленной задачи с подробным объяснением каждого шага.
6	Заполните матрицу случайными числами (от 1 до 100). Напишите алгоритм, который генерирует ответы к проверочной работе с матрицами. Суть работы заключается в том, что учащийся должен определить, является ли заданная матрица магическим квадратом. Вам необходимо вывести решение поставленной задачи с подробным объяснением каждого шага.
7	Заполните матрицу случайными числами (от 1 до 100). Напишите алгоритм, который генерирует ответы к проверочной работе с матрицами. Суть работы заключается в том, что учащийся должен определить, является ли заданная матрица ортонормированной. Вам необходимо вывести решение поставленной задачи с подробным объяснением каждого шага.
8	Заполните матрицу случайными числами (от 1 до 100). Напишите алгоритм, который генерирует ответы к проверочной работе с матрицами. Суть работы заключается в том, что учащийся должен определить и вывести индексы всех седловых

	точек матрицы (элемент является наименьшим в свой строке и одновременно наибольшим в своем столбце или, наоборот). Вам необходимо вывести решение поставленной задачи с подробным объяснением каждого шага.
9	Заполните матрицу случайными числами (от 1 до 100). Напишите алгоритм, который генерирует ответы к проверочной работе с матрицами. Суть работы заключается в том, что учащийся должен транспонировать матрицу. Вам необходимо вывести решение поставленной задачи с подробным объяснением каждого шага.
10	Создайте матрицу 3x3, заполните ее случайными числами (-30 до 30). Напишите алгоритм, который генерирует ответы к проверочной работе с матрицами. Суть работы заключается в том, что учащийся должен найти определитель матрицы. Вам необходимо вывести решение поставленной задачи с подробным объяснением каждого шага.

ПРАКТИЧЕСКАЯ РАБОТА №5
ТЕКСТОВЫЕ СТРОКИ КАК МАССИВЫ СИМВОЛОВ

Цель работы: изучение способов обработки текстовых данных. Получение практических навыков работы с файлами. Изучение различных алгоритмов поиска подстроки в строке.

5.1. Работа с файлами

Для работы с файлами в языке C++ используют потоки трех видов:

- поток ввода (класс `ifstream`);
- поток вывода (класс `ofstream`);
- поток ввода-вывода (класс `fstream`);

Класс ifstream используется для выполнения чтения данных из файлов. Поток `ofstream` – для записи данных в файл. Поток `fstream` – для чтения и записи данных в файл.

Открыть файл можно двумя способами:

1) сначала создать поток, а потом связать его с файлом:

```
ofstream File;
File.open(“D:\\test.txt”);
```

2) создать поток и связать поток с файлом:

```
ofstream File (“D:\\test.txt”);
```

После того, как поток был связан с файлом, необходимо обязательно проверить открылся ли файл. Если файл открыть не удалось, то переменная потока (`File`) принимает значение **false**, если файл открыт – **true**.

Проверку успешного открытия файла можно выполнить с помощью условного оператора:

```
If (!File) // другой вариант if (!File.is_open())
cout << “Ошибка при открытии файла!”;
```

Функция потока `is_open()` возвращает логическое значение в зависимости от результата операции открытия файла.

Файл закрывается с помощью функции потока `close()`:

```
File.close();
```

Каждый поток использует свой вариант функции `open`.

Первый параметр определяет имя открываемого файла (представляет собой массив символов).

Второй параметр определяет режим открытия файла. Этот параметр имеет значение по умолчанию, то есть является необязательным. Возможные значения этого параметра:

- `ios::app` – при открытии файла на запись (поток ofstream) обеспечивает добавление всех выводимых в файл данных в конец файла;
- `ios::ate` – обеспечивает начало поиска данных в файле начиная с конца файла;
- `ios::in` – файл открывается для чтения из него данных;
- `ios::out` – файл открывается для записи данных в файл;
- `ios::binary` – открытие файла в двоичном режиме (по умолчанию все файлы открываются в текстовом режиме);
- `ios::trunc` – содержимое открываемого файла уничтожается (его длина становится равной 0).

Эти флаги можно комбинировать с помощью побитовой операции ИЛИ (|).

Если файл открывается без использования функции open, эти флаги тоже можно использовать:

```
fstream File ("E:\\test.txt", ios :: out | ios :: app);
```

Файл открывается на вывод с добавлением записываемых данных в конец файла.

Запись и чтение данных в текстовых файлах ничем не отличается от способов ввода-вывода данных с помощью потоков `cin` и `cout`. Методы форматирования вывода и вводы данных остаются такими же (флаги форматирования, манипуляторы, функции потоков).

Необходимо помнить, что при использовании операции >> для чтения данных из текстовых файлов, процесс чтения останавливается при достижении пробельного символа. Поэтому для чтения строки текста, содержащей несколько слов, необходимо использовать, например, функцию `getline()`.

5.2. Определение текстовой строки

Текстовые строки представляются с помощью одномерных массивов символов. В языке C++ текстовая строка представляет собой набор символов, обязательно заканчивающийся нулевым символом (‘\0’). Поэтому, если вы хотите создать текстовый массив для хранения 10 (N) символов, нужно выделить память под 11($N + 1$) символов.

Объявленный таким образом массив может использоваться для хранения текстовых строк, содержащих не более 10 символов. Нулевой символ позволяет определить границу между содержащимся в строке текстом и неиспользованной частью строки.

При определении строковых переменных их можно инициализировать конкретными значениями с помощью строковых литералов:

```
char S1[15] = “This is text”;
char S2[] = “Пример текста”;
```

Последние два элемента переменной $S1$ просто не используются, а строка $S2$ автоматически подстраивается под длину инициализирующего текста.

При работе со строками можно обращаться к отдельным символам строки как в обычном одномерном массиве с помощью индексов:

```
cout << S1[0]; // На экране будет выведен символ ‘T’
```

Если строка формируется при помощи цикла (или иного способа), то необходимо ее конец обязательно записать нулевой символ '\0'.

5.3. Ввод текстовых строк с клавиатуры

При выводе строк можно использовать форматирование (манипуляторы или функции потока вывода). Вывод текстовых строк на экран крайне простая задача:

```
char Str[21] = “Это пример текста”;
cout  <<  Str << endl;
cout  <<  “Это текстовый литерал.” << endl;
```

Ввод текста с клавиатуры можно осуществлять разными способами, каждый из которых имеет определенные особенности.

Непосредственное чтение текстовых строк из потока вывода осуществляется до первого знака пробела.

Такой способ чтения обеспечивает ввод символов до первого пробельного символа (не до конца строки). Остальные символы введенного с клавиатуры остаются в потоке ввода и могут быть прочитаны из него следующими операторами $>>$.

Для того чтобы прочесть всю строку полностью, можно воспользоваться одной из функций `gets` или `gets_s` (для этого в программу должен быть включен заголовочный файл `<stdio.h>`).

Функция `gets` имеет один параметр, соответствующий массиву символов, в который осуществляется чтение. Вторая функция (`gets_s`) имеет второй параметр, задающий максимальную длину массива символов Str.

Ввод текста, длина которого (вместе с нулевым символом) превышает значение второго параметра (то есть длины символьного массива Str), приводит к возникновению ошибки при выполнении программы

Предпочтительно использование функции потока ввода `cin.getline`:

```
const int N = 21;
char Str [N];
cin.getline (Str, N);      // Пусть введена строка “Это пример текста”
cout  <<  Str << endl;  // На экран будет выведено “ Это пример текста”
```

Если длина введенного с клавиатуры текста превышает максимальную длину массива Str, в него будет записано (в нашем примере) 20 символов вводимого текста и нулевой символ. Остальные символы введенного текста остаются во входном потоке и могут быть взяты из него следующими инструкциями ввода.

Функция `cin.getline` может иметь третий параметр, задающий символ, при встрече которого чтение строки из потока прекращается:

```
cin.getline (Str, N,  ‘.’);
```

Иногда чтение из потока невозможно (например, попытка считать слишком длинный текст). Для того чтобы продолжить чтение из потока, необходимо восстановить его нормальное состояние. Этого можно достигнуть с помощью функции потока `cin.clear()`, которая сбрасывает состояние потока в нормальное.

Если забирать остатки данных из потока ввода не надо, то следует очистить его с помощью функции `cin.sync()`.

5.4. Класс string

Класс `string` предназначен для работы со строками типа `char`, которые представляют собой строчку с завершающим нулем (символ ‘\0’). Класс `string` был введен как альтернативный вариант для работы со строками типа char.

Чтобы использовать возможности класса string, нужно подключить библиотеку <string> и пространство имен `std`. Объявление же переменной типа string осуществляется схоже с обычной переменной:

```
string S1; // Переменная с именем s1 типа string
string S2 = “Пример”; // объявление с инициализацией
```

Создание нового типа `string` было обусловлено недостатками работы с строками символов, который показывал тип `char`. В сравнении с этим типом `string` имеет ряд основных преимуществ:

- возможность использования для обработки строк стандартные операторы C++ (=, +, <, ==, >, +=, ! =, <=, >=, []), пример их использования представлен в листинге 5.1. Использование типа char приводило требовало написание чрезмерного программного кода;
- обеспечение лучшей надежности программного кода;
- обеспечение строки, как самостоятельного типа данных.

Листинг 5.1 – Пример использования класса `string`

```
/* string.cpp: Этот файл содержит функцию "main". Здесь начинается и заканчивается
выполнение программы */

#include <iostream>
#include <string>

using namespace std;

	int main()
	{
		setlocale(LC_ALL, "Russian");
		string A = "Пример"; // Объявляем и инициализируем строку A
		string B;            // Объявляем строку B
		string C = "текста"; // Объявляем и инициализируем строку C
```

Окончание листинга 5.1

```
            B = A;                // Копируем текст из строки A в строку B
            C = B + " " + C;
            if(C == "Пример текста") // Проверяем корректно ли произошло присоединение
строк
            cout << C;
            return 0;
      }
```

Результат выполнения программы – вывод в консоль сообщения:

```
Пример текста
```

С string можно использовать оператор индексации и получать значения символа. Принцип действия такой же, как и у типа char:

```
string s1 = “hello!”;
cout << s[0]; // Будет выведен символ ‘h’
```

Класс string обладает широким функционалом:

- функция compare() сравнивает одну часть строки с другой;
- функция length() определяет длину строки;
- функции find() и rfind() служат для поиска подстроки в строке (отличаются функции лишь направлением поиска);
- функция erase() служит для удаления символов;
- функция replace() выполняет замену символов;
- функция insert() необходима, чтобы вставить одну строку в заданную позицию другой строки;
- Аналогом использования оператора + является функция append();
- Аналогом использования оператора = является функция assign();

Но весь функционал string накладывает и свой негативный отпечаток. Основным недостатком string в сравнении с типом char является замедленная скорость обработки данных.

5.5. Поиск подстроки в строке

При работе со строками часто будет возникать потребность в поиске набора символа или слов (поиска подстроки в строке). При условии, что текст может быть крайне большим, хочется, чтобы алгоритм поиска подстроки работал быстро.

Самый простой способ подстроки в строке – Линейный поиск – циклическое сравнение всех символов строки с подстрокой. Действительно, этот способ первый приходит в голову, но очевидно, что он будет самым долгим. Убедиться в этом можно по примеру, приведенному на рис. 5.1.

1-й шаг цикла	**H**	E	L	L	O		W	O	R	L	D
	L	O									
2-й шаг цикла	H	**E**	L	L	O		W	O	R	L	D
		L	O								
3-й шаг цикла	H	E	**L**	**L**	O		W	O	R	L	D
			L	**O**							
4-й шаг цикла	H	E	L	**L**	**O**		W	O	R	L	D
				L	**O**						

Рисунок 5.1 – Поиск подстроки “LO” в строке “HELLO WORLD”

На первых двух итерациях цикла сравниваемые буквы не будут совпадать. На третьей же итерации, совпал символ ‘L’, это означает, что теперь нужно сравнивать следующий символ подстроки со следующим символом строки. Видно, что символы отличаются, поэтому алгоритм продолжает свою работу. На четвертой же итерации подстрока была найдена.

Если представить, что исходная строка непорядок больше и подстрока находится в конце строки (или вовсе отсутствует), то сразу видны минусы данного алгоритма.

Одним из самых популярных алгоритмов, который работает быстрее, чем приведенный выше алгоритм, является алгоритм Кнута-Морриса-Пратта (КМП). Идея заключается в том, что не нужно проходить и сравнивать абсолютно все символы строки, если известны символы, которые есть и в строке, и в подстроке.

Суть алгоритма: дана подстрока S и строка T. Требуется определить индекс, начиная с которого образец S содержится в строке T. Если S не содержится в T, необходимо вернуть индекс, который не может быть интерпретирован как позиция в строке. Пример работы алгоритма КМП представлен на рис 5.2.

ТАБЛИЦА ВЛЮЧЕНИЙ

0	1	**2**	**3**	4	5	6	7	8	**9**	10
H	E	L	L	O		W	O	R	L	D

	0	1	**2**	**3**	4	5	6	7	8	**9**	10
1-й шаг цикла	H	E	**L**	**L**	O		W	O	R	L	D
			L	**O**							
2-й шаг цикла	H	E	L	**L**	**O**		W	O	R	L	D
				L	**O**						

Рисунок 5.2 – пример работы алгоритма КМП

Хоть алгоритм и работает быстрее, по-прежнему необходимо сначала пройти всю строку, чтобы определить префиксы или суффиксы (вхождение (индексы) символов).

Алгоритм Бойера-Мура в отличие от КМП полностью не зависим и не требует заранее проходить по строке. Этот алгоритм считается наиболее быстрым среди алгоритмов общего назначения, предназначенных для поиска подстроки в строке.

Преимущество этого алгоритма в том, что ценной некоторого количества предварительных вычислений над подстрокой (но не над исходной строкой, в которой ведётся поиск), подстрока сравнивается с исходным текстом не во всех позициях (пропускаются позиции, которые точно не дадут положительный результат).

Поиск подстроки ускоряется благодаря созданию таблиц сдвигов. Сравнение подстроки со строки начинается с последнего символа подстроки, а затем происходит прыжок, длина которого определяется по таблице сдвигов. Таблица сдвигов строится по подстроке так чтобы перепрыгнуть максимальное количество символов строки и не пропустить вхождение подстроки в строку.

Правила построения таблицы сдвигов:

1) Значение элемента таблицы равно удаленности соответствующего символа от конца шаблона (подстроки).
2) Если символ встречается более одного раза, то применятся значение, соответствующее символу, наиболее близкому к концу шаблона.

3) Если символ в конце шаблона встречается 1 раз, ему соответствует значение, равное длине образа; если более одного раза – значение, соответствующее символу, наиболее близкому к концу образа.
4) Для символов, отсутствующих в образе, применяется значение, равное длине шаблона.

На рис. 5.3. представлен пример поиска подстроки “ELLO” в строке “HELLO WORLD”.

	0	1	**2**	**3**	4	5	6	7	8	**9**	10
1-й шаг цикла	H	E	L	**L**	O		W	O	R	L	D
	E	L	L	**O**							
2-й шаг цикла	H	**E**	**L**	**L**	**O**		W	O	R	L	D
		E	**L**	**L**	**O**						

Таблица отступов

3	1	1	4
E	L	L	O

Длина подстроки: 4

Рисунок 5.3 – Пример работы алгоритма Бойра-Мура

Сначала была построена таблица отступов и подсчитана длина подстроки. Затем начинается алгоритм поиска подстроки в строке. Сравнивает символ ‘L’ строки и ‘O’ подстроки. Элементы не совпадают, поэтому необходимо определить длину отступа. Символ ‘L’ присутствует в таблице отступа, длина отступа равняется 1. Подстрока смещается на 1 символ вперед. На следующей итерации подстрока найдена.

5.6. Задание к практической работе

Необходимо написать программу, которая реализует поставленную задачу:

1) С клавиатуры или с файла (пользователь сам может выбрать способ ввода) вводится последовательность, содержащая от 1 до 50 слов, в каждом из которых от 1 до 10 строчных латинских букв и цифр. Между соседними словами произвольное количество пробелов. За последним символом стоит точка.
2) Необходимо отредактировать входной текст:
    - удалить лишние пробелы;

- удалить лишние знаки препинания (под «лишними» подразумевается несколько подряд идущих знаков (обратите внимание, что «…» - корректное использование знака) в тексте);
- исправить регистр букв, если это требуется (пример некорректного использования букв: пРиМЕр);

3) Необходимо найти подстроку, которую введёт пользователь в имеющейся строке. Реализуйте два алгоритма: первый алгоритма – Линейный поиск, а второй алгоритм согласно вашему номеру в списке. Четные номера должны реализовать алгоритм КНМ, а нечетные – Бойера-Мура.
4) Напишите алгоритм генерации вариантов контрольной работы по поиску подстроки в строке и ответы к ним (Используйте алгоритмы п. 3.). В текстовый файл “task” выведите задание, в текстовый файл “response” выведите ответы на сгенерированные варианты. Важно, что учащийся, выполняющий работу, должен подробно объяснить ход действий, соответственно ответы должны быть полностью расписаны.

   Описание действия можно выполнить в виде таблиц (двумерной матрицы) с минимальными текстовыми пояснениями.

Необходимо выполнить индивидуальное задание, которое представлено в табл. 5.1 и сделать вывод по проделанной работе.

Таблица 5.1 – Индивидуальные задания

Вариант	Правило обработки последовательности
1	Вывести на экран слова последовательности в обратном порядке.
2	Вывести на экран слова последовательности в алфавитном порядке.
3	Вывести на экран слова последовательности, не содержащие цифр.
4	Вывести на экран только те слова последовательности, в которых встречаются одинаковые буквы.
5	Вывести на экран только те слова последовательности, в которых первая буква слова встречается в этом слове еще раз.
6	После окончания ввода последовательности вывести на экран сначала все слова, содержащие только буквы, затем слова, содержащие только цифры, а потом слова, содержащие и буквы, и цифры.

Окончание таблицы 5.1

7	Вывести на экран ту же последовательность, заменив во всех словах первую букву соответствующей прописной буквой.
8	Вывести на экран количество символов в каждом слове исходной последовательности.
9	Вывести на экран ту же последовательность, удалив из всех слов заданный набор букв и (или) цифр.
10	Вывести на экран ту же последовательность, заменив во всех словах цифры на буквы латинского алфавита, номера которых в алфавите равны заменяемой цифре.
11	Вывести на экран ту же последовательность, переместив все цифры, содержащиеся в словах, в конец соответствующих слов.
12	Вывести все слова исходной последовательности на экран вертикально.
13	Вывести на экран все слова последовательности в две или три колонки (в зависимости от количества слов) с выравниванием слов по правой границе колонки.

ПРАКТИЧЕСКАЯ РАБОТА №6
РЕКУРСИВНОЕ ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ

Цель работы: изучить рекуррентные соотношения. Научиться создавать и применять рекурсивные функции для достижения поставленных целей. Введение в модульное программирование.

6.1. Объявление и определение функции

Функция – это именованная последовательность описаний и операторов, выполняющая какое-либо законченное действие. Функция может принимать параметры и возвращать значение.

Использование функций позволяет:

1) значительно упростить разработку сложных программ;
2) сократить объем текста программы и генерируемого результирующего кода программы;
3) значительно упростить откладку и модификацию программ;
4) распределить работу над одной программой между различными исполнителями программистами.

Фактически разработка более-менее сложных программ практически невозможна без использования функций. Более того любая программа на C++ состоит из функций, одна из которых должна иметь имя main (определят точку начала выполнения программы). Функция начинает выполняться в момент вызова.

Любая функция должна быть объявлена и определена. Как и для других величин, объявлений может быть несколько, а определение только одно. Объявление функции должно находиться в тексте раньше ее вызова для того, чтобы компилятор мог осуществить проверку правильности вызова.

Любая функция состоит из двух основных элементов: заголовка и тела функции. Заголовок функции имеет следующий формат:

```
<Тип возвращаемого значения> <Идентификатор – имя функции> (<Параметры>)
```

Тело функции представляет собой блок инструкций языка программирования, разделенных символами “точка с запятой”:

```
{
    <Инструкция 1>;
    <Инструкция 2>;
    …
    <Инструкция N>;
}
```

Внутри тела функции могут использоваться любые инструкции языка программирования. Количество инструкций не ограничено.

Функция может возвращать одно значение, сформированное внутри функции, через свое имя.

```
int Sum(int a, int b) { return a+b; }
```

Тип данных возвращаемого значения определяется элементом заголовка <Тип возвращаемого значения>. Тип возвращаемого значения может быть любым, за исключением типа массива (но указателем он может быть). Пример использования функции представлен в листинге 6.1.

Листинг 6.1 – Аппроксимация функции по методу наименьших квадратов

```
/* Аппроксимация функции по методу наименьших квадратов.cpp : Этот файл содержит функцию
"main". Здесь начинается и заканчивается выполнение программы */

#include <stdio.h>
#include <stdlib.h>
#include <iostream>

using namespace std;

const int N = 20;

// Вычисление коэффициентов аппроксимирующей прямой
void getApprox(double x[2][N], double *a, double *b, int n) {
	double sumx = 0;
	double sumy = 0;
	double sumx2 = 0;
	double sumxy = 0;
	for (int i = 0; i < n; i++) {
		sumx += x[0][i];
		sumy += x[1][i];
		sumx2 += x[0][i] * x[0][i];
		sumxy += x[0][i] * x[1][i];
	}
	*a = (n*sumxy - (sumx*sumy)) / (n*sumx2 - sumx * sumx);
	*b = (sumy - *a*sumx) / n;
}
```

```
int main() {
	setlocale(0, "");
	double x[2][N], a, b;
	int n;
	system("cls");
	for (int i = 0; i < 2; ++i)
	{
		for (int j = 0; j < N; ++j)
		{
			cout << "x[" << i << "][" << j << "]"<<" = ";
			cin >> x[i][j];
		}
	}
	for (int i = 0; i < N; i++)
		cout << x[0][i] << " - " << x[1][i] << endl;
	getApprox(x, &a, &b, N);
	cout << "Коэфициент a :" << a << endl << "Коэфициент b :" << b << endl;
	return 0;
}
```

Не все функции должны возвращать значения. В этом случае `<Тип возвращаемого значения>` задается ключевым словом `void`, которое означает – “пусто” – возвращаемое значение отсутствует.

6.2. Обращение к функциям в программе

Функция может возвращать значение или не возвращать его. От этого зависит способ вызова функции:

```
void Foo1 (int a, int b); // Прототип некоторой функции void
int Foo2 (int a, int b); // Прототип некоторой функции int
…
Foo1(a,b); // Вызов функции void
c = Foo(a,b); // Вызов функции int
…
```

Однако если даже функция возвращает значение, но необходимости в этом значении есть, ее можно вызвать, как и `void` функцию.

При вызове функции на места параметров подставляются некоторые конкретные значения, которые обычно называют аргументами функции.

Параметры функции служат для обеспечения взаимодействия между функцией и вызвавшей ее программой. Другими словами, параметры служат для обмена данными между программой и функцией.

Количество, типы данных и порядок следования аргументов должны соответствовать списку параметров функции. Функции, возвращающие значения, могут использоваться в качестве элементов различных выражений.

Не у всех процедур должны быть параметры (примером этого является функция `main`). Если у функции нет параметров, то соответствующий элемент заголовка пропускается или обозначатся словом `void`.

Существует несколько способов передачи параметров в функцию:

1) по значению;
2) по адресу.

Механизм передачи данных через параметры функции очень прост. При вызове функции в определенной области памяти (в стеке программы) для каждого параметра функции создается переменная соответствующая типу данных параметра. В эти переменные копируются значения аргументов, использовавшихся при вызове функции.

При выполнении кода функции эти копии значений аргументов могут использоваться для обработки, могут изменять свои значения, но эти изменения никак не затрагивают значений самих аргументов. Поэтому после завершения работы функции, значения аргументов, которые были использованы при вызове функции, останутся такими же, какими они были до вызова функции.

Такой способ передачи данных обычно носит название передача данных по значению. Можно считать, что такой способ служит для передачи данных только внутрь функции, но не из функции обратно в программу.

6.3. Передача данных с помощью указателей

При передаче данных только по значению функции могут возвращать только одно значение – через свое имя. Если это был бы единственный способ передачи данных в функцию, то это было бы очень серьезным ограничением полезности функции. Очень часто требуется, чтобы функции возвращали более одного значения, сформированного внутри функции.

Решение весьма тривиально: необходимо заставить функцию обрабатывать не копию аргумента, а само значение аргумента. Для этого можно использовать передачу данных с помощью указателей.

В принципе здесь также используется передача данных по значению. Но в качестве значения используется адрес аргумента, а не само значение аргумента. И далее в функции осуществляется работа со значением аргумента путем обращения к нему через его адрес:

Таким образом, для использования передачи данных с помощью указателей необходимо обязательно выполнить три следующих пункта:

1. Соответствующий параметр в заголовке функции необходимо определить как указатель на тип данных аргумента.
2. При вызове функции на месте параметров-указателей необходимо использовать адрес аргумента, а не сам аргумент.
3. При обращении внутри функции к значению аргумента через параметр-указатель необходимо осуществить разыменование этого указателя.

Несколько проще обстоит дело с передачей массивов, так как переменные типа массив сами являются указателями на первый элемент массива. В связи с этим отпадает необходимость в выполнении пунктов 2 и 3 из перечисленных выше.

Этот способ передачи данных часто называют передачей данных по адресу.

6.4. Передача данных по ссылке

Передача данных с использованием указателей несколько трудоемка, поскольку требует при вызове функций передавать адрес аргумента, а внутри функции разыменовывать параметры-указатели. Кроме того, неправильное выполнение этих дополнительных действий может привести к появлению ошибок.

В языке C++ имеется более простой способ передачи данных по адресу, а именно – передача данных по ссылке.

Переделаем пример предыдущего параграфа так, чтобы данные в нем передавались не через указатель, а по ссылке:

```
int  Div (int N1, int N2, int &Ost)    //  int &Ost – параметр-ссылка
{
    Ost = N1 % N2;    // Разыменования параметра-ссылки Ost не требуется
    return N1 / N2;
}

int main()
{
    int I = 10, J = 3, R, O;
    R = Div (I, J, O);   // Используется сам аргумент O, а не его адрес
    return 0;
}
```

Из этого примера видно, что для определения параметра, с помощью которого функция может вернуть остаток от деления, используется символ &, а не ∗. Именно так определяются параметры-ссылки. Внутри процедуры, для получения доступа к значению данных, разыменовывать параметр-ссылку не нужно. При вызове процедуры используется сам аргумент, а не его адрес. Таким образом, использование передачи данных по ссылке значительно проще, чем передача данных через указатели.

На самом деле в этом способе передача данных осуществляется точно так же, как и при использовании указателей, только операции взятия адреса аргумента и разыменования параметра-ссылки осуществляется компилятором автоматически, скрытно от нас.

Способ передачи данных по ссылке также можно отнести к передаче данных по адресу, и он полностью эквивалентен по своему эффекту передачи данных через указатели.

В целом, передача данных по адресу (с помощью указателей или по ссылке), является более эффективной, чем передача данных по значению. Это объясняется тем, что при передаче данных по значению осуществляется создание копии аргумента функции, а на это тратится и память, и время. При передаче по адресу затраты памяти и времени существенно меньше, так как в функцию передается только адрес данных, а не сами данные, объем которых часто существенно превышает размер адреса.

Возникает вопрос: почему всегда не использовать передачу данных по адресу, поскольку это более эффективно? Недостатком передачи данных по адресу является

скрытый побочный эффект, связанный с возможным непредвиденным изменением внутри функции значения аргумента, переданного по адресу.

6.5. Рекурсивное использование функций

Рекурсивной называется функция, которая вызывает саму себя. Такая рекурсия называется прямой. Существует еще и косвенная рекурсия, когда две или более функций взвывают друг друга. Если функция вызывает себя, в стеке создается копия значений ее параметров, как и при вызове обычной функции, после чего управление передается первому исполняемому оператору функции. При повторном вызове этот процесс повторяется.

Для завершения вычислений каждая рекурсивная функция должна содержать хотя бы одну не рекурсивную ветвь алгоритма, заканчивающуюся оператором возврата. При завершении функции соответствующая часть стека освобождается, и управление передается вызывающей функции, выполнение которой продолжается с точки, следующей за рекурсивным вызовом.

Классическим примером рекурсивной функции (листинг 6.2) является вычисление факториала (существуют и иные способы вычисления факториала). Для того чтобы получить значения факториала числа n, требуется умножить на n факториал числа $(n-1)$. Факториал нуля при этом равняется единице.

Листинг 6.2 – Вычисление факториала при помощи рекурсии

```
/* factorial.cpp: Этот файл содержит функцию "main". Здесь начинается и заканчивается выполнение программы */

#include <iostream.h> // Подключение cin/cout
#include <conio.h> // Подключение getch

using namespace std;

int factorial(int i) // Функция вычисляющая факториал числа i
{
  if (i==0) return 1;
  else return i*factorial(i-1);
}

int main()
{
  int number; // Число, факториал которого необходимо вычислить
```

Окончание листинга 6.5

```
    cout << “Введите число: ”;
    cin >> number; // Ввод числа number
    cout << “Факториал числа number = ” << factorial(number) << endl;
    getch(); // Вызывание прерывания. Ожидание ввода с клавиатуры
    return 0;
}
```

Рекурсивные функции чаще всего применяют для компактной реализации рекурсивных алгоритмов, а также для работы со структурами данных, описанными рекурсивно.

Любую рекурсивную функцию можно реализовать без применения рекурсии, для этого программист должен обеспечить хранение всех необходимых данных самостоятельно.

Достоинством рекурсии является компактная запись, а недостатками – расход времени и памяти на повторные вызовы функции и передачу ей копий параметров, и, самое главное – опасность переполнения стека.

6.6. Задание к практической работе

Необходимо написать программу, которая реализует поставленную задачу:

1) Кузнечик скачет по вершинам восьмиугольника $ABCDEFGH$ (рис. 6.1), каждый раз перемещаясь в одну из соседних вершин. За 1 прыжок он может переместиться в одну из соседних вершин.
2) Определить (используя рекурсивный вызов функций), сколькими способами он может попасть из A в C за n прыжков. Количество прыжков определяет пользователь. Помимо вывода количества удовлетворительных и неудовлетворительных исходов необходимо вывести весь путь кузнечика (например, $A \to B \to C -$ successful).
3) Сколькими способами кузнечик может попасть из A в C за n прыжков, если через вершину E он не может прыгать. Количество прыжков определяет пользователь. Помимо вывода количества удовлетворительных и неудовлетворительных исходов необходимо вывести весь путь кузнечика (например, $A \to B \to C -$ successful).

4) Подсчитайте вероятность (количество положительных исходов / общее количество исходов) выживания кузнечика за n секунд, если в вершине E находится мина (в случае попадания в вершину E кузнечик считается мертвым, а рекурсивный вызов функции прекращается). Кузнечик скачет каждую секунду. Помимо вывода вероятности выживания необходимо вывести весь путь кузнечика (например, $A \to B \to C \to D \to E - \text{RIP}$).
5) Модернизируйте алгоритм таким образом, чтобы пользователь сам мог выбирать размерность фигуры (k-угольник), стартовую и конечную вершину, ограничивающую вершину и вершину, в которой лежит бомба.

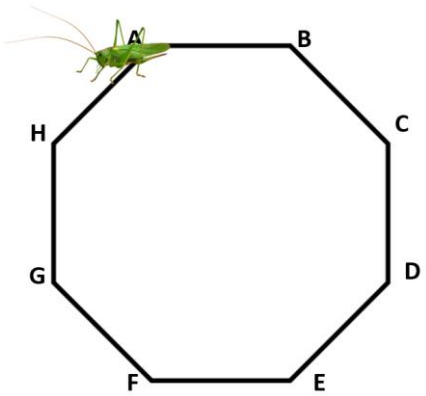


Рисунок 6.1 – Графическое представление задачи

Сделайте вывод по проделанной работе.

ПРАКТИЧЕСКАЯ РАБОТА №7
ТИПЫ ДАННЫХ, ОПРЕДЕЛЯЕМЫЕ ПОЛЬЗОВАТЕЛЕМ. СТРУКТУРЫ

Цель работы: изучение и организация структур. Получение практических навыков работы со структурами. Определение преимуществ и недостатков использования структур.

7.1. Переименование типов (typedef)

Чтобы программы была более ясной, можно задать типу новое имя с помощью ключевого имя `typedef`:

```
typedef тип имя [размерность];
```

Квадратные скобки являются элементом синтаксиса. Размерность может отсутствовать. Кроме задания типам с длинными описаниями более коротких псевдонимов, `typedef` используют для облегчения переносимости программ: если машинно-зависимые типы объявить с помощью оператора `typedef`, при переносе программы потребуется внести изменения только в эти операторы.

7.2. Понятие структуры (struct)

Структуры представляют собой группы связанных между собой, как правило, разнотипных переменных, объединенных в единый объект, в отличие от массива, все элементы которого однотипны. В языке C++ структура является видом класса и обладает всеми его свойствами. Чаще всего ограничиваются тем, как структуры представлены в языке C:

```
struct [имя_типа] {
 тип_1 элемент_1;
 тип _2 элемент_2;
 …
 тип_k элемент_k;
} [ список_описателей ];
```

Описание структуры начинается ключевым словом `struct`. Каждая входящая в структуру переменная называется членом (полем, элементом) структуры и

описывается типом данных и именем. Поля структуры могут быть любого типа данных. Их количество не лимитировано.

Вся эта конструкция является инструкцией языка программирования, поэтому после нее всегда должен ставиться символ ‘;’.

При описании структуры память для размещения данных не выделяется. Работать с описанной структурой можно только после того, как будет определена переменная (переменные) этого типа данных, только при этом компилятор выделит необходимую память.

Для инициализации структуры значения ее элементов перечисляют в фигурных скобках в порядке их описания:

```
struct complex{
     float real, im;
} data [2][2] = {
 {{1,1}, {2,2}},
 {{3,3}, {4,4}}
};
```

Все поля структурных переменных располагаются в непрерывной области памяти одно за другим. Общий объем памяти, занимаемый структурой, равен сумме размеров всех полей структуры. Для определения размера структуры следует использовать инструкцию `sizeof()`.

7.3. Доступ к полям структур

Чтобы записать данные в структурную переменную, необходимо каждому полю структуры присвоить определенное значение. Для этого необходимо использовать оператор “.” (“точка”):

```
struct Stack { // Стек
   float arr[100];
   short topIndex;
};
…
   Stack stack; // Объявляем переменную типа Stack
   Stack.arr[0] = 1;
…
```

При доступе к определенному полю его следует рассматривать как обычную переменную, тип данных которой соответствует типу этого поля. Поля структур могут участвовать в качестве операндов любых выражений, допускающих использование операндов соответствующего типа данных.

Копирование данных из одной структурной переменной в другую осуществляется простой операцией присваивания не зависимо от количества полей и размера структуры (это можно делать только в том случае, когда обе переменные одного и того же типа).

В программировании очень часто используются такие конструкции, как массивы структур. Например, сведения о студентах некоторой учебной группы можно хранить в массиве студентов:

```
t_Student Gruppa_N [30];
```

Был определен 30-ти элементный массив, каждый элемент которого предназначен для хранения данных одного студента. Получение доступа к данным некоторого студента из группы *N* осуществляется обычной индексацией переменной массива. Поскольку поля структуры могут быть любого типа данных, то они могут в свою очередь быть другой структурой или массивом других структур:

```
struct Stud {
   char FN[100];
   short listNumber;
};

struct Group {
   int groupNumber;
   short students;
   Stud stud[30];
};
```

Но в структуре поля нельзя использовать элемент, тип которого совпадает с типом самой структуры, так как рекурсивное использование структур запрещено.

7.4. Указатели на структуры

Любая структурная переменная занимает в памяти определенное положение, характеризующееся конкретным адресом. Для работы с адресами структурных

переменных (как и для простых переменных) можно использовать указатели. Указатели на структурные переменные определяются точно так же, как и для обычных переменных. Разыменование указателя (обращение к данным по адресу, хранящемуся в указателе) осуществляется так же обычным образом.

Через указатели можно работать с отдельными полями структур. Для доступа к полю структуры через указатель используется оператор “→” (“стрелка”), а не “точка”.

7.5. Структурные параметры функции

Структуры можно использовать в качестве параметров функций, как и обычные переменные. Для структур поддерживаются все три механизма передачи данных – по значению, через указатели и по ссылке.

Передачу структур в функции по значению необходимо использовать аккуратно:

```
void WriteStudent ( t_Student S )
{
    cout << "Фамилия: " << S.Fam << endl;
    cout << "Имя: " << S.Name << endl;
    cout << "Год рождения: " << S.Year << endl;
    if ( S.Sex )
         cout << "Пол: " << "М\n";
    else
         cout << "Пол: " << "Ж\n";
    cout << "Средний балл: " << S.Grade << endl;
}
```

Вызов такой функции сопровождается дополнительным расходом памяти для создания локальной переменной *S* и дополнительными затратами времени на физическое копирование данных из аргумента в параметр *S*. Учитывая то, что объем структур может быть очень большим, то эти дополнительные затраты вычислительных ресурсов могут быть чрезмерными.

Предпочтительно использование передачи структуры по указателю или ссылке:

```
void WriteStudent ( t_Student *S )
{
    cout << "Фамилия: " << S -> Fam << endl;
    cout << "Имя: " << S -> Name << endl;
    cout << "Год рождения: " << S -> Year << endl;
```

```
    if ( S -> Sex )
         cout << "Пол: " << "М\n";
    else
         cout << "Пол: " << "Ж\n";
    cout << "Средний балл: " << S -> Grade << endl;
}
```

Фактической передачи данных в функцию не осуществляется. Дополнительные затраты памяти для создания локальной переменной небольшие – это адрес памяти (4 байта, независимо от размера самой структуры). Вызов такой функции будет осуществляться быстрее, а расход памяти существенно меньше, чем при передачи данных по значению.

Передача по ссылке по своей эффективности эквивалентна передачи данных через указатель. Однако, поскольку при передаче данных по ссылке все адресные преобразования берет на себя компилятор, существенно упрощается программирование действий со структурами. При использовании ссылочных параметров структурных типов доступ к членам структуры осуществляется обычным способом – с помощью оператора “точка”.

Недостатком этих способов является то, что случайные изменения значений полей структуры внутри функции отразятся на значении аргумента после окончания работы функции. Если необходимо предотвратить изменения переданных по адресу аргументов, можно при определении соответствующего параметра объявить его константой (использовать спецификатор const).

7.6. Задание к практической работе

Необходимо создать массив структур, содержащий информацию о студентах: ФИО, пол, номер группы, номер в списке группы, оценки за прошедшую сессию (всего 3 экзамена и 5 дифф. зачётов), форма обучения, отметка времени о внесении или изменении данных. Если студент имеет оценку 2, то запись о нем нужно удалить. Написать функции, реализующие операции со структурами:

1) Создание новой записи о студенте.
2) Внесение изменений в уже имеющуюся запись.
3) Вывод всех данных о студентах.

4) Вывод информации о всех студентов группы N. N – инициализируется пользователем.
5) Вывод топа самых умных студентов с наивысшим по рейтингу среднем баллом за прошедшую сессию.
6) Вывод количества студентов мужского и женского пола.
7) Определение количества студентов, которые будут получать стипендию (стипендия начисляется если у студента нет 3 и форма обучения - очная);
8) Вывод данных о студентах, которые не получают стипендию; учатся только на «хор»; учатся на «хор» и «отл»; учатся только на «отл».
9) Вывод всех записей, сделанных в день, который введёт пользователь. Вывод всех записей, сделанных после полудня. Вывод всех записей, сделанных до полудня.
10) Вывод данных о студентах, имеющий номер в списке – k.
11) Вывод данных о всех студентах в файл students.

Далее необходимо выполнить индивидуальное задание (табл. 7.1), информация должна вноситься в достаточном количестве, и сделать вывод по проделанной работе.

Таблица 7.1 – Задания по вариантам

Вариант	Задание
1	Необходимо составить очередь на предоставление места в общежитии. Для этого необходимо добавить в структуру дополнительные данные: иногородний ли студент или нет, доход семьи. Общежитие предоставляется, если студент иногородний, в первую очередь тем, у кого наименьший доход в семье. Выведите информацию о студентах, у которых доход меньше N.
2	В университете проходит «Кубок первокурсника» – спортивные соревнования по волейболу, баскетболу, футболу, скалолазанию и плаванию. Создайте новую структуру, которая будет содержать информацию о названии команды, ФИО капитана команды, наличие золотого значка ГТО, годе рождения. Выведите информацию о студентах, которые: моложе 18 лет; имеют золотой значок ГТО, являются капитанами команды.

Продолжение таблицы 7.1

3	На территории университета действует библиотека. Каждому студенту нужно присвоить номер его читательского билета. Создайте новую структуру, где будет храниться данные о книгах: автор, название, год издания, количество страниц, номер студенческого билета студента, который взял книгу. Если книга находится в библиотеке по умолчанию номер билета приравнивается к 0. Студент может взять несколько книг. Необходимо обеспечить возможность возврата книги и ее приобретения.
4	Создайте структуру, в которой будет храниться информация о преподавателях, которые будут вести в текущем семестре у студентов групп (можно ограничиться 8 преподавателями). Информация: ФИО, пол, возраст, ученная степень, наименование дисциплины, почта, номер группы, у которой ведет. Добавьте возможность вывода информации о преподавателе по Фамилии, номеру группы, наименованию дисциплины.
5	Создайте структуру, в которой будет храниться информация о студентах волейбольной спортивной секции: ФИО, пол, возраст, размер одежды, наличие разряда, номер курса. Внесите информацию о студентах в достаточном количестве. Выведите информацию: о всех студентах, имеющих разряд по волейболу; о женской сборной волейбольной спортивной секции (подразумевается, что все студенты входят в сборную мужскую или женскую в зависимости от пола); о всех студентах k курса (всего 6 курсов); о студентах мужского пола, чей размер одежды больше, чем S.
6	На территории университета работает столовая. Создайте структуру, в которой будет храниться информация о блюдах столовой: наименование, вес, стоимость, оценка блюда от студентов (по 5-ти бальной шкале). Внесите информацию о студентах в достаточном количестве. После каждой покупки студент оставляет оценку блюда. Итоговая оценка блюда равняется среднему всех оценок этого блюда. Выведите информацию: о самых высоко оцениваемых блюдах столовой; о всех блюдах, стоимость которых меньше C (C – инициализируется пользователем); Подсчитайте, сколько должен заплатить пользовать за блюда, которые он выбрал.

Продолжение таблицы 7.1

7	На территории университета скоро пройдет научная конференция. Создайте структуру, в которой будет храниться информация о гостях, которые приглашены на неё: ФИО, статус (выступающий/слушатель), ученная степень, наименование университета, из которого прибыли, тема выступление, если статус – выступающий. Внесите информацию о студентах в достаточном количестве. Выведите информацию: о всех выступающих, о всех иногородних приглашенных, о всех студентах, которые приглашены выступать, о всех студентах из университета, в котором проходит конференция (по умолчанию это университет «ЛЭТИ»)
8	Необходимо составить очередь на предоставление материальной помощи студенту. Для этого необходимо добавить в структуру дополнительные данные: иногородний ли студент или нет, доход семьи, получает ли социальную стипендию. Материальная помощь предоставляется, если студент иногородний, в первую очередь тем, у кого наименьший доход в семье. Если студент получает социальную стипендию, то он обязательно должен получить материальную помощь. Если количество студентов, получающих материальную помощь, больше выделенного количества материальных помощей, то составляется рейтинг по среднему доходу семьи, и материальную поддержку получат студенты с наименьшим доходом в семье. Внесите информацию о студентах в достаточном количестве. Всего университет может выделить 10 материальных помощей. Выведите информацию: о студентах, у которых доход меньше N; которые обязательно получат материальную помощь.
9	Создайте структуру, в которой будет храниться информация о студентах баскетбольной спортивной секции: ФИО, пол, возраст, размер одежды, наличие разряда, номер курса. Внесите информацию о студентах в достаточном количестве. Выведите информацию: - о всех студентах, имеющих разряд по баскетболу; – о мужской сборной баскетбольной спортивной секции (подразумевается, что все студенты входят в сборную мужскую или женскую в зависимости от пола); – о всех студентах 1 курса; – о студентах мужского пола, чей размер одежды меньше, чем L.

Окончание таблицы 7.1

10	Создайте структуру, в которой будет храниться информация о студентах спортивной секции легкой атлетики: ФИО, пол, возраст, размер одежды, наличие разряда, номер курса. Внесите информацию о студентах в достаточном количестве. Выведите информацию: – о всех студентах, не имеющих разряда; – о женской сборной волейбольной спортивной секции (подразумевается, что все студенты входят в сборную мужскую или женскую в зависимости от пола); – о всех студентах 2 курса; – о студентах женского пола, чей размер одежды S.
11	Создайте структуру, в которой будет храниться информация об абитуриентах бакалавриата: ФИО, пол, возраст, город, средний балл ЕГЭ, наличие оригинала аттестата, средний балл аттестата. Внесите информацию об абитуриентах в достаточном количестве. Выведите информацию: – об абитуриентах младше 18 лет; – об абитуриентах, чей средний балл ЕГЭ больше 85; – об абитуриентах, чей средний балл ЕГЭ меньше 80; – об иногородних абитуриентах (университет, в который хотят поступить абитуриенты находится в Санкт-Петербурге); – об абитуриентах, которые подали оригинал аттестация; – об абитуриентах с аттестатом с отличием;
12	Создайте структуру, в которой будет храниться информация об абитуриентах магистратуры: ФИО, пол, возраст, город, средний диплома, заканчивал ли бакалавриат в этом университете. Внесите информацию об абитуриентах в достаточном количестве. Выведите информацию: - об абитуриентах с красным дипломом (средний балл 4,85 и выше); - об иногородних абитуриентах (университет, в который хотят поступить абитуриенты находится в Санкт-Петербурге); - вывести всех абитуриентов мужского пола; Составьте топ абитуриентов с самым высоким средним баллом диплома.

ПРАКТИЧЕСКАЯ РАБОТА №8
ЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ. ДИНАМИЧЕСКИЕ МАССИВЫ И ДВУСВЯЗНЫЕ СПИСКИ

Цель работы: Изучение свойств и организация динамических массивов и двусвязных списков. Получение практических навыков в работе с динамическими массивами и двусвязными списками. Проведение сравнительной характеристики скорости вставки, получения и удаления элементов из них.

8.1. Распределение памяти при работе программы

Схема распределения памяти под программу показана на рис. 8.1.

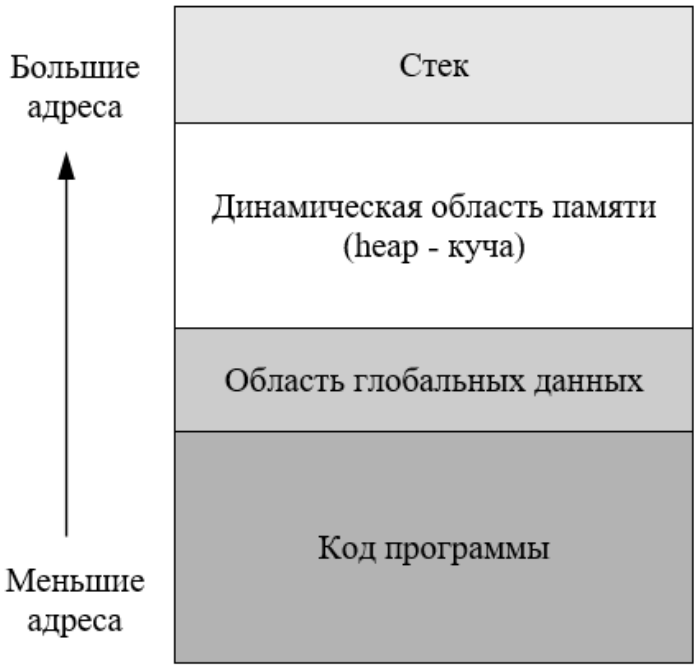


Рисунок 8.1 – Схема распределения памяти под программу

Область кода программы предназначена для хранения инструкций функций программы, обеспечивающих обработку данных. Данные в программе представляются переменными и константами. Для хранения глобальных данных предназначена область глобальных данных. Стек программы используется при вызове функций для передачи параметров и хранения локальных данных.

Распределение памяти для хранения всех обычных переменных осуществляется компилятором, и адреса, и объемы соответствующих участков памяти (в области

глобальных данных) жестко закреплены за этими переменными на все время работы программы и изменено быть не может.

Однако во многих задачах невозможно заранее предсказать, сколько места (количество переменных, объемы массивов и т.д.) потребуется для решения задачи – это так называемые задачи с неопределенной размерностью. Решить эту проблему можно лишь в том случае, если иметь механизм, позволяющий создавать новые объекты по мере возникновения необходимости в этих объектах или изменять объемы памяти, выделенные под эти объекты (например, объемы массивов).

Между областью глобальных данных и стеком располагается так называемая динамическая область памяти, которую как раз и можно использовать в процессе работы программы для реализации механизма динамического управления памятью.

8.2. Одномерные динамические массивы

Для того чтобы создать в динамической области некоторый объект необходима одна обычная (не динамическая переменная) переменная-указатель. Сколько таких объектов нам понадобится для одновременной обработки – столько необходимо иметь обычных переменных-указателей. Таким образом, проблема “задач неопределенной размерности” созданием одиночных динамических объектов решена быть не может.

Решить эту проблему поможет возможность создавать в динамической области памяти массивы объектов с таким количеством элементов, которое необходимо в данный момент работы программы – то есть создание динамических массивов. Действительно, для представления массива требуется всего одна переменная-указатель, а в самом массиве, на который ссылается этот указатель, может быть столько элементов, сколько требуется в данный момент времени.

Для создания одномерного динамического массива, элементами которого являются, например, действительные числа, используется следующий синтаксис инструкции `new (стиль C++)`:

```
double *Arr = new double [100];
```

Освободить динамическую область от этого массива можно с помощью инструкции `delete`.

```
delete [] Arr;
```

После чего занятый участок памяти будет возвращен в список свободной памяти и может быть повторно использован для размещения других динамических объектов.

Язык C++ поддерживает и “старый”, заимствованный от языка C, стиль работы с динамической областью. Достаточно часто бывает полезно использовать именно этот механизм управления динамической памятью.

В языке C отсутствуют инструкции `new` и `delete`. Вместо них для управления динамической памятью используются библиотечные функции:

```
void *malloc (size );
void *calloc(num, size );
void free( void *memblock );
void *realloc( void *memblock,  size );
```

Функция `malloc` выделяет в динамической области `size` байт памяти и возвращает адрес этого участка в виде указателя (`void` *). Поскольку возвращаемый указатель не привязан ни к какому типу данных, при работе с ним потребуется явное приведение типов данных (см. пример ниже).

Функция `calloc` выделяет в динамической области `size * num` байт памяти и возвращает адрес этого участка в виде указателя (`void` *).

Функция `free` освобождает участок динамической памяти по адресу `memblock` и возвращает его в список свободной памяти для повторного использования.

Функция `realloc` позволяет изменить размер (уменьшить или увеличить) ранее выделенной по адресу `memblock` памяти, установив новый размер выделенного участка равным `size` байт. При увеличении размера выделенного участка данные, которые хранились в старом участке, копируются в новый участок памяти. При уменьшении объема выделенного участка, данные которые хранились в нем, усекаются до нового размера. Функция возвращает указатель на область памяти нового размера.

Работа с одномерным динамическим массивом осуществляется так же, как и с обычным. При этом стиль использования динамических массивов С имеет весомое преимущество над С++, которое заключается в изменении размерности массива. Дело в том, что в С++ нет функций увеличения размерности. Увеличить размер массива можно, создав новый динамический массив нужной размерности, скопировав данные из старого массива в новый и освободив память от старого массива.

8.3. Двумерные динамические массивы

Создание многомерных массивов несколько сложнее, нежели одномерных динамических массивов. Условно двумерный динамических массив можно представить, как на рис. 8.2.

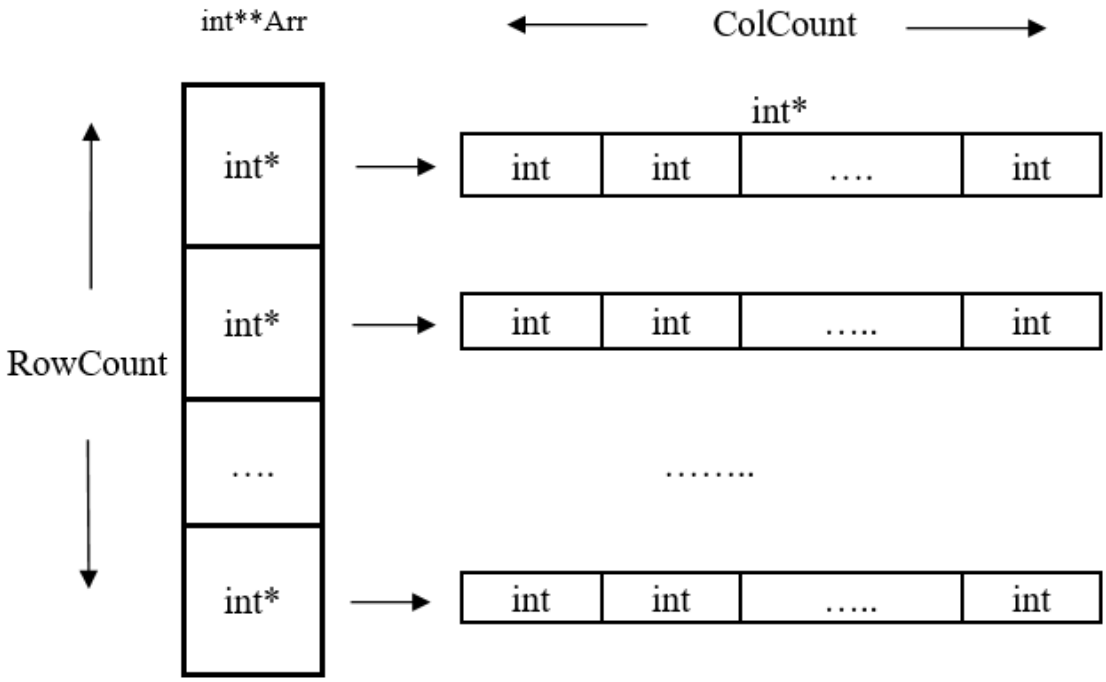


Рис 8.2 – Представление двумерного динамического массива

Чтобы получить двумерный массив, необходимо:

1) создать одномерный динамический массив из `RowCount` указателей на базовый тип элементов массива (в нашем случае указателей на тип `int`);
2) в цикле создать `RowCount` одномерных динамических массивов, каждый из которых содержит `ColCount` элементов базового типа (в нашем случае указателей на тип `int`) и адреса их первых элементов записать в соответствующие элементы “вертикального” массива.

Обычный одномерный массив определяется как указатель на базовый тип данных элементов этого массива. Базовым типом элементов этого массива являются указатели `int*`. Для того чтобы определить указатель на указатель достаточно использовать следующую конструкцию: `(int*)*` или проще `int**`.

Таким образом, для того чтобы создать динамический массив `Arr` из указателей, можно поступить так:

```
int ** Arr = new int * [ RowCount ];
```

Тогда для создания всего двумерного динамического массива необходимо выполнить следующие действия:

```
int ** Arr = new int * [ RowCount ];  // Создаем “вертикальный” массив
for ( int i = 0; i  <  RowCount; ++ i )
     Arr [ i ] = new int [ ColCount ];     // Создаем i-ый массив-строку
```

Для освобождения памяти необходимо:

1. сначала в цикле удалить `RowCount` массивов-строк;
2. затем удалить “вертикальный” массив.

```
for ( int i = 0; i  <  RowCount; ++ i )
     delete [ ] Arr [ i ];              // Удаляем i-ый массив-строку
delete [ ] Arr;                         // Удаляем “вертикальный” массив
```

Для создания динамических двумерных массивов с другими базовыми типами элементов достаточно в предыдущих примерах заменить тип данных `int`, на необходимый тип данных. Ну, и конечно, изменить работу с элементами массива в соответствии с их типом данных. Обязательные места исправлений выделены красным цветом.

По аналогии с двумерными динамическими массивами можно создавать и массивы большей мерности.

8.4. Односвязные линейные списки

Одномерный однонаправленный список представляет собой совокупность отдельных элементов, каждый из которых содержит две части – информационную ($Data$) и адресную ($Tail$). Информационная часть предназначена для хранения “полезных” данных и может иметь практически любой тип. Адресная часть каждого

элемента содержит адрес следующего элемента списка. Схематическое изображение такого списка представлено на рис. 8.3.

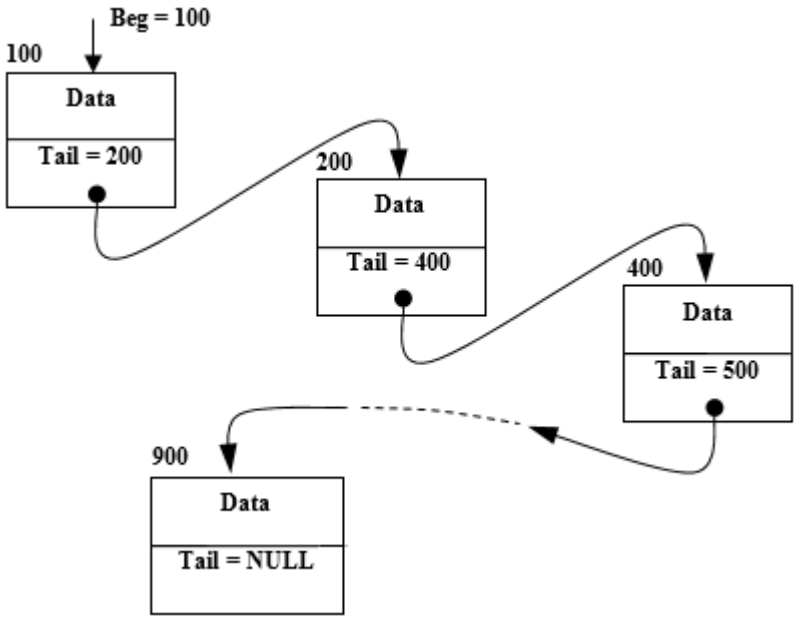


Рисунок 8.3 – Схематическое изображение односвязного списка

Для работы со списком достаточно знать только адрес первого элемента списка (Beg). Зная адрес первого элемента списка можно последовательно получить доступ к любому другому его элементу.

Поскольку каждый элемент списка должен иметь две части, логичнее всего представить его в виде следующей структуры:

```
struct list
{
   int data;
   list *tail;
};
```

Типовыми операциями при работе со списками являются:

- создание списка;
- освобождение памяти от списка (удаление списка);
- доступ к заданному элементу списка для манипуляций с его информационной частью;
- добавление нового элемента к списку;
- удаление элемента из списка;

- перестановка элемента списка на новую позицию внутри списка.

Достоинством подобных структур являются простота добавления, удаления и перестановки элементов списка, которые осуществляются путем манипуляций с адресными частями без перезаписи всего списка.

8.5. Удаление, добавление и обмен элементов односвязного списка

На рис. 8.4 представлена схема удаления первого элемента из односвязного списка.

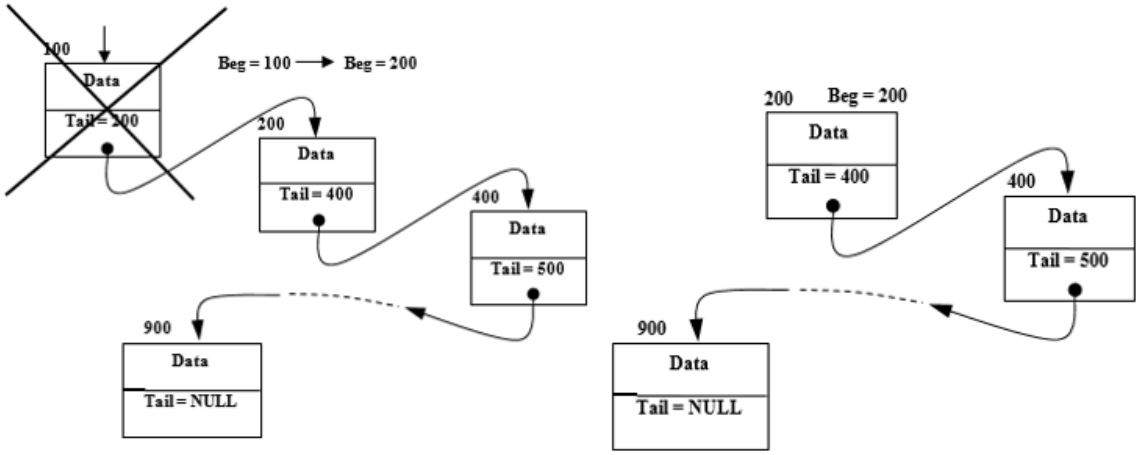


Рис. 8.4 – Схема удаления первого элемента из односвязного списка

Если стоит зада удалить первый элемент связного списка, то необходимо изменить адрес первого элемента списка на адрес следующего за первым элемента и освободить динамическую память от первого элемента.

Схожий принцип действия и у удаления последнего элемента (рис. 8.5).

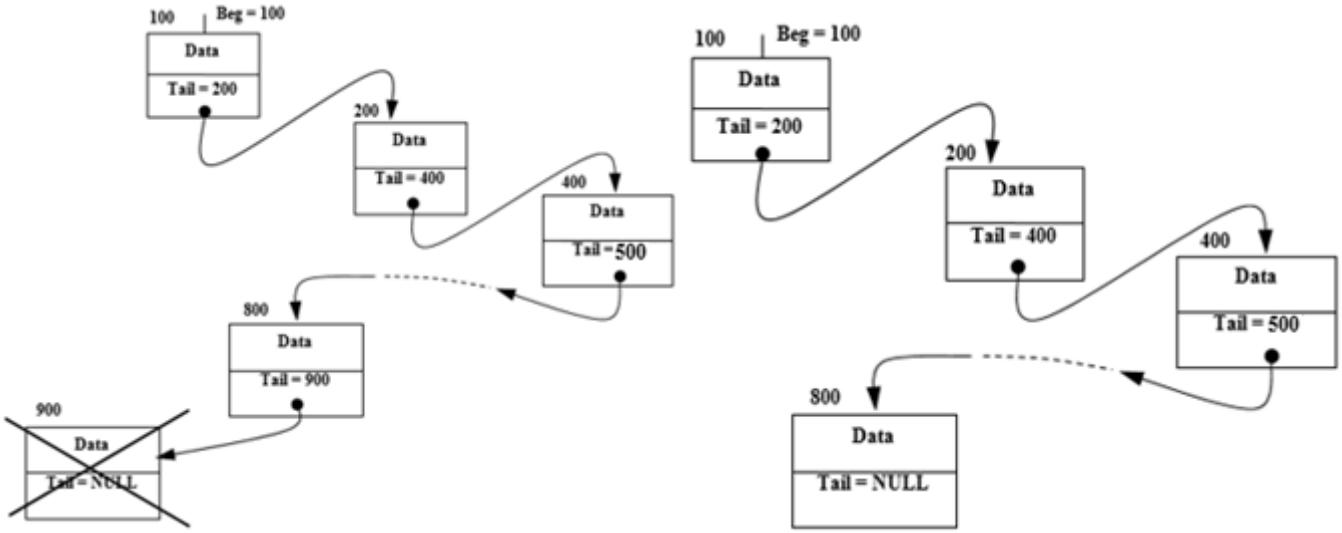


Рисунок 8.5 – Удаление последнего элемента списка

Сначала необходимо изменить адресную часть $N-1$ элемента так, чтобы теперь он имел нулевой указатель, затем нужно освободить память N-го от элемента.

С удалением элемента из середины списка (рис. 8.6) дело обстоит практически так же, как и с удалением с конца списка.

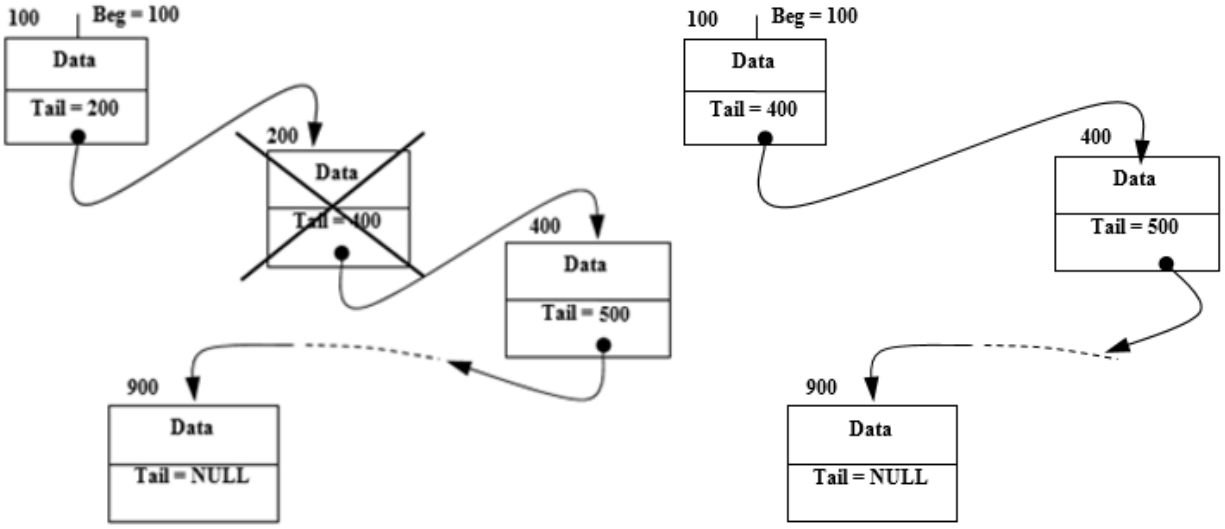


Рисунок 8.6 –Удаление элемента из середины списка

Вставка и обмен элементов односвязного списка происходит по схожему принципу. Более подробно эти операции будут рассмотрены на двусвязных списках.

8.6. Двусвязные линейные списки

Одним из недостатков односвязных списков является то, что узел (элемент списка) имеет указатель только на следующий элемент. Вернутся из текущего элемента к предыдущему явным способом невозможно.

Каждый узел двусвязного (двунаправленного) линейного списка содержит два поля указателей – на следующий и на предыдущий узлы. Указатель на предыдущий узел корня списка содержит нулевое значение. Указатель последнего узла также содержит нулевое значение.

Поскольку каждый элемент списка должен иметь три части, логичнее всего представить его в виде следующей структуры:

```
struct list
{
    int data;
    list *head;
```

```
    list *tail;
};
```

На рис. 8.7 показано схематическое представление двусвязного списка. Поле $Head$ содержит адрес предыдущего элемента, поле $Tail$ содержит адрес следующего элемента списка. Такая организация списка позволяет перемещаться по его элементам в двух направлениях.

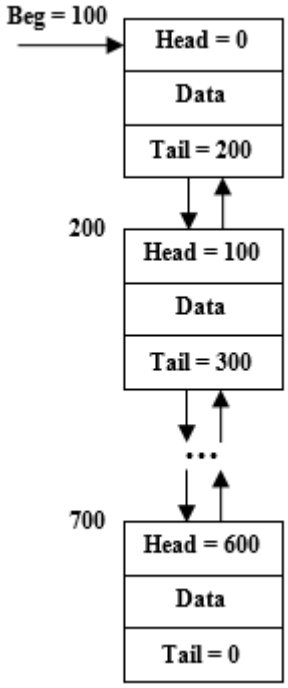


Рисунок 8.7 – Схематическое изображение двусвязного списка

Основные действия, производимые над узлами ДЛС (двусвязного линейного списка):

- инициализация списка;
- добавление узла в список;
- удаление узла из списка;
- удаление корня списка;
- вывод элементов списка;
- вывод элементов списка в обратном порядке;
- взаимообмен двух узлов списка.

Порядок действия очень похож на односвязный линейный список, но необходимо учитывать то, что в двусвязном списке имеется два указателя: на следующий и предыдущий элементы.

8.7. Удаление, вставка, обмен элементов двусвязного списка

Удаление первого элемента и последнего элемента практически аналогично удалению элемента из односвязного списка. Нужно изменить значение указателя на первый элемент, обнулить значение указателя головы/хвоста, освободить динамическую память. С удалением элемента из середины списка (рис. 8.8) дело обстоит сложнее – необходимо проделать аналогичные операции по отношению к двум узлам, а не к одному.

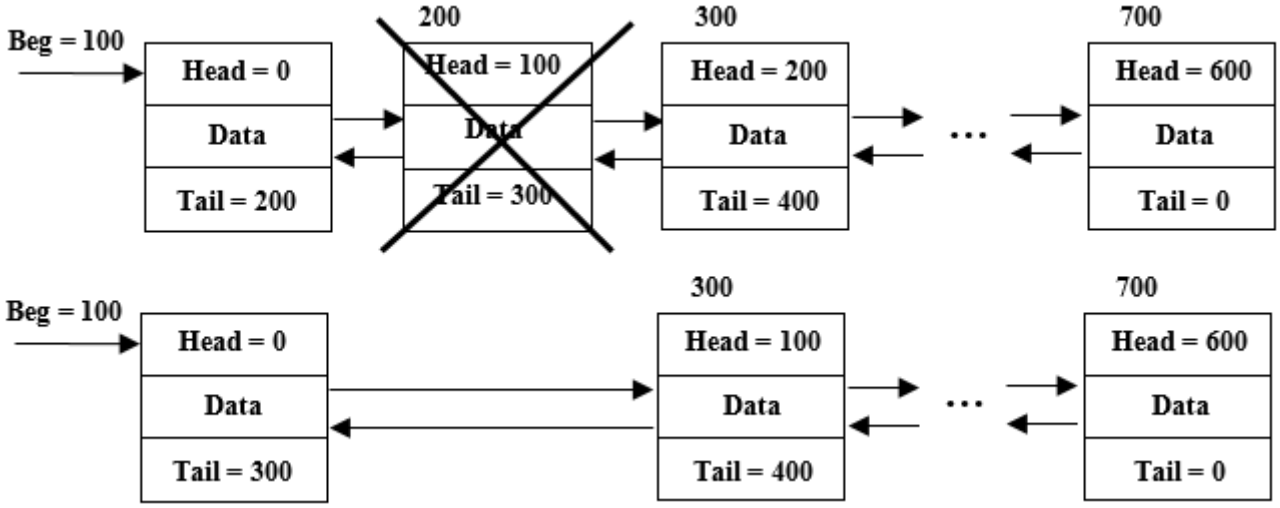


Рисунок 8.8 – Удаление узла из середины двусвязного списка

Чтобы удалить узел из середины списка необходимо изменить указатели $Head$ и $Tail$ предыдущего и следующего элемента, связав их вместе. Затем нужно освободить динамическую память от удаленного элемента.

Со вставкой нового узла в двусвязный линейный список дело обстоит аналогично. Так же присутствует три ситуации: вставка нового корня, вставка последним элементом и вставка узла в середину списка.

На рис. 8.9 изображена вставка нового корневого узла. Видно, что от удаления процедура не особо отличается: необходимо изменить $Head$ указатель прошлого корневого узла, указатель на корневой узел и правильно связать новый узел со списком.

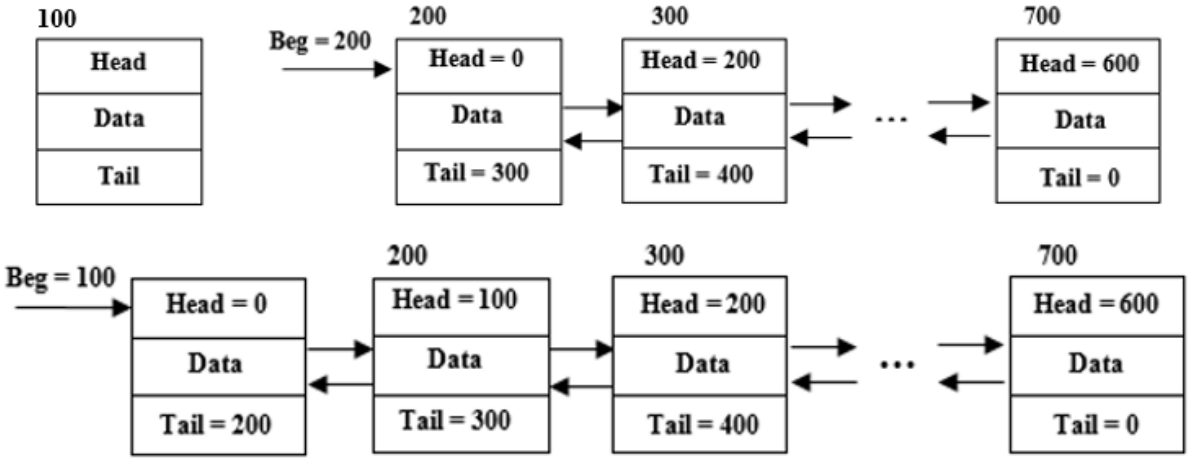


Рисунок 8.9 – Вставка корневого узла

Вставка узла в конец списка не должна вызывать затруднений: необходимо изменить $Tail$ указатель прежнего последнего элемента на адрес нового узла, затем правильно связать новый узел с двусвязным списком ($Head$ – адрес на предыдущий элемент, $Tail$ – `NULL` указатель). Вставка же узла в середину списка представлена на рис. 8.10.

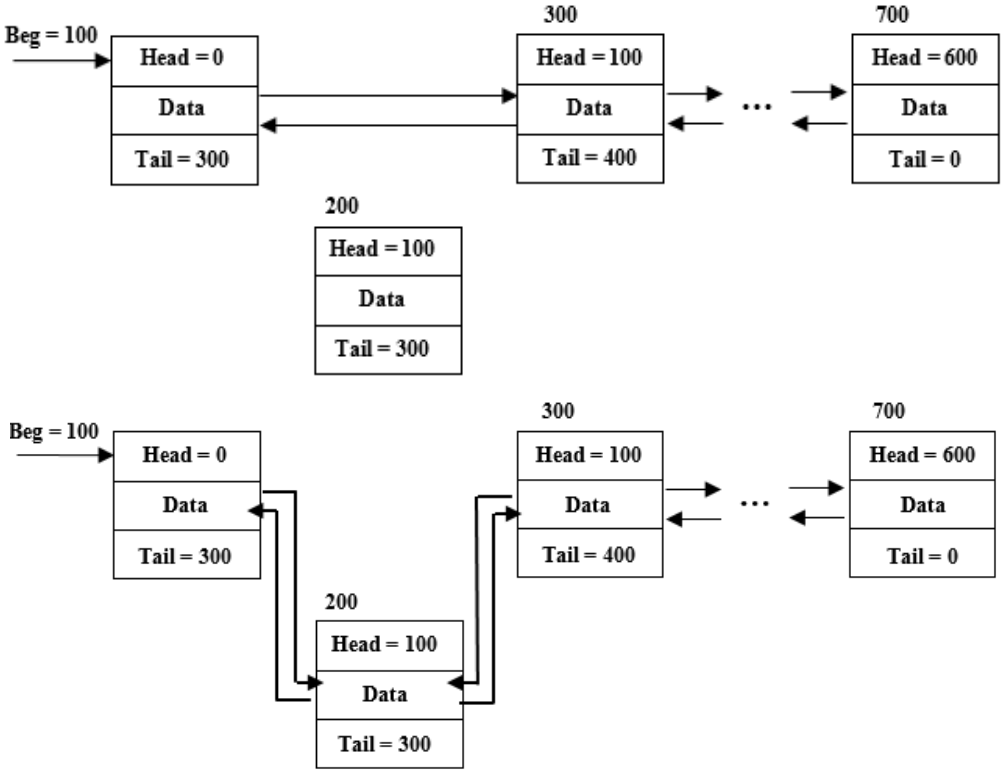


Рисунок 8.10 – Вставка узла в середину двусвязного списка

Больше всего трудностей может возникнуть именно с взаимообменом элементов (рис 8. 11) двусвязного списка. Главная проблема – возможная путаница в указателях. Нужно правильно изменить указатели всех затрагиваемых узлов.

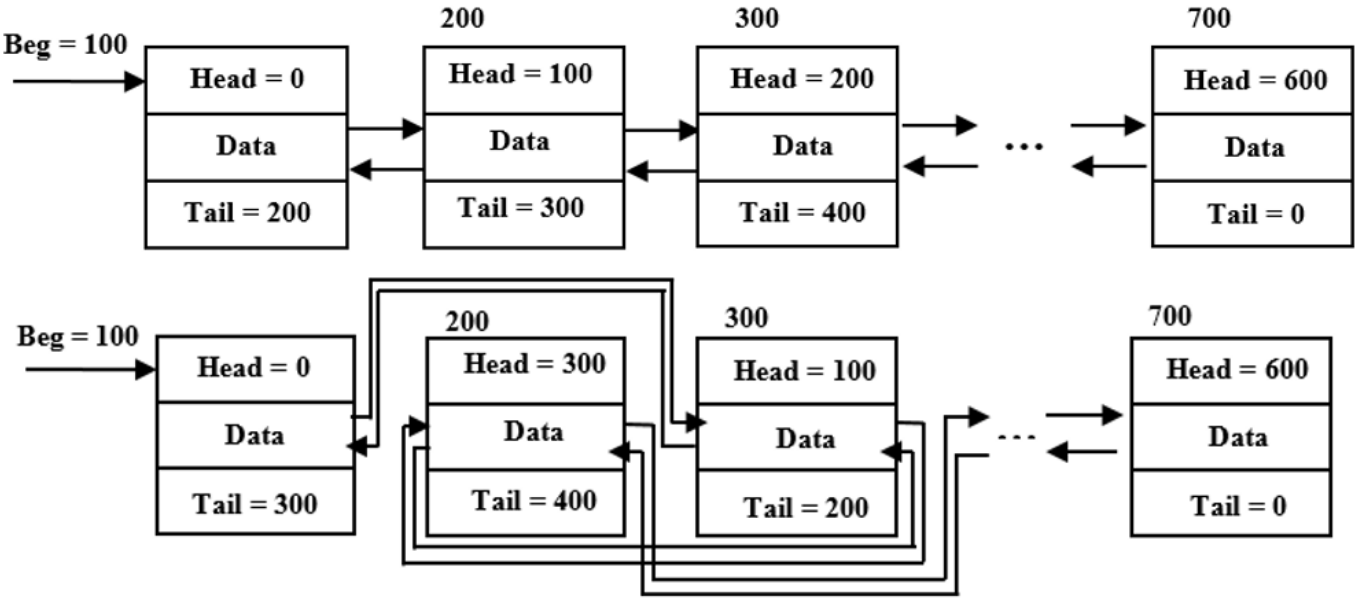


Рисунок 8.11 – Взаимообмен узлов двусвязного списка

Видно, сколько необходимо изменить адресов указателей, чтобы двумерный список работал корректно. Если хоть один указатель будет ссылаться неправильно, то список будет работать некорректно и может вовсе зациклиться. Причем программист этого может не заметить, если поле $Data$ однородно или вовсе отсутствует. Поэтому (как и с любыми указателями) нужно быть внимательным.

8.8. Задание к практической работе

Необходимо реализовать программу, которая выполняет следующие действия:

1) Формирование целочисленного одномерного массива размерности N, где:
   а) пользователь вводит количество элементов в массиве, который будет автоматически заполняться случайными числами (0 до 99);
   б) пользователь вводит в консоль элементы массива, N определяется автоматически по количеству введенных элементов;
   в) массив считывается с файла, N определяется как количество элементов массива в файле.
2) Определение скорости создания динамического массива п. 1.
3) Вставка, удаление и получение элемента массива. Удаление и получение элемента необходимо реализовать по индексу и по значению.

4) Определение скорости вставки, удаления и получения элемента массива п. 3.
5) Формирование двусвязного списка размерности N, где:
   а) пользователь вводит количество элементов в списке, который будет автоматически заполняться случайными числами (0 до 99);
   б) пользователь вводит в консоль элементы списка, N определяется автоматически по количеству введенных элементов;
   в) список считывается с файла, N определяется как количество элементов списка в файле.
6) Определение скорости создания двусвязного списка п. 6.
7) Вставка, удаление и получение элемента двусвязного списка. Удаление и получение элемента необходимо реализовать по индексу и по значению.
8) Определение скорости вставки, удаление и получения элемента двусвязного списка п. 8.
9) Задание по варианту табл. 8.1.

Должна присутствовать возможность запуска каждого пункта многократно, если есть возможность (если в списке/массиве нет элементов, то нельзя ничего удалить и об этом нужно сообщить пользователю). Необходимо сравнить результаты. Для этого пункты 1-4 и 5-9 должны принимать одинаковые значения.

После выполнение всех заданий необходимо сделать вывод по проделанной работе.

Таблица 8.1 – Индивидуальные задания

Вариант	Задание
1	Реализуйте и сравните скорость работы bubble sort для исходного динамического массива и двусвязного списка.
2	Реализуйте и сравните скорость работы shaker sort для исходного динамического массива и двусвязного списка.
3	Выведите на экран сколько чётных элементов имеют четные значения, сколько нечетных элементов имеют нечетные значения. Сравните скорость работы динамического массива и двусвязного списка.

Продолжение таблицы 8.1

4	Уменьшите каждый нечетный элемент на значение, введенное пользователем. Каждый четный элемент умножьте на случайную величину в диапазоне от 1 до 5. Сравните скорость работы динамического массива и двусвязного списка.
5	Уменьшите каждый четный элемент на значение, введенное пользователем. Каждый нечетный элемент умножьте на случайную величину в диапазоне от 1 до 5. Сравните скорость работы динамического массива и двусвязного списка.
6	Для двусвязного списка составьте несколько вариантов проверочной работы. Варианты проверочной работы необходимо вывести в файл test, ответы к проверочной работе необходимо вывести в файл reply. Задание проверочной работы: - удаление элемента(-ов) из двусвязного списка; - вставка элемента(-ов) в двусвязный список. Обучающий, выполняющий проверочную работу, должен поэтапно объяснить ход действий при вставке и удалении.
7	Выведите количество элементов, которые нацело делятся на 1, 2, 3, 4, 5, 6, 7, 8, 9. Сравните скорость работы динамического массива и двусвязного списка.
8	Удалите четные элементы в динамическом массиве и четные элементы в двусвязном списке. Сравните скорость выполнения операций.
9	Удалите нечетные элементы в динамическом массиве и нечетные элементы в двусвязном списке. Сравните скорость выполнения операций.
10	Увеличьте каждое значение исходного динамического массива на случайное число (в диапазоне от 0 до 5). Добавьте в двусвязный список все положительные элементы динамического массива.
11	Увеличьте каждое значение исходного двусвязного списка на случайное число (в диапазоне от 0 до 5). Добавьте в динамический массив все положительные элементы динамического массива.
12	Увеличьте каждое значение исходного динамического массива на случайное число (в диапазоне от 0 до 5). Добавьте в двусвязный список все отрицательные элементы динамического массива.
13	Увеличьте каждое значение исходного двусвязного списка на случайное число (в диапазоне от 0 до 5). Добавьте в динамический массив все отрицательные элементы динамического массива.

Окончание таблицы 8.1

14	Создайте еще один двусвязный список размерности N, заполненный случайными числами (от 0 до 5). Соедините два списка в один. Причем значение узла второго двусвязного списка должно быть меньше, чем значение узла первого списка. Если условие не выполняется, проверяется следующий узел первого списка. Если условие ложно для всех узлов перового списка, то узел добавляется в конец списка.
15	Поменяйте местами все узлы исходного двусвязного списка и все значения исходного динамического массива. Сравните скорость работы.
16	Поменяйте k случайных узлов исходного двусвязного списка и k значений исходного динамического массива. Сравните скорость работы. k – введенное пользователем число (k должно быть меньше N).
17	Удалите все узлы исходного двусвязного списка и все значения исходного динамического массива. Сравните скорость работы.
18	Удалите k случайных узлов исходного двусвязного списка и k значений исходного динамического массива. Сравните скорость работы. k – введенное пользователем число (k должно быть меньше N).
19	Добавьте k случайных узлов исходного двусвязного списка и k значений исходного динамического массива. Сравните скорость работы выполнения операций. k – введенное пользователем число (k должно быть меньше N). Позиция добавления должна быть случайна.
20	Создайте односвязный список. Проделайте с ним те же действия, как и в п. 1-3. Сравните скорость работы односвязного списка и двусвязного списка.

ПРАКТИЧЕСКАЯ РАБОТА №9
ПОЛЬСКИЕ НОТАЦИИ. СТЕК И ОЧЕРЕДЬ

Цель работы: получение практических навыков работы со стеками и очередями. Изучение обратной и прямой польской нотации. Проведение сравнительного анализа этих структур данных.

9.1. Понятие стека

Стек – это частный случай однонаправленного списка, добавление элементов в который и выборка их которого выполняется с одного конца, называемого вершиной стека. Другие операции со стеком не определены. При выборке элемент исключается из стека. Говорят, что стек реализует принцип обслуживания LIFO (последним пришел – первым ушел).

Основные операции над стеками:

1) чтение верхнего элемента;
2) добавление нового элемента;
3) удаление существующего элемента.

Графически его удобно изобразить в виде вертикального списка (см. рис. 9.1), например, стопки книг, где чтобы воспользоваться одной из них, и не нарушить установленный порядок, нужно поднять все те книги, что лежат выше нее, а положить книгу можно лишь поверх всех остальных.

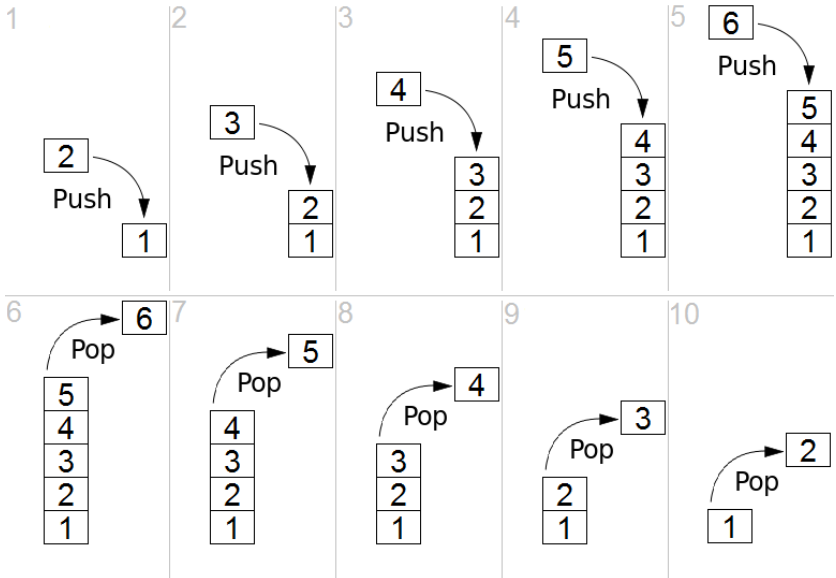


Рисунок 9.1 – Графическое представление стека

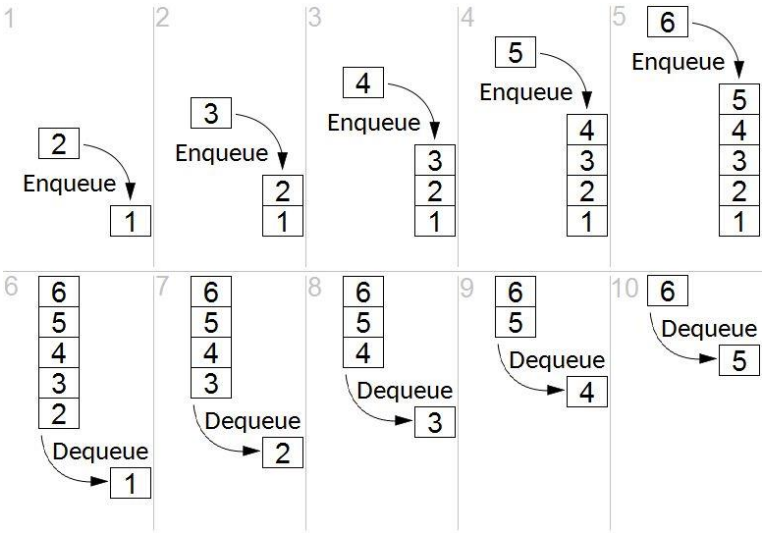
Стек, чаще всего, реализуется на основе обычных массивов, односвязных и двусвязных списков. В зависимости от конкретных условий, выбирается одна из этих структур данных.

9.2. Понятие очереди и кольцевой очереди

Очередь – частный случай однонаправленного списка, добавление элементов в который выполняется в один конец, а выборка – из другого конца. Другие операции с очередью не определены. При выборке элемент исключается из очереди. Говорят, что очередь реализует принцип обслуживания FIFO (первым пришел – первым ушел). В программировании очереди применяются при моделировании, диспетчеризации задач операционной системой, буферизованном вводе/выводе.

Графически её удобно изобразить в виде вертикального списка (см. рис. 9.2), например, очередь в магазине, где чтобы дойти до кассы, и не нарушить установленный порядок, нужно дождаться, пока все клиенты перед вами не купят товар. Разумеется, будут появляться новые покупатели, которые будут занимать свое место в очереди в ожидании покупки.

Рисунок 9.2 – Графическое представление очереди



Основные операции над очередями:

1) чтение первого элемента;
2) добавление нового элемента;
3) удаление существующего элемента.

Если для стека в момент добавления или удаления элемента допустимо задействование лишь его вершины, то касательно очереди эти две операции должны быть применены так, как это регламентировано в определении этой структуры данных, т. е. добавление – в конец, удаление – из начала.

Выделяют два способа программной реализации очереди. Первый из них основан на базе массива, а второй на базе указателей (связного списка). Первый способ – статический, т. к. очередь представляется в виде простого статического массива, второй – динамический.

Кольцевой буфер также известен, как очередь или циклический буфер и является распространенной формой очереди. Это популярный, легко реализуемый стандарт, и, хотя он представлен в виде круга (рис. 9.3), в базовом коде он является линейным. Кольцевая очередь существует как массив фиксированной длины с двумя указателями: один представляет начало очереди, а другой - хвост.

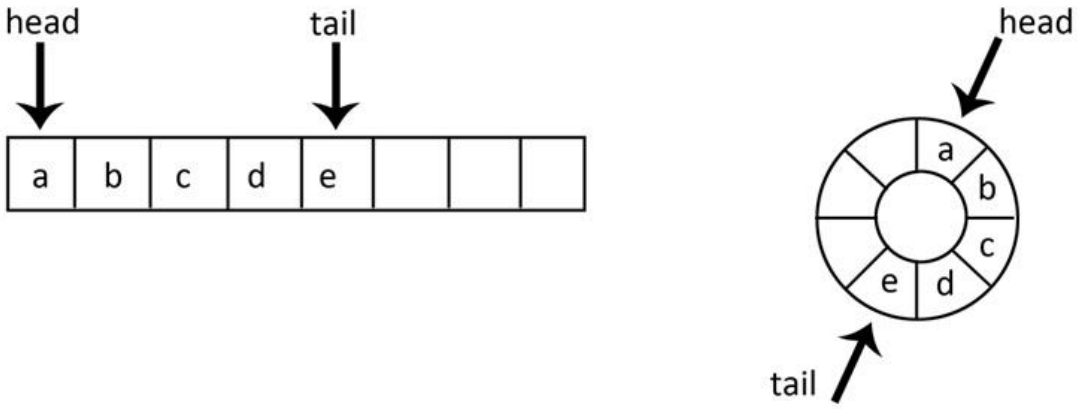


Рисунок 9.3 – Кольцевая очередь

Операции с очередями работают следующим образом:

- Два указателя, называемые $HEAD$ и $TAIL$, используются для отслеживания первого и последнего элементов в очереди.
- При инициализации очереди значения $HEAD$ и $TAIL$ устанавливаются равными -1.
- При добавлении элемента постепенно увеличивается значение индекса $TAIL$ и помещается новый элемент в положение, на которое указывает $TAIL$.
- При снятии очереди с элемента возвращается значение, на которое указывает $HEAD$, и постепенно увеличивается индекс $HEAD$.

- Перед постановкой в очередь проверяется, заполнена ли очередь.
- Перед снятием очереди мы проверяется, пуста ли очередь.
- При инициализации первого элемента устанавливается значение $HEAD$ в 0.
- При удалении последнего элемента сбрасываются значения $HEAD$ и $TAIL$ в -1.

Недостатком метода является его фиксированный размер. Для очередей, где элементы должны быть добавлены и удалены в середине, а не только в начале и конце буфера, реализация в виде связанного списка является предпочтительным подходом.

9.3. Обратная польская нотация

Обратная польская запись (рис. 9.4) – форма записи математических и логических выражений, в которой операнды расположены перед знаками операций. Обратная польская запись имеет ряд преимуществ перед инфиксной записью при выражении алгебраических формул. Одно из которых то, что инфиксные операторы имеют приоритеты, которые произвольны и нежелательны.

Простое выражение	Прямая польская запись	Обратная польская запись
X + 3 * Y	+ X * 3 Y	X 3 Y * +
(X + 3) * Y	* + X 3 Y	X 3 + Y *
1 +2	+ 1 2	1 2 +

Рисунок 9.4 – Представление прямой и обратной польской записи

Обратная польская запись отлично подходит для вычисления выражений при помощи стека. Причем сам алгоритм достаточно прост. Необходимо просто прочитать обратную польскую запись слева направо. Если встречается операнд, то его нужно поместить в стек. Если встречается оператор, нужно выполнить заданную им операцию.

9.4. Задание к практической работе

Необходимо написать программу, которая:

1) Реализует преобразование введенного выражения (если используются переменные, то пользователь должен их инициализировать). Ввод

выражения должен быть реализовать двумя способами: с клавиатуры и с файла.

2) Реализует проверку на корректность простого выражения и выражения, записанного в прямой и обратной польским нотациям (на выбор пользователя). Ввод выражения должен быть реализовать двумя способами: с клавиатуры и с файла.
3) Реализует вычисления простого выражения и выражения, записанного в прямой и обратной польским нотациям (на выбор пользователя). Ввод выражения должен быть реализовать двумя способами: с клавиатуры и с файла.
4) Генерирует несколько (на выбор пользователя) вариантов проверочной работы по польской нотации (прямой и обратной). Задание и ответы к ним необходимо вывести в отдельные файлы (ответы должны быть максимально подробными).

Реализуйте программу выше двумя способами: с использованием стека и массивов. Также сравните скорость вычисления, создания и проверки на корректность выражения в прямой и обратной польских нотациях.

Выполните индивидуальное задание (табл. 9.1) согласно вашему варианту, после чего сделайте вывод по проделанной работе.

Таблица 9.1 – Задания по вариантам

Вариант	Задание
1	Реализуйте «электронную» очередь в магазине.
2	Реализуйте стек для нескольких стоп книг.
3	Реализуйте буфер клавиатуры.
4	Реализуйте стек. Заполните его случайными положительными и отрицательными числами. Преобразуйте стек в два стека. Первый должен содержать только положительные числа, второй – отрицательные.
5	Реализуйте очередь. Заполните её случайными положительными и отрицательными числами. Преобразуйте очередь в две очереди. Первая должен содержать только положительные числа, вторая – отрицательные.

ПРАКТИЧЕСКАЯ РАБОТА №10
БИНАРНОЕ ДЕРЕВО ПОИСКА

Цель работы: Изучение свойств и организации деревьев как структуры данных. Получение практических навыков в работе с бинарным деревом поиска. Определение преимуществ и недостатков структуры данных вида дерева. Проведение сравнительной характеристики скорости вставки, удаление и поиска элемента в различных структурах данных.

10.1. Понятие бинарного дерева

Бинарное дерево – это динамическая структура данных, состоящая из узлов, каждый из которых содержит, кроме данных, не более двух ссылок на различные бинарные деревья. На каждый узел имеется ровно одна ссылка. Начальный узел называется корнем дерева.

На рис. 9.1 приведен пример бинарного дерева. Узел, не имеющий поддеревьев, называется листом. Исходящие узлы называются предками, входящие – потомками. Высоте дерева определяется количеством уровней, на которых располагаются его узлы.

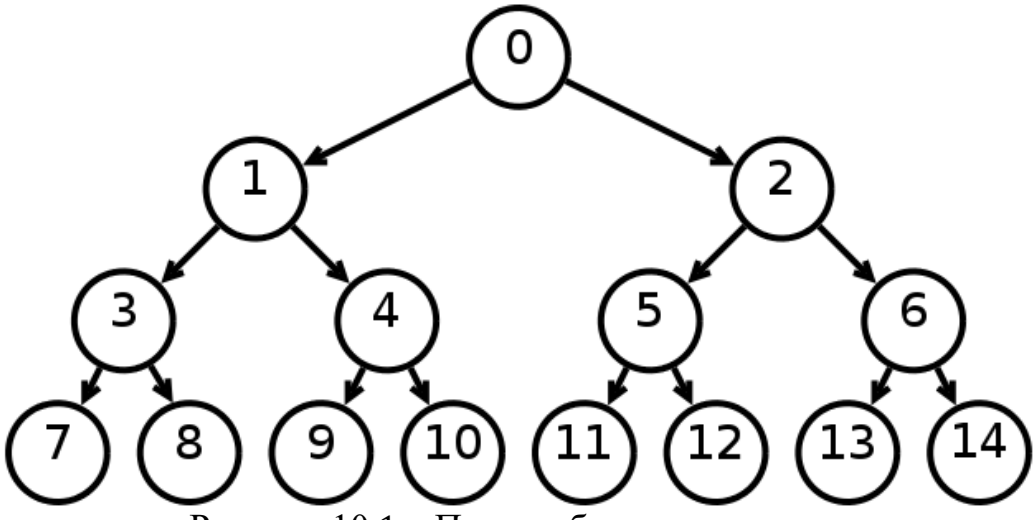


Рисунок 10.1 – Пример бинарного дерева

Дерево является рекурсивной структурой данных, поскольку каждое поддерево также является деревом. Действия с такими структурами лучше всего описывать с помощью рекурсивных алгоритмов.

10.2. Понятие бинарного дерева поиска

Бинарное дерево поиска – бинарное дерево, которое обладает дополнительными свойствами. Если дерево организовано таким, что для каждого узла все ключи левого поддерева меньше ключа этого узла, а все ключи его правого поддерева – больше, то его можно называть бинарным деревом поиска. Иными словами, данные в бинарном дереве поиска хранятся в отсортированном виде. Одинаковые ключи не допускаются.

Пример бинарного дерева поиска представлен на рис. 9.2.

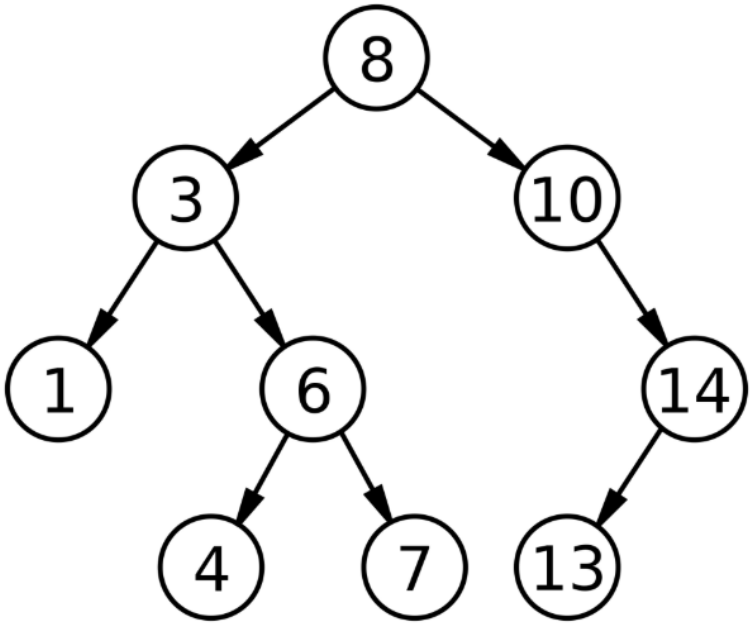


Рисунок 10.2 – Бинарное дерево поиска

При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое значение больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

Поскольку каждый корень дерева должен иметь три части, логичнее всего представить его в виде следующей рекурсивной структуры:

```
struct Node{
   int key;
   Node *left;
   Node *right;
 };
```

Сбалансированное бинарное дерево поиска – это бинарное дерево поиска с логарифмической высотой. Для данного случая это правило не обязательно, в отличие от AVL или красно-чёрных деревьев, где строгое определение оперирует разницей глубины самого глубоко и самого неглубокого листа и отношением глубины самого глубоко и самого неглубокого листа соответственно.

Для бинарных деревьев определены следующие операции:

- добавление узла в дерево;
- поиск по дереву;
- обход дерева;
- удаление узла.

Сбалансированное бинарное дерево поиска применяется, когда необходимо осуществлять быстрый поиск элементов, чередующийся со вставками новых элементов и удалением существующих.

10.3. Вставка и удаление узла из дерева

Так как одинаковые ключи не допускаются, перед вставкой нового элемент нужно проверить, нет ли уже в бинарном дереве такого элемента. Для этого необходимо начать обход дерева с корневого узла и проверить, не превосходит ли значение корневого узла добавляемого значения. Если корневой узел больше добавляемого элемента, то необходимо переместиться в левое дочернее дерево. В противном случае - в правое.

После выполнения данной проверки можно переходить к добавлению узла в бинарное дерево. Функция вставки аналогична функции поиска: необходимо пройти по дереву и вставить узел в нужное место. Для этого следует выбрать левое или правое поддерево корневого узла, а затем рекурсивно перемещаться по выбранному поддереву до тех пор, пока не будет найдена позиция для вставки узла.

С удалением узла дело обстоит немного иначе. Существует несколько возможных ситуаций:

1) У узла нет наследников (узел является листом).
2) У узла имеется наследник при том только один.

3) У узла есть оба наследника.

Если у узла нет наследника (рис. 9.3), то необходимо просто удалить этот узел, а у его родителя обнулить указатель на него. Это самая простоя ситуация.

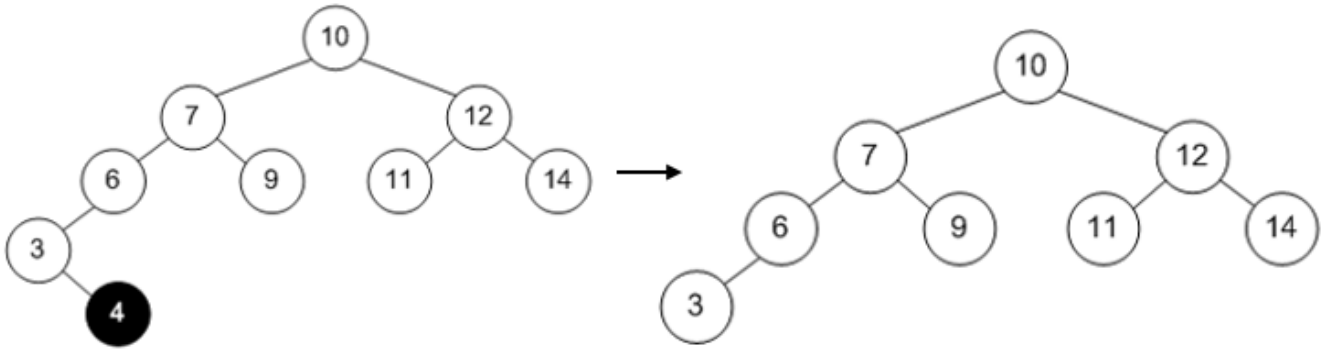


Рисунок 10.3 – Удаление узла “4” из бинарного дерева поиска

Все слегка усложняется, когда у удаляемого узла имеются наследники. Если у узла один наследник, то узел необходимо подменить своим наследником (рис. 9.4).

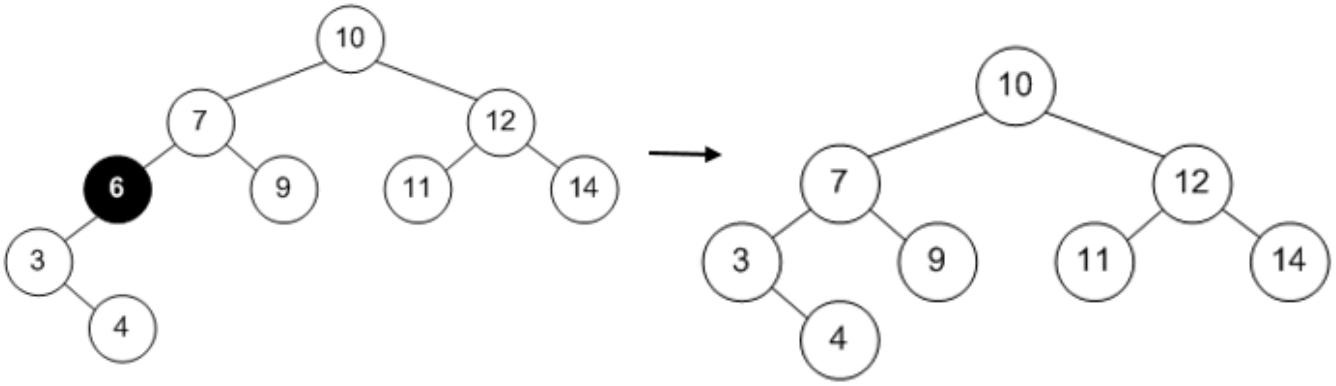


Рисунок 10.4 – Удаление узла “6” из бинарного дерева поиска

Если же у удаляемого два наследника (рис. 9.5), то узле не удаляется, а заменяется его значение на максимум левого поддерева. После чего удаляется максимум левого поддерева.

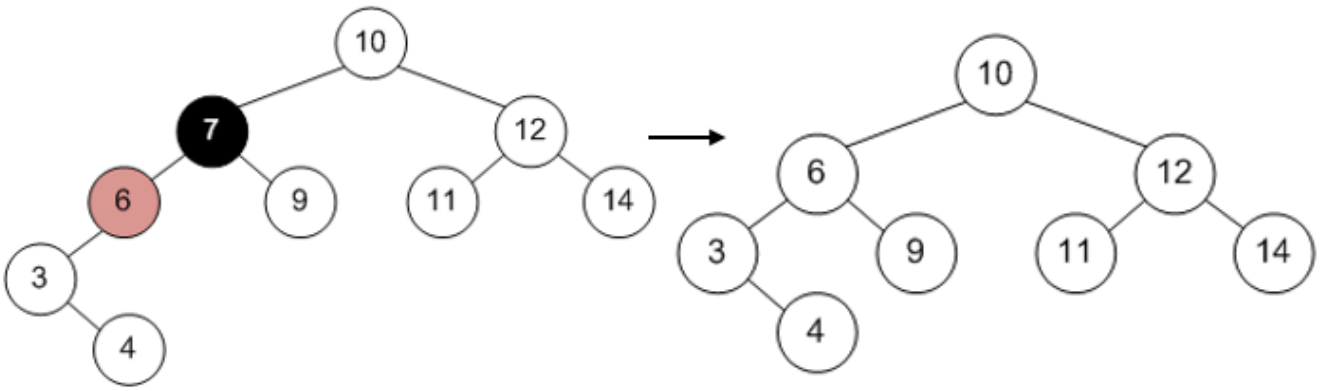


Рисунок 10.5 – Удаление узла “7” из бинарного дерева поиска

Максимум левого поддерева имеет не более одного наследника, так что он удаляется просто, аналогично ситуации, рассмотренной выше.

10.4. Обходы дерева

Граф – это множество вершин и ребер. Ребро – это связь между двумя вершинами. Количество возможных ребер в графе квадратично зависит от количества вершин (для понимания можно представить турнирную таблицу сыгранных матчей).

Дерево – это связный граф без циклов. Связность означает, что из любой вершины в любую другую существует путь по ребрам. Отсутствие циклов означает, что данный путь – единственный.

Обход графа – это систематическое посещение всех его вершин по одному разу каждой. Существует два вида обхода графа:

1) поиск в глубину;

2) поиск в ширину.

Поиск в ширину идет из начальной вершины, посещает сначала все вершины, находящиеся на расстоянии одного ребра от начальной, потом посещает все вершины на расстоянии два ребра от начальной и так далее. Алгоритм поиска в ширину является по своей природе нерекурсивным (итеративным). Для его реализации применяется структура данных очередь (FIFO).

Поиск в глубину идет из начальной вершины, посещая еще не посещенные вершины без оглядки на удаленность от начальной вершины. Алгоритм поиска в глубину по своей природе является рекурсивным. Для эмуляции рекурсии в итеративном варианте алгоритма применяется структура данных стек.

Обходу в ширину в графе соответствует обход по уровням бинарного дерева. При данном обходе идет посещение узлов по принципу сверху вниз и слева направо. Обходу в глубину в графе соответствуют три вида обходов бинарного дерева: прямой, симметричный и обратный (рис. 9.6).

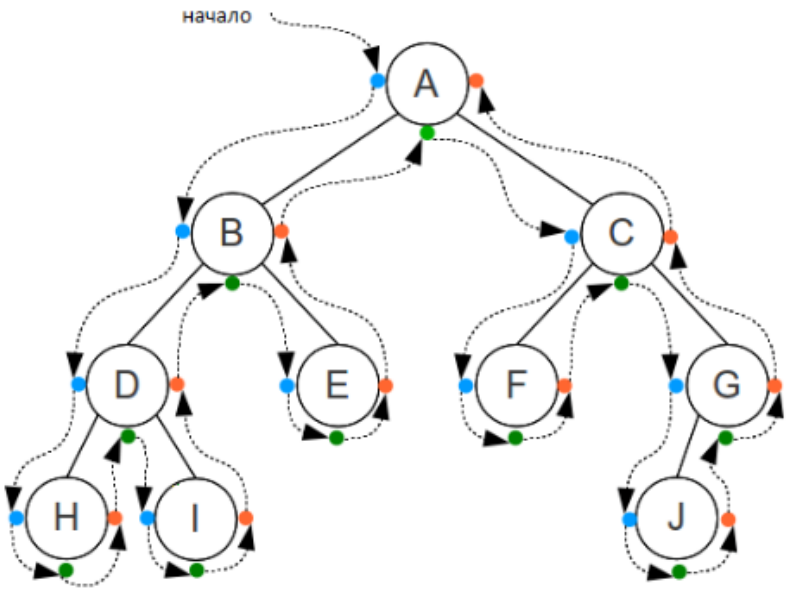


Рисунок 10.6 – Прямой, симметричный, обратный обходы дерева

Прямой обход: ABDHIECFGJ.

Симметричный обход: HDIBEAFCJG.

Обратный обход: HIDEBFJGCA.

Прямой обход идет в следующем порядке: корень, левый потомок, правый потомок. Симметричный — левый потомок, корень, правый потомок. Обратный – левый потомок, правый потомок, корень. В коде рекурсивной функции соответствующего обхода сохраняется соответствующий порядок вызовов (порядок строк кода), где вместо корня идет вызов данной рекурсивной функции.

Для бинарных деревьев поиска симметричный обход проходит все узлы в отсортированном порядке. Если мы хотим посетить узлы в обратно отсортированном порядке, то в коде рекурсивной функции симметричного обхода следует поменять местами правого и левого потомка.

10.5. Задание к практической работе

Необходимо реализовать программу, которая выполняет следующие действия:

1) Формирование бинарного дерева из N элементов:

   а) пользователь вводит количество элементов N бинарного дерева, которое автоматически заполняется случайными числами (-99 до 99);

   б) пользователь вводит в консоль элементы массива, N определяется

автоматически по количеству введенных элементов;

в) элементы считываются с файла, в котором хранится массив из чисел, N – количество элементов в файле.

Определение скорости формирования бинарного дерева.

2) Вывод в консоль и файл tree бинарного дерева. Бинарное дерево должно иметь подобный вид:

```
Бинарное дерево:
         .-->8
    .-->1
-->-3
    `-->-6
         `-->-18
```

3) Определение скорости вставки, удаления и получения элемента массива. Сравните скорость работы бинарного дерева с линейной структурой (двусвязным списком или динамическим массивом).
4) Прямой обход, обратный обход и обход в ширину бинарного дерева.
5) Задание по варианту табл. 10.1.

При выполнении пунктов не должны нарушаться правила построения бинарного дерева.

Должна присутствовать возможность запуска каждого пункта многократно, если есть возможность (если в дереве нет элементов, то нельзя ничего удалить и об этом нужно сообщить пользователю).

После выполнение всех заданий необходимо сделать вывод по проделанной работе.

Таблица 10.1 – Задания по вариантам

Вариант	Задание
1	Уменьшите каждый нечетный элемент на значение, введенное пользователем. Каждый четный элемент умножьте на случайную величину в диапазоне от 1 до 5. Сравните скорость работы линейной структуры (динамический массив/двусвязный список) и бинарного дерева.

Окончание таблицы 10.1

2	Создайте еще одно бинарное дерево размерности N, заполненное случайными числами (от – 99 до 99). Соедините два дерево в одно. Сравните скорость выполнение объединения при аналогичных обстоятельствах с линейной структурой (динамический массив/двусвязный список)
3	Удалите все отрицательные числа. Каждый нечетный элемент умножьте на случайную величину в диапазоне от -2 до 2. Сравните скорость работы линейной структуры (динамический массив/двусвязный список) и бинарного дерева.
4	Выведите количество элементов, которые нацело делятся на 1, 2, 3, 4, 5, 6, 7, 8, 9. Сравните скорость работы линейной структуры (динамический массив/двусвязный список) и бинарного дерева.
5	Составьте несколько вариантов проверочной работы. Варианты проверочной работы необходимо вывести в файл test, ответы к проверочной работе необходимо вывести в файл reply. Задание проверочной работы: - удаление элемента(-ов) из бинарного дерева; - вставка элемента(-ов) в бинарное дерево. Обучающий, выполняющий проверочную работу, должен поэтапно объяснить ход действий при вставке и удалении. Варианты должны быть одинаковыми по сложности.

ПРАКТИЧЕСКАЯ РАБОТА №11
СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ

Цель работы: Изучение свойств и организации сбалансированных деревьев. Получение практических навыков в работе с АВЛ-деревьями. Определение преимуществ и недостатков подобных структур данных. Проведение сравнительной характеристики скорости вставки, удаление и поиска элемента в АВЛ-деревах.

11.1. Понятие АВЛ-дерева

Сбалансированным называется такое двоичное дерево поиска, в котором высота каждого из поддеревьев, имеющих общий корень, отличается не более чем на некоторую константу, и при этом выполняются условия характерные для двоичного дерева.

АВЛ-дерево – сбалансированное двоичное дерево поиска с константой отличия, равной 1. Для узлов АВЛ-дерева определен коэффициент сбалансированности. Это разность высот правого и левого поддеревьев, принимающая одно значения из множества {-1, 0, 1}. Пример АВЛ-дерева представлен на рис. 10.1. Вместо привычных значений узла отображены соответствующие коэффициенты сбалансированности.

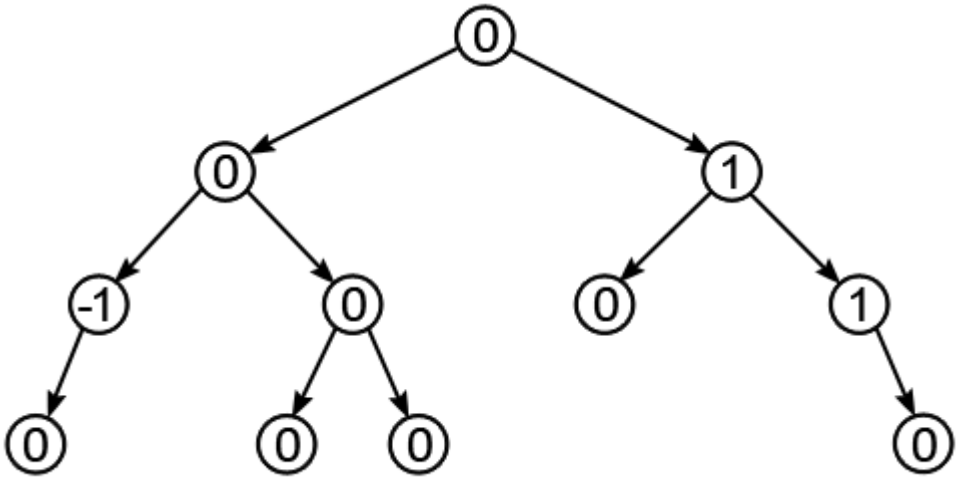


Рисунок 11.1 – Пример АВЛ-дерева

Отличительной особенностью сбалансированного дерева является эффективность в обработке. Действительно, ведь максимальное количество шагов,

которое необходимо для обнаружения нужного узла, равно количеству уровней самого бинарного дерева поиска. Так как поддеревья практически симметричны, высота дерева сводится к оптимальному минимуму. Это позволяет ускорить поиск, что сказывается на общей производительности.

Для АВЛ-деревьев определены следующие основные операции:

- добавление узла в дерево;
- поиск по дереву;
- удаление узла;
- балансировка дерева.

Поскольку каждый корень дерева должен иметь четыре части, логичнее всего представить его в виде следующей рекурсивной структуры:

```
struct Node{
   int key;
   int height;
   Node *left;
   Node *right;
};
```

Структура `Node` описывает узлы АВЛ-дерева. Ее поля $right$ и $left$ являются указателями на правое и левое поддеревья. Поле key хранит ключ узла, $height$ – высоту поддерева.

11.2. Балансировка АВЛ-дерева

В процессе обработки АВЛ-дерева, балансировка может нарушиться, тогда потребуется осуществить операцию балансировки. Если после выполнения операций вставки или удаления, коэффициент сбалансированности какого-либо узла АВЛ-дерева становиться равен 2, то необходимо выполнить операцию балансировки.

Операция балансировки осуществляется путем поворота узлов (изменения связей в поддереве). Вращения не меняют свойств бинарного дерева поиска, различают несколько типов вращения:

1) малый правый поворот;
2) большой правый поворот;
3) малый левый поворот;

4) большой левый поворот.

Оба типа больших поворотов являются комбинацией малых вращений (право-левым или лево-правым поворотом).

Возможны два случая нарушения сбалансированности. Один из них исправляется малым правым и малым левым поворотами (рис. 10.2), другой – большим правым и большим левым поворотами (рис 10.3).

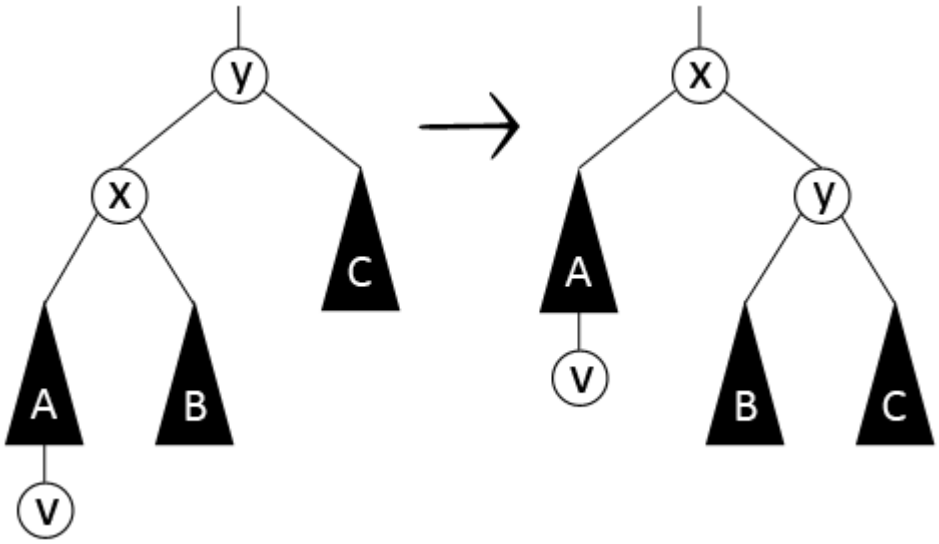


Рисунок 11.2 – Малый правый поворот

Здесь x и y – узлы, а A, B, C – поддеревья. После добавления к поддереву A узла v, баланс нарушиться, и потребуется балансировка. Она осуществляется правым поворотом у узла y. Малый левый поворот выполняется симметрично малому правому.

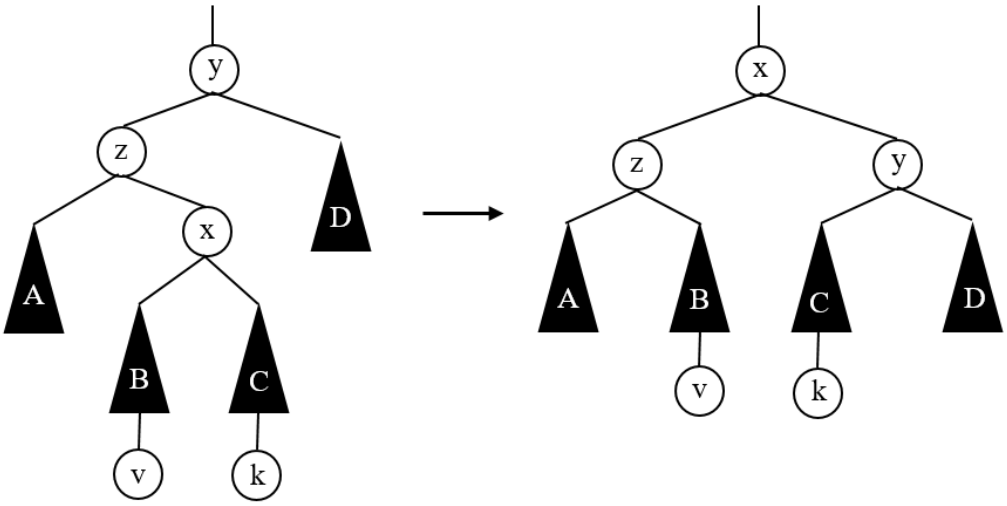


Рисунок 11.3 – Большой правый поворот

Вставка узлов в поддерево A или D не нарушит сбалансированности, но добавление их в B или C приведет к необходимости произвести балансировку большими поворотами. Большое левое вращение выполняется симметрично большому правому.

11.3. Добавление и удаление узлов

Операция вставки нового узла в АВЛ-дерево выполняется рекурсивно. По ключу данного узла производится поиск места вставки: спускаясь по дереву вниз, алгоритм сравнивает ключ добавляемого узла со встречающимися ключами, далее происходит вставка нового элемента.

По возвращению из рекурсии, выполняется балансировка. Для осуществления балансировки следует знать, с каким из рассмотренных выше случаев дисбаланса имеется дело. Именно поэтому процесс вставки узла в АВЛ-дерево отличается от добавления узла в обычное бинарное дерево. Общий алгоритм выглядит следующим образом:

1) поиск места, где должен будет находиться новый узел;
2) добавление узла в дерево на найденную позицию;
3) пересчет коэффициентов сбалансированности для узлов, находящихся выше добавленного узла;
4) если дерево стало несбалансированным, то необходимо выполнить балансировку.

Удаление узла из АВЛ-дерева сложнее аналогичной операции для простого двоичного дерева и включает в себя следующие этапы:

1) поиск узла, который требуется удалить (в процессе поиска запоминаются пройденные узлы для выполнения последующей балансировки);
2) удаление искомого узла и обновление списка пройденных узлов;
3) выполнение балансировки и перерасчет коэффициентов сбалансированности.

Пример вставки узла в АВЛ-дерева представлен на рис. 10.4.

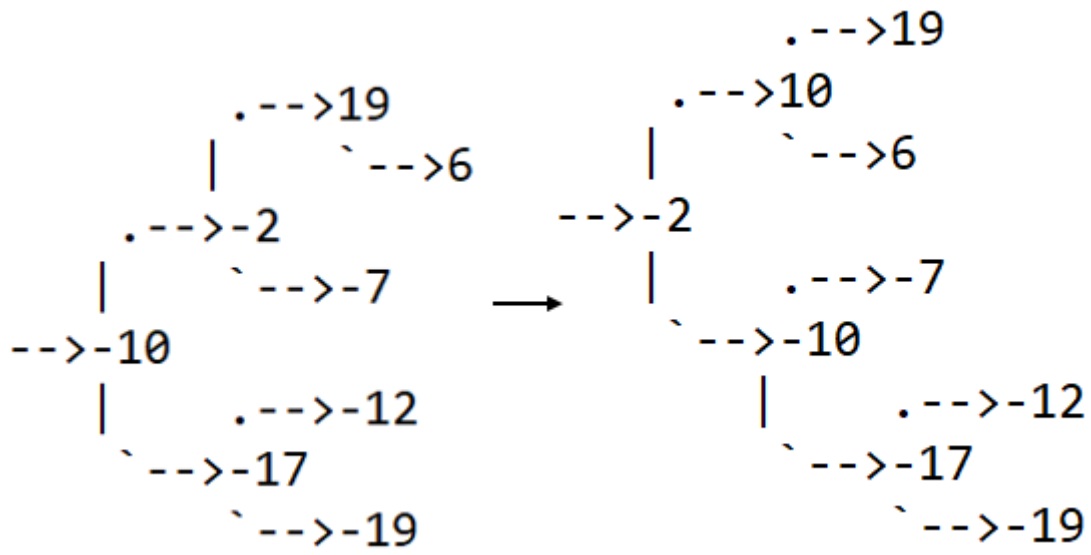


Рисунок 11.4 – Вставка ключа “10” в АВЛ-дерево

Из рисунка выше видно, насколько изменилось изначальное АВЛ-дерево.

11.4. Задание к практической работе

Необходимо реализовать программу, которая выполняет следующие действия:

1) Формирование АВЛ-дерева из N элементов:

   а) пользователь вводит количество элементов N АВЛ-дерева, которое автоматически заполняется случайными числами (-99 до 99);

   б) пользователь вводит в консоль элементы массива, N определяется автоматически по количеству введенных элементов;

   в) элементы считываются с файла, в котором хранится массив из чисел, N – количество элементов в файле.

   Определение скорости формирования АВЛ-дерева.

2) Вывод в консоль и файл AVL_tree АВЛ-дерева. АВЛ-дерево должно иметь подобный вид:

```
        .-->8
    .-->1
-->-3
    |   .-->-6
    `-->-18
        `-->-20
```

3) Определение скорости вставки, удаления и получения элемента массива.

4) Определение скорости проверки на сбалансированность.

5) Задание по варианту табл. 11.1.

При выполнении пунктов не должны нарушаться правила структур.

Должна присутствовать возможность запуска каждого пункта многократно, если есть возможность (если в дереве нет элементов, то нельзя ничего удалить и об этом нужно сообщить пользователю).

После выполнения всех заданий необходимо сделать вывод по проделанной работе.

Таблица 11.1 – Задания по вариантам

Вариант	Задание
1	Удалите все отрицательные числа. Каждый нечетный элемент умножьте на случайную величину в диапазоне от -2 до 2. Проанализируйте скорость работы АВЛ-дерева.
2	Составьте несколько вариантов проверочной работы. Варианты проверочной работы необходимо вывести в файл test, ответы к проверочной работе необходимо вывести в файл reply. Задание проверочной работы: - удаление элемента(-ов) из АВЛ-дерева; - вставка элемента(-ов) в АВЛ-дерево. Обучающий, выполняющий проверочную работу, должен поэтапно объяснить ход действий при вставке и удалении. Варианты должны быть одинаковыми по сложности.
3	Создайте еще одно бинарное дерево размерности N, заполненное случайными числами (от – 99 до 99). Соедините это бинарное дерево с исходным АВЛ-деревом.
5	Выведите количество элементов, которые нацело делятся на 1, 2, 3, 4, 5, 6, 7, 8, 9. Проанализируйте скорость работы АВЛ-дерева.

ПРАКТИЧЕСКАЯ РАБОТА №12
ПРОСТЕЙШИЙ ПРИМЕР МАШИННОГО ОБУЧЕНИЯ. КЛАСТЕРИЗАЦИЯ

Цель работы: освоение основных понятий и некоторых методов кластерного анализа. Определение преимуществ и недостатков различных методов кластеризации.

12.1. Понятие кластеризации

Кластерный анализ – задача разбиения заданной выборки объектов (ситуаций) на подмножества, называемые кластерами, так, чтобы каждый кластер состоял из схожих объектов, а объекты разных кластеров существенно отличались. Задача кластеризации относится к статистической обработке, а также к широкому классу задач обучения без учителя.

Можно встретить описание двух фундаментальных требований, которые предъявляются к данным – полнота и однородность. Однородность требует, чтобы все объекты кластеризации описывались схожим набором свойств.

Основные цели кластеризации:

1) Получение информации о данных, их понимание при помощи выявления кластерной структуры;
2) Сжатие данных. Когда выборка сильно большая, то можно её сократить, оставив от каждого кластера по одному самому типичному представителю;
3) Выделение нетипичных объектов, которые невозможно определить ни к какому их имеющиеся кластеров.

Применение кластерного анализа в общем виде сводится к следующим этапам:

1) Отбор выборки объектов для кластеризации;
2) Определение множества переменных, по которым будут оцениваться объекты в выборке. При необходимости – нормализация значений переменных;
3) Вычисление значений меры сходства между объектами;

4) Применение метода кластерного анализа для создания групп сходных объектов (кластеров);
5) Предоставление результатов анализа.

После получения и анализа результатов возможна корректировка выбранной метрики и метода кластеризации до получения оптимального результата. Далее будет рассматриваться методы кластеризации применительно к нормализированным данным (рейтинг и стоимость вина). Графическое отображение представлено на рис. 12.1.

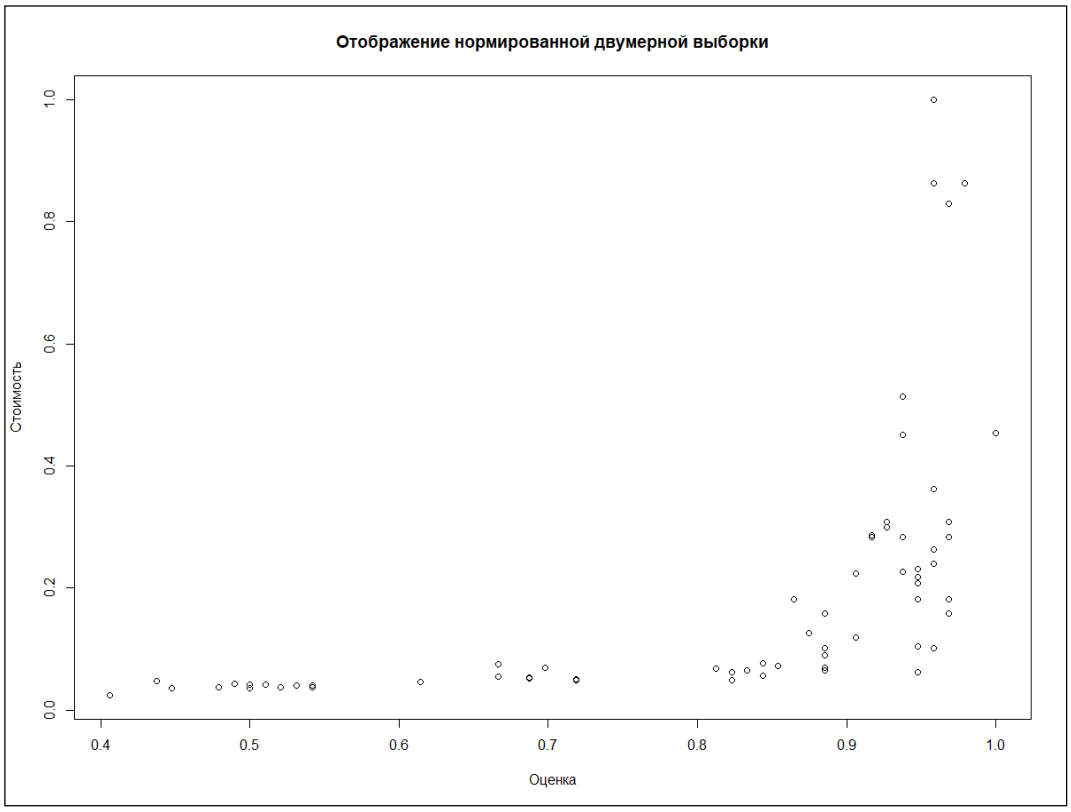


Рисунок 12.1 – Отображение кластеризуемых данных

Прежде чем приступить к кластеризации данных, необходимо нормировать выборку. Это позволит избежать искажения расстояния между объектами выборки из-за различной размерности.

12.2. Алгоритмы кластеризации k-means, k-medians, k-medoids

По своей сути алгоритмы k-means, k-medians и k-medoids очень похожи. Главное различие – каким образом определятся новый центр кластера и что им является.

Метод k-means – наиболее популярный метод кластеризации. Суть алгоритма заключается в минимизации суммарного квадратичного отклонения точек кластеров от центров этих кластеров.

В качестве критерия используется минимальное расстояние внутри кластера относительно среднего. Как только элемент включается в кластер, среднее пересматривается. Это продолжается до тех пор, пока не размещены все точки. Алгоритм завершается, когда все кластеры стабилизируются. Иными словами, когда на какой-то итерации не происходит изменения центра масс кластеров. На рис. 12.2 отображен результат кластеризации исходных данных.

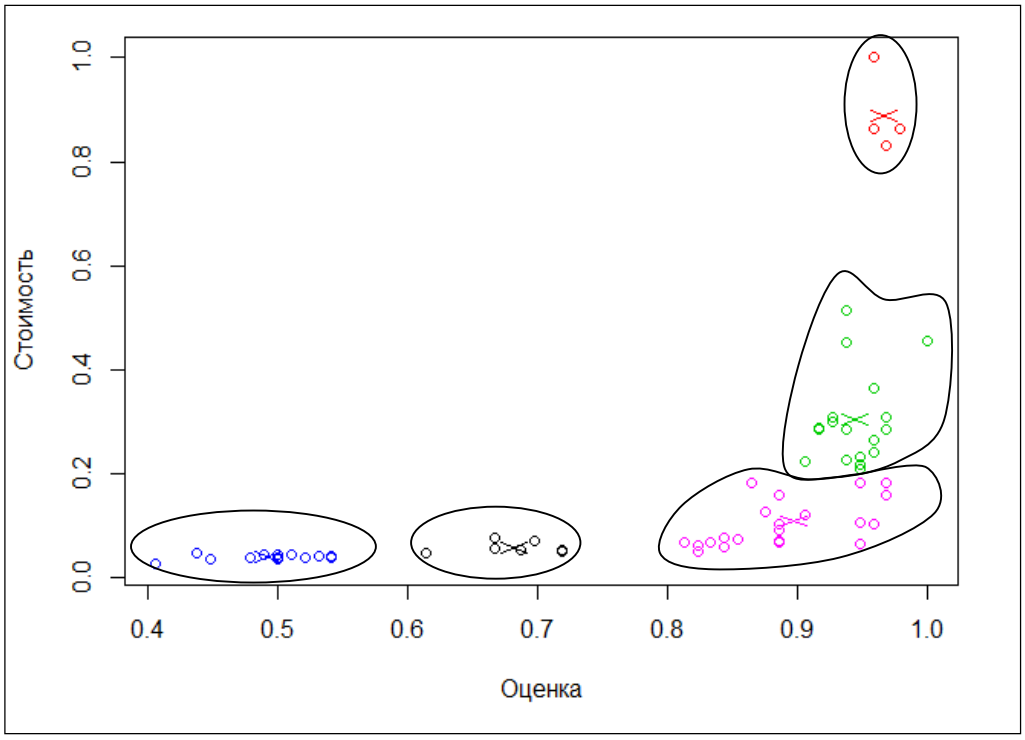


Рисунок 12.2 – Разбиение исходных данных на 5 кластеров методом k-means

Основная идея заключается в том, что на каждой итерации пересчитывается центр масс (центроид) для каждого кластера, полученного на предыдущем шаге, затем векторы разбиваются на кластеры вновь в соответствии с тем, какой из новых центров оказался ближе по выбранной метрике.

Это происходит за конечное число итераций, так как количество возможных разбиений коечного множества конечно, а на каждом шаге суммарное квадратичное отклонение уменьшается.

Данный алгоритм состоит из следующих шагов:

1) Случайно выбрать k точек, являющихся начальными центроидами кластеров;
2) Отнести каждый объект к кластеру с ближайшим центроидом;
3) Пересчитать центроиды кластеров согласно текущему членству;
4) Если критерий остановки алгоритма не удовлетворен, то необходимо перейти ко 2 шагу.

Даже на стадии теоретического рассмотрения этого алгоритма кластеризации видна главная проблема – результат зависит от выбора исходных центров кластеров. А так как они выбираются случайно, итоговое решение может быть некорректным.

Алгоритм k-medians – это вариация алгоритма k-means метода кластеризации, где для определения центра кластера вместо среднего вычисляется медиана по каждому из измерений.

Алгоритм завершается, как только все кластеры стабилизируются. Доказана сходимость метода за конечное число итераций.

Данный алгоритм состоит из следующих шагов:

1) Случайно выбрать k точек, являющихся начальными центроидами кластеров;
2) Отнести каждый объект к кластеру с ближайшим центроидом;
3) Пересчитать центроиды кластеров;
4) Если критерий остановки алгоритма не удовлетворен, то необходимо перейти ко 2 шагу.

Данный алгоритм является лишь вариацией алгоритма k-means, поэтому глобальных проблем метода k-means он не решает. Алгоритм предназначен для того, чтобы давать более качественную кластеризацию, если алгоритм k-means с этим не справился.

Алгоритм k-medoids, в отличие от k-means, использует для представления центра кластера не центр масс, а представительный объект – один из объектов кластера. Стоимость в алгоритме k-medoids определяется как сумма всех расстояний точек до центра кластера во всех кластерах.

Как и в методе k-means, сначала произвольным образом выбирается k представительных объектов. Каждый из оставшихся объектов объединяется в кластер с ближайшим медоидом (представительным объектом), используя любые распространенные методы измерения расстояния. Далее необходимо выполнять алгоритм пока стоимость выбора нового медоида снижается. Необходимо определить точку, принадлежащую кластеру, расстояние которой до всех других точек кластера минимально. Если таковых точек нет, то алгоритм завершает свою работу.

Из описания алгоритма уже видно, что он менее чувствителен к выбросам нежели k-means или k-medians. Но он полностью зависит от начального выбора медоидов. Чтобы хоть как-то улучшить качество результата необходимо определить оптимальное количество кластеров.

12.3. Определение верхней оценки количества кластеров. Методы поиска оптимального количества кластеров

Количество кластеров влияет качество кластеризации. Чрезмерное количество кластеров противоречит основным задачам кластеризации. Верхний предел количества кластеров равняется объему исследуемого признака. Но ни к чему полезному такое разбиение не приведет.

Для небольших множеств из объектов существует верхняя оценка количества кластеров, после которой увеличение количества кластеров нецелесообразно:

$$K = \left\lfloor \sqrt{\frac{N}{2}} \right\rfloor,$$

где K – количество кластеров, N – объем выборки.

Но это лишь верхняя оценка количества кластеров. Она зачастую не является оптимальной.

Поскольку кластерный анализ предназначен для создания однородных групп, естественно рассмотреть процедуры, позволяющие определить оптимальное число полученных групп.

При кластеризации методом типа k-means количество кластеров чаще всего определяют с помощью метода локтя (рис. 12.3). Он подразумевает многократное

циклическое исполнение алгоритма с увеличением количества выбираемых кластеров, а также последующим откладыванием на графике разброса кластеризации, вычисленного как функция от количества кластеров.

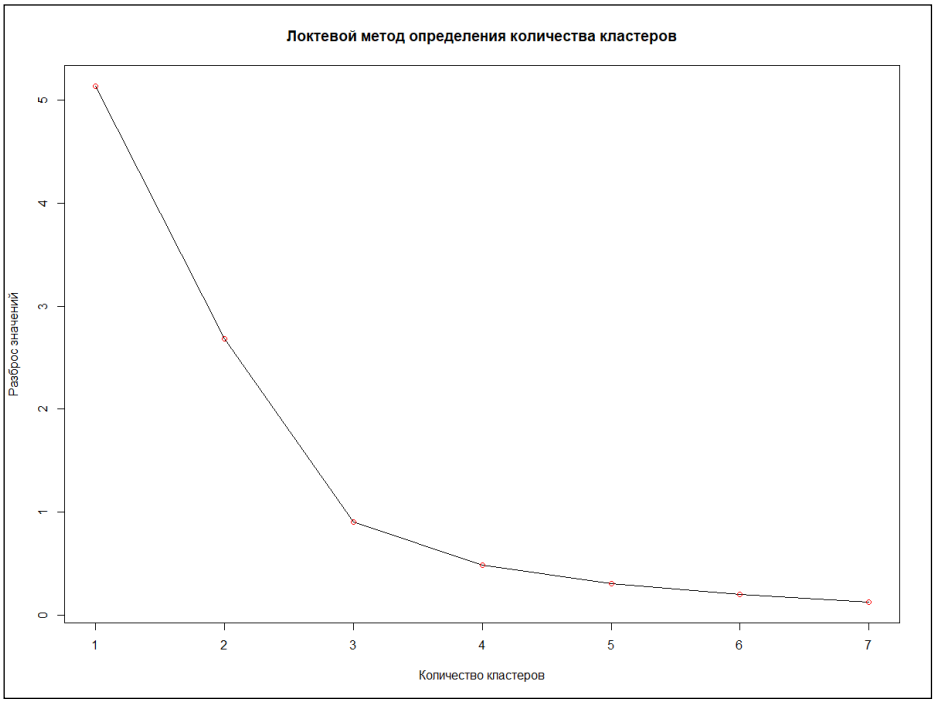


Рисунок 12.3 – Локтевой метод для k-means

После представления разброса при различных количествах кластеров на графике, необходимо определить, при каком количестве кластеров последующий разброс будет уменьшаться с наименьшей степенью. На графике это выглядит как перегиб функции, напоминающий локоть, за счет чего метод и получил свое название.

Но часто бывает ситуация, когда точно определить оптимальный кластер крайне трудно. Существует более точная метрика – силуэт.

Коэффициент силуэта вычисляется с помощью внутрикластерного расстояния (a) и среднего расстояния до ближайшего кластера (b) по каждому кластеру:

$$S_i = \frac{b_i - a_i}{\max\{a_i, b_i\}},$$

$$S = \frac{1}{K}\sum_{i=1}^{K} S_i.$$

График силуэта (рис. 12.4) имеет пиковый характер, в отличие от мягко изогнутого графика при использовании метода локтя. Его проще визуализировать и обосновать.

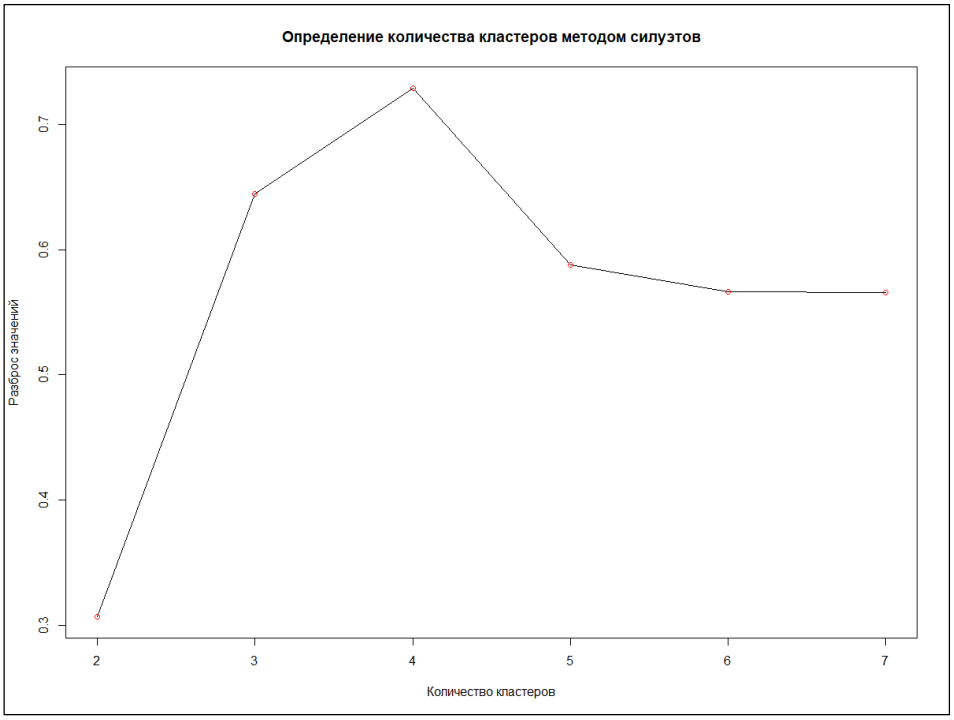


Рисунок 12.4 – Метод силуэтов для k-means

Эти методы решают проблему с выбором оптимального количества кластеров, но зависимость от выборка начальных центров кластеров по-прежнему остается. Это делает алгоритмы типа k-means нестабильными.

12.4. Задание к практической работе

Необходимо написать программу, которая может сгенерировать двумерную выборку данных или обработать имеющеюся (ввод с файла) и которая:

1) Определяет верхнюю оценку количества кластеров.
2) Методом k-means кластеризует исходные данные. Кластеризованные данные необходимо отобразить и графически, и списком (массивом). Каждый кластер должен иметь свой цвет, а центроид должен быть помечен.
3) Методом k-medians кластеризует исходные данные. Кластеризованные данные необходимо отобразить и графически, и списком (массивом). Каждый кластер должен иметь свой цвет, а центроид должен быть помечен.
4) Методом k-medoids кластеризует исходные данные. Кластеризованные данные необходимо отобразить и графически, и списком (массивом). Каждый кластер должен иметь свой цвет, а центроид должен быть помечен.

5) С помощью метода локтя и метода силуэтов определяет для каждого метода оптимальное число кластеров.

После выполнения всех задний сравните методы между собой и сделайте выводы по проделанной работе.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Павловская Т. А. C/C++. Программирование на языке высокого уровня. СПб: Питер, 2007. – 460 с.
2. Стивен Прата. Язык программирования C++: лекции и упражнение, 6-е издание. М.: Вильямс, 2012. 1240 с.
3. Алгоритмы: построение и анализ, 3-е издание. Т. Кормен, Ч. Лейзерсон, Р. Ривест. К. Штайн. Пер. с англ. – М.: ООО «И.Д. Вильямс», 2013. – 1328 с.
4. Адитья Бхаргава. Грокаем алгоритмы: иллюстрированное пособие для программистов и любопытствующих. Спб: Питер, 2017. – 288 с.
5. Классификация и кластер. Под ред. Дж. Вэн Райзина. М.: Мир, 1980. 390 с.