С.В. Самуйлов, С.В. Самуйлова

ДИСКРЕТНАЯ МАТЕМАТИКА. МНОЖЕСТВО СРЕДСТВАМИ ЯЗЫКА ПРОГРАММИРОВАНИЯ C++

Учебное пособие

Москва
Ай Пи Ар Медиа
2025

УДК 004.05
ББК 22.1
С17

Авторы:
Самуйлов С.В. — канд. техн. наук, доц. кафедры математического обеспечения и применения ЭВМ Пензенского государственного университета;
Самуйлова С.В. — канд. техн. наук, доц. кафедры высшей и прикладной математики Пензенского государственного университета

Рецензенты:
Бычков А.С. — канд. техн. наук, доц. кафедры вычислительной техники Пензенского государственного университета;
Афонин А.Ю. — канд. техн. наук, начальник управления системного программного обеспечения НТП «Криптософт»

Самуйлов, Сергей Владимирович.
С17 Дискретная математика. Множество средствами языка программирования С++ : учебное пособие / С.В. Самуйлов, С.В. Самуйлова. — Москва : Ай Пи Ар Медиа, 2025. — 84 с. — Текст : электронный.

ISBN 978-5-4497-4554-5

В учебном пособии рассмотрены основные подходы к реализации абстрактной структуры данных «Множество» средствами языка программирования С++. Приводится теоретический материал по связным спискам, объектно-ориентированному программированию, основным контейнерным классам языка С++. Задания на лабораторные работы построены таким образом, чтобы студентами было выполнено сравнение реализации основных операций над множеством с использованием описанных в теории подходов. Приведены основные сведения по работе с бинарными деревьями. Большое внимание в работе уделено рассмотрению материалов к курсовой работе, даны структура курсовой работы, примерные темы курсовой.

Подготовлено в соответствии с Федеральным государственным образовательным стандартом высшего образования.

Предназначено для студентов всех специальностей и направлений подготовки, изучающих дисциплину «Дискретная математика».

Учебное электронное издание

ISBN 978-5-4497-4554-5

© Самуйлов С.В., Самуйлова С.В., 2025
© ООО Компания «Ай Пи Ар Медиа», 2025

Учебное издание

Самуйлов Сергей Владимирович
Самуйлова Светлана Валентиновна

Редактор *М.В. Леман*
Технический редактор, компьютерная верстка *А.Д. Талмаева*
Корректор *Ю.С. Королева*
Обложка *Я.А. Кирсанов, С.С. Сизиумова*, фотобанк Freepik

Подписано к использованию 10.07.2025. Объем данных 7 Мб.

ООО Компания «Ай Пи Ар Медиа»
8 800 555 22 35 (бесплатный звонок по России)
E-mail: sales@iprmedia.ru

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **6**

1. ОДНОСВЯЗНЫЕ СПИСКИ **8**

1.1. Теоретическая часть 8

1.2. Задание на лабораторную работу № 1. Разработка функций, реализующих операции создания и работы с множествами 17

2. ЭЛЕМЕНТЫ ТЕОРИИ МНОЖЕСТВ **19**

2.1. Теоретическая часть 19

2.2. Задание на лабораторную работу № 2. Создание библиотеки функций, реализующих операции над множествами 29

3. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ **33**

3.1. Теоретическая часть 33

3.2. Задание на лабораторную работу № 3. Разработка класса «Множество» на основе односвязного списка 38

4. КОНТЕЙНЕРНЫЙ КЛАСС LIST **40**

4.1. Теоретическая часть 40

4.2. Задание на лабораторную работу № 4. Разработка класса «Множество» на основе контейнера list 47

5. КОНТЕЙНЕРНЫЙ КЛАСС SET **48**

5.1. Теоретическая часть 48

5.2. Задание на лабораторную работу № 5. Разработка класса «Множество» на основе контейнера set 50

6. ОБЗОР КОНТЕЙНЕРНЫХ КЛАССОВ 51
6.1. Контейнерный класс vector 51
6.2. Контейнерный класс stack 53
6.3. Контейнерный класс queue 55
6.4. Контейнерный класс deque 57
6.5. Контейнерный класс priority_queue 59
6.6. Контейнерный класс map 62
6.7. Контейнерный класс bitset 64
6.8. Задание на лабораторную работу № 6.
Сравнение подходов к реализации структуры данных «Множество» 67

7. ДЕРЕВЬЯ И ИХ СВОЙСТВА 68
7.1. Теоретическая часть 68
7.2. Задание на лабораторную работу № 7.
Разработка класса «Бинарное дерево поиска» 74

8. КУРСОВАЯ РАБОТА 75
8.1. Примерные темы курсовой работы 75
8.2. Документы к курсовой работе 77
8.3. Разделы курсовой работы 78

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 84

ВВЕДЕНИЕ

Дискретная математика — это раздел математики, который изучает дискретные математические структуры, такие как множества, графы, деревья и другие объекты, которые могут быть представлены в виде дискретных данных.

Основные понятия, которые изучаются в рамках этого раздела математика, это теория множеств, теория высказываний, теория графов, комбинаторика и т.п.

Для студентов-программистов важным является прикладной аспект перечисленных выше понятий, т.е. каким образом изучаемая в дисциплине теория может быть реализована средствами языков программирования.

В учебном пособии рассматриваются вопросы реализации структур данных «Множество» и «Бинарное дерево» средствами языка программирования C++.

Учебное пособие состоит из восьми разделов. В *первом разделе* дан теоретический материал и приведено практическое задание для реализации структуры данных «Множество» с помощью односвязного списка.

Второй раздел дает теоретическую информацию по теории множеств и содержит практическое задание на реализацию основных операций над множествами.

Третий раздел содержит описание основных положений объектно-ориентированного подхода к программированию, а также задание на реализацию операций над множествами методами класса «Односвязный список».

Четвертый раздел раскрывает понятие контейнерного класса, описывает один из самых востребованных контейнерных классов list, включая рассмотрение его основных методов, а также содержит практическое задание на реализацию основных операций над множествами методами контейнерного класса list.

Пятый раздел описывает один из самых оптимальных для работы с множествами контейнерный классов set и его основные методы, а также содержит практическое задание на реализацию основных операций над множествами методами контейнерного класса set.

Шестой раздел описывает наиболее часто используемые контейнерные классы, такие как контейнерный класс vector, контейнерный класс stack, контейнерный класс queue и т.д., их назначение и основные методы. Также в раз-

деле приведено практическое задание на разработку программы для сравнения различных подходов к реализации структуры данных «Множество».

В *седьмом разделе* приведена теории по бинарным деревьям, дано задание на реализацию операций обхода бинарного дерева с помощью контейнерных классов языка C++.

Восьмой раздел учебного пособия посвящен вопросам написания курсовой работы по дисциплине. Приведены примерные темы курсовой работы, а также необходимые для написания работы документы.

В результате освоения материала обучающиеся должны ***знать*** основные положения теории множеств, ***уметь*** реализовать основные операции над множествами и деревьями средствами языка программирования C++, ***получить навыки*** использования контейнерных классов языка C++ для решения прикладных задач.

1. ОДНОСВЯЗНЫЕ СПИСКИ

Цель лабораторной работы — разработка функций, реализующих операции создания и работы с множествами с использованием односвязных списков.

1.1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1.1. Основные определения

Связный список — такая структура, элементами которой служат записи с одним и тем же форматом, связанные друг с другом с помощью указателей, хранящихся в самих элементах.

Простейшей из обсуждаемых структур является однонаправленный список. Он строится подобно очереди на прием к врачу: пациенты сидят на любых свободных местах, но каждый из них знает, за кем он в очереди (т. е. данные размещаются на свободных местах в памяти, но каждый элемент содержит ссылку на предыдущий или следующий элемент). Поскольку количество пациентов заранее не очевидно, структура является динамической.

Списки являются удобным способом хранения динамических множеств, позволяющим реализовать все операции, хотя и не всегда эффективно.

В односвязном списке каждые элемент или узел списка состоят из двух различных по назначению полей: **поля информации (или поля данных)** и **поля указателя** (рис. 1) [1].

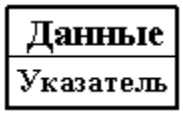


Рис. 1. Структура узла списка

В **поле информации** хранятся данные, ради которых и создается список. Это поле может представлять в общем случае запись, состоящую из нескольких полей.

Поле указателя хранит адрес следующего элемента списка. С помощью указателя можно получить доступ к следующему элементу списка, из следующего — к очередному и т. д., пока не будет достигнут последний элемент списка.

Поле указателя последнего элемента списка должно содержать признак нулевого, или пустого, указателя, свидетельствующего о конце списка.

Прежде чем двигаться по указателям, надо знать хотя бы один элемент списка. Поэтому очень важной частью логической структуры односвязного списка является **указатель на начало списка (указатель на голову списка)**. Логическая структура односвязного списка приведена на рис. 2.

Рис. 2. Логическая структура односвязного списка

Односвязный список можно рассматривать как аналог массива, но со следующими особенностями:

– список удобно составлять из переменных типа «запись», поскольку нужны и доступ к данным элемента, и работа с ним как с единым целым;

– список может иметь произвольный размер;

– доступ к элементу списка уже не прямой, как в массиве, а последовательный (надо поставить указатель в начало списка и, двигаясь по цепочке, отсчитать нужное количество элементов), что, разумеется, замедляет работу с ним;

– вставка элемента в список, напротив, осуществляется быстрее, не надо перемещать в памяти оставшийся «хвост» массива; достаточно лишь отправить указатель очередного элемента на новый, а его указатель — на следующий элемент списка, так же как и удаление (переместить один указатель в списке — и все);

– в отличие от массива, список может состоять из разнотипных элементов, лишь бы у каждого из них были поле указателя на следующий элемент и какое-нибудь поле, задающее размер элемента, чтобы можно было единообразно удалять элементы списка.

Физическая структура односвязного списка представляет собой совокупность одинаковых по размеру и формату записей, размещенных произвольно в некоторой области памяти и связанных друг с другом в линейно упорядоченную цепочку с помощью указателей.

Таким образом, физическая смежность элементов связного списка в памяти не требуется, причем между элементами одного списка в памяти могут находиться элементы другого списка.

1.1.2. Операции над односвязным списком

Выделим **типовые операции над списками** [1]:

– добавление элемента в начало списка;

– удаление элемента из начала списка;

– добавление элемента в произвольное место списка, отличное от начала (например, после элемента, указатель на который задан);

– удаление элемента из произвольного места списка, отличного от начала (например, после элемента, указатель на который задан);

– проверка, пуст ли список;

– перестановка элементов списка;

– слияние двух списков;

– очистка списка;

– печать списка.

Прежде чем рассматривать операции над связными списками, введем обозначения переменных, которые потребуются для описания соответствующих алгоритмов:

Data **—** поле данных в элементе списка;

Next **—** поле указателя в элементе списка;

Start **—** указатель на начало списка;

Work **—** рабочий указатель;

ВП(Work) **—** выделить память под элемент списка;

ОП(Work) **—** освободить память.

Если ***Work*** — указатель на элемент списка, то:

Work–>Data — значение информационного поля ***Data***,

Work–>Next — значение адреса (указателя), который хранится в поле ***Next***.

Включение элемента в голову списка

Алгоритм включения элемента ***X*** в начало списка может быть следующим (рис. 3).

Процедура Включить_В_Начало (X)

1. ВП(Work).
2. Work–>Data = X (сформировать поле информации).
3. Work–>Next = Start (сформировать поле указателя).
4. Start = Work.

Легко убедиться в том, что алгоритм работает правильно даже в том случае, если список изначально пуст, т. е. ***Start = 0***.

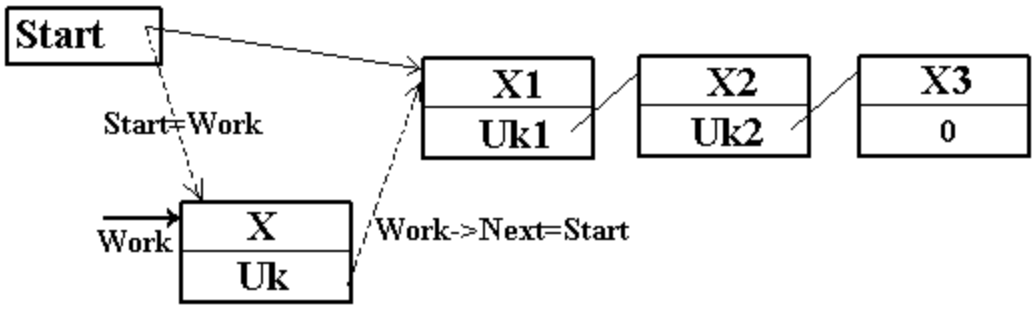


Рис. 3. Включение элемента в голову списка

Формирование списка путем добавления нового элемента в начало

Операция включения элемента в голову списка является основой для алгоритма формирования списка. Формируется пустой список, и последовательно добавляются элементы в его начало.

Запишем алгоритм формирования списка из n элементов путем добавления элементов в его начало.

Процедура Создание_Списка

1. Start = 0 (сформировать пустой список).
2. ВП(Work).
3. Work–>Data = n (сформировать поле информации).
4. Work–>Next = Start (сформировать поле указателя).
5. Start = Work.
6. n = n – 1.
7. Если n = 0, закончить. Иначе на шаг 2.

Это самый простой алгоритм формирования списка, однако порядок элементов в этом случае обратен порядку их включения, что не всегда желательно.

Включение элемента в список после заданного

Необходимо включить элемент, заданный ссылкой ***Work1***, после элемента списка, заданного ссылкой ***Work2***.

Алгоритм состоит из следующих шагов (рис. 4).

Процедура Включить_В_Список (X, Work2)

1. ВП(Work1).
2. Work1–>Data = X.
3. Work1–>Next = Work2–>Next.
4. Work2–>Next = Work1.

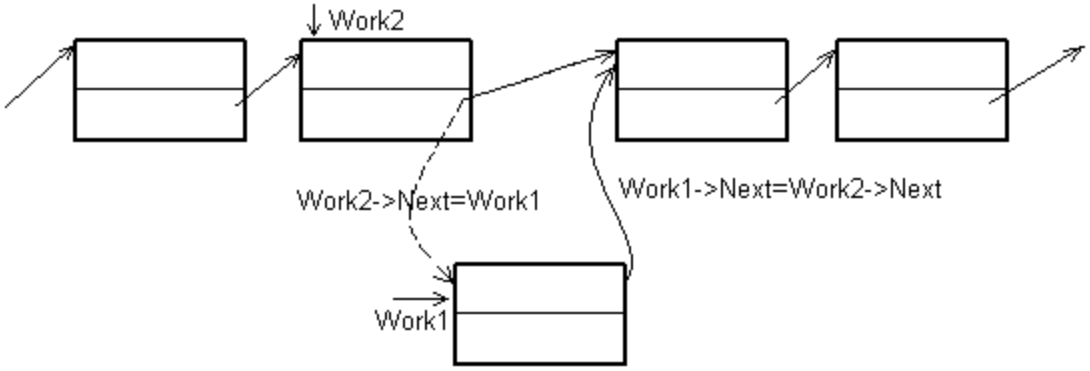


Рис. 4. Включение элемента в список после заданного

Включение элемента в конец списка

Зная алгоритм включения элемента после заданного, поставленную задачу можно решить следующим образом. Просмотрев весь список, найти последний его элемент и вставить новый элемент за последним.

Однако такой метод допустим только для списков с небольшим количеством элементов. Если же количество элементов в списке велико, то такой алгоритм по очевидным причинам неэффективен. Лучшее решение — ввод второй ссылки, указывающей на последний элемент списка.

Недостаток этого метода создания списка в том, что первый из включаемых элементов нужно обрабатывать иначе, чем остальные.

Включение элемента в список перед заданным элементом

Пусть требуется включить элемент ***X***, заданный ссылкой ***Work1***, перед элементом списка, заданным ссылкой ***Work2***. Алгоритм решения этой задачи неочевиден, хотя и прост.

Трудность заключается в том, что в односвязном списке нет возможности вернуться к элементу, предшествующему данному. Однако, как вы увидите из алгоритма, это и не требуется. Достаточно включить новый элемент после текущего, а потом поменять местами значения информационных полей (рис. 5).

1. Work1–>Data = Work2–>Data.
2. Work1–>Next = Work2–>Next.
3. Work2–>Next = Work1.
4. Work2–>Data = X.

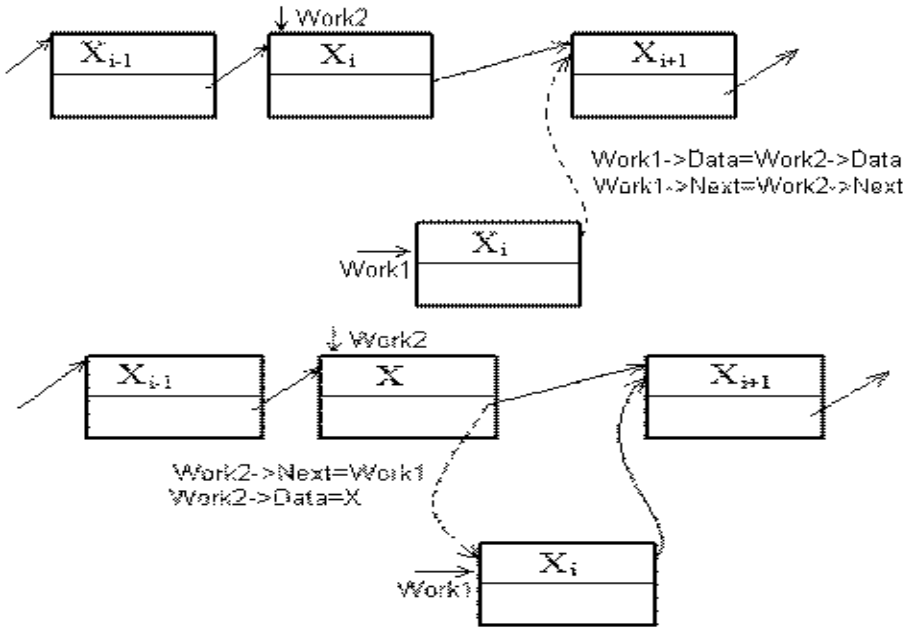


Рис. 5. Включение элемента в список перед заданным

Удаление элемента из списка

Удаление элемента, следующего за заданным

Сначала рассмотрим алгоритм исключения элемента, следующего за элементом, на который указывает *Work1* (рис. 6).

Процедура Исключить (Work1)

1. Work2 = Work1–>Next (получить адрес исключаемого элемента).
2. Work1–>Next = Work2–>Next.
3. ОП(Work2).

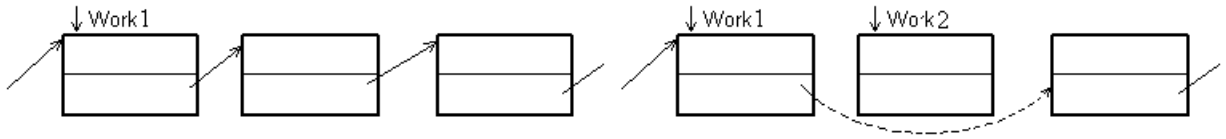


Рис. 6. Удаление элемента, следующего за заданным

Удаление текущего элемента

Труднее исключить элемент, **на который указывает указатель**. Та же проблема, что и при включении. Однако и решение этой проблемы аналогично. Исключается следующий элемент, но перед этим значение его информационного поля передвигается вперед (рис. 7).

1. Work2 = Work1–>Next.
2. Work1–>Data = Work2–>Data.
3. Work1–>Next = Work2–>Next.
4. ОП(Work2).

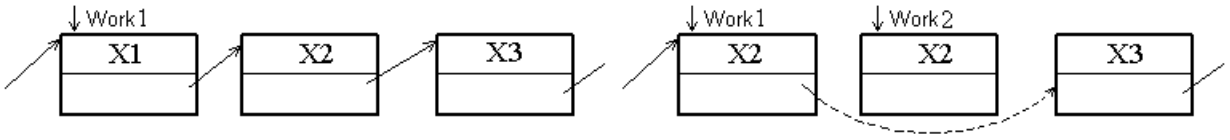


Рис. 7. Удаление текущего элемента

Следует иметь в виду, что рассмотренные выше алгоритмы можно применять только в случае, если за удаляемым элементом есть последующий элемент, т. е. он не является последним в списке.

Просмотр списка

Просмотр списка выполняется с целью обработки его элементов. При последовательном просмотре списка обработка может состоять в:

– печати содержимого информационной части или отдельных ее полей;

– модификации полей информационной части;

– сравнении полей информационной части с образцом при поиске по ключу;

– подсчете количества элементов в списке;

– очистке списка и т. п.

Рассмотрим операцию печати элементов списка.

Процедура Печать_Списка (Start)

1. Work1 = Start.
2. Пока Work1 <> 0.
3. Печать Work1–>Data.
4. Work1 = Work1–>Next.
5. Конец Пока.

В **алгоритме поиска** некоторого значения в списке на втором шаге добавляется еще одна проверка информационного поля на равенство искомому значению.

Проверка списка на наличие элементов

Проверка списка на наличие элементов заключается в анализе хранящегося в указателе на голову списка значения.

Функция Пусто (Start)

1. Пусто = Ложь.
2. Если Start = 0, то Пусто = Истина.

Перестановка элементов списка

Изменчивость динамических структур данных предполагает не только изменения размера структуры, но и изменения связей между элементами. Для связных структур изменение связей не требует пересылки данных в памяти, а только изменения указателей в элементах связной структуры.

В качестве примера рассмотрим алгоритм перестановки двух соседних элементов списка. Пусть задан указатель ***Work1*** — адрес элемента, предшествующего паре, в которой производится перестановка (рис. 8).

Процедура Обмен (Work1)

1. Work2 = Work1–>Next { указатель на 1-й элемент пары}
2. Work3 = Work2–>Next { указатель на 2-й элемент пары}
3. Work2–>Next = Work3–>Next; {1-й элемент пары теперь указывает на следующий за парой}
4. Work3–>Next = Work2 { 1-й элемент пары теперь следует за 2-м}
5. Work1–>Next = Work3 { 2-й эл-т пары теперь становится 1-ым}

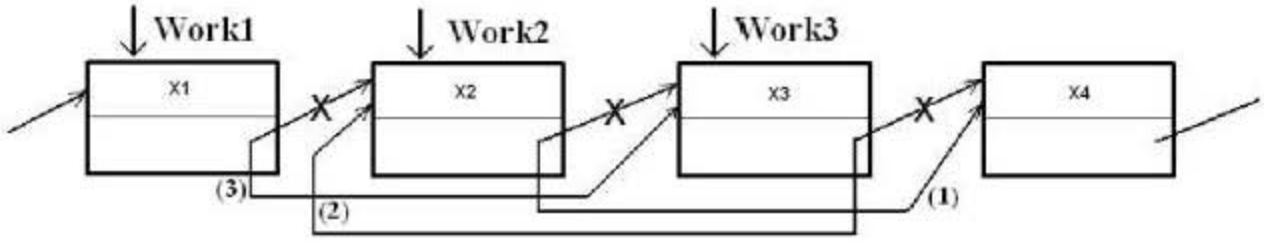


Рис. 8. Перестановка элементов списка

В приведенном алгоритме не учитывается случай перестановки первого и второго элементов.

Слияние двух списков

Операция слияния заключается в формировании из двух списков одного. В случае односвязного списка слияние выполняется очень просто. Последний элемент первого списка содержит пустой указатель на следующий элемент, этот указатель служит признаком конца списка. Вместо этого пустого указателя в последний элемент первого списка заносится указатель на начало второго списка. Таким образом, второй список становится продолжением первого.

Однако алгоритм решения этой задачи содержит некоторые особенности.

Пусть заданы два указателя: ***Work1*** и ***Work2*** на голову первого и второго списков соответственно. Необходимо добавить элементы второго списка в хвост первого.

Процедура Слияние (Var Work1, Work2)

1. Если Work2 = 0, то второй список пуст. Выход.
2. Если Work1 = 0, то Work1 = Work2. Выход.
3. Work3 = Work1.
4. Пока Work3–>Next <> 0, выполнять Work3 = Work3–>Next.
5. Work3–>Next = Work2.
6. Work2 = 0.

Во всех рассмотренных выше операциях чрезвычайно важна последовательность коррекции указателей, которая обеспечивает корректное изменение списка, не затрагивающее другие элементы. При неправильном порядке коррекции легко потерять часть списка.

1.2. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ № 1
РАЗРАБОТКА ФУНКЦИЙ, РЕАЛИЗУЮЩИХ ОПЕРАЦИИ СОЗДАНИЯ И РАБОТЫ С МНОЖЕСТВАМИ

ВНИМАНИЕ!!! ОГРАНИЧЕНИЯ. ООП не использовать. Для представления множества использовать односвязный список целочисленных значений. Структура программы должна включать:

– заголовочный файл;

– файл реализации — SetLab1_2_<**Фамилия**>.cpp;

– главную программу Lab1_2_<**Фамилия**>.cpp.

Лабораторная работа выполняется в консольном приложении на C++.

Программа должна включать следующие функции:

1. **F1. Создание пустого множества**. Входные параметры: нет. Выходные параметры: указатель на первый элемент списка, равный **NULL**.

2. **F2. Пустое множество.** Входные параметры: указатель на первый элемент списка. Выходные параметры: логическое значение.

3. **F3. Проверка принадлежности элемента множеству**. Входные параметры: указатель на первый элемент списка, значение элемента. Выходные параметры: логическое значение. Использовать **F2**.

4. **F4. Добавление нового элемента в множество**. Входные параметры: указатель на первый элемент списка, добавляемое в список значение. Выходные параметры: указатель на первый элемент списка-результата. Использовать **F3. Ограничение**: добавление осуществлять в начало списка.

5. **F5. Создание множества.** Входные параметры: количество элементов, интервал допустимых значений (от min до max). Выходные параметры: указатель на первый элемент списка-результата. Генерация значений — датчиком случайных чисел. Использовать **F4. Требования:** проверить возможность создания множества с заданными параметрами.

6. **F6. Мощность множества.** Входные параметры: указатель на первый элемент списка. Выходные параметры: целочисленное значение. Использовать **F2**.

7. **F7. Вывод элементов множества.** Входные параметры: указатель на первый элемент списка, символ разделителя. Выходные параметры: строка, содержащая элементы множества, разделенные символом разделителя. Использовать **F2**. **Требования:** в конце строки-результата разделитель стоять не должен.

8. **F8. Удаление множества (очистка памяти, занимаемой списком).** Входные параметры: указатель на первый элемент списка. Выходные параметры: указатель на первый элемент списка, равный **NULL**.

В главной программе Lab1_2_<**Фамилия**>.cpp протестировать функции F5, F6, F7 и F8. Мощность множества — от 6 до 9 элементов (разная от запуска к запуску, задается датчиком случайных чисел).

Примерный вариант результата работы программы приведен на рис. 9.

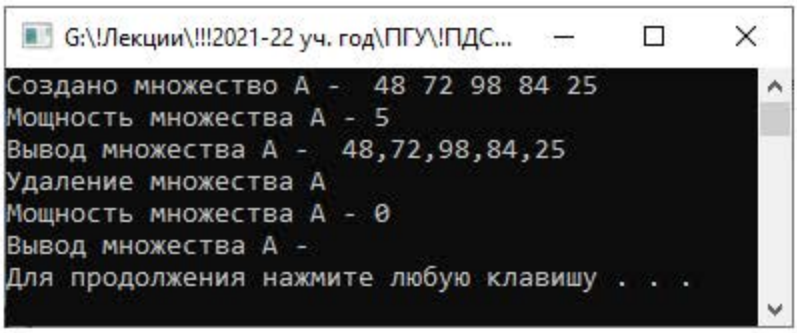


Рис. 9. Примерный вариант результатов работы ЛР № 1

Весь вывод осуществлять **только** в главной программе.

2. ЭЛЕМЕНТЫ ТЕОРИИ МНОЖЕСТВ

Цель лабораторной работы — разработка функций, реализующих базовые операции над множествами.

2.1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

2.1.1. Понятие множества

При построении доступной для рационального анализа картины мира часто используется термин «**объект**» для обозначения некой сущности, отделимой от остальных.

Выделение объектов — это не более чем произвольный акт нашего сознания. В одной и той же ситуации объекты могут быть выделены по-разному, в зависимости от точки зрения, целей анализа и других обстоятельств. Но, как бы то ни было, выделение объектов и их совокупностей — естественный (или даже единственно возможный) способ организации нашего мышления, поэтому неудивительно, что он лежит в основе главного инструмента описания точного знания — математики.

Понятие множества принадлежит к числу фундаментальных понятий математики.

Можно сказать, что **множество** — это любая определённая совокупность объектов.

Георг Кантор (немецкий математик, основатель современной теории множеств) писал, что **под «множеством я понимаю** вообще все то многое, которое возможно мыслить как единое, т. е. такую совокупность определенных элементов, которая посредством одного закона может быть соединена в одно целое» [2].

Множество — это совокупность объектов, объединенных по какому-либо признаку.

Множество — неупорядоченная именованная совокупность элементов, удовлетворяющая следующим условиям:

– каждый элемент совокупности уникален, т. е. отличим от других;

– для любого объекта существует возможность установить, принадлежит ли он множеству или нет [3].

Объекты, из которых составлено множество, называются его **элементами. Элементы множества различны и отличимы друг от друга.** Как множествам, так и элементам можно давать имена или присваивать символьные обозначения.

Элементы множества в математике принято заключать в **фигурные скобки**.

Традиционно множества будем **обозначать** заглавными буквами: *A, B, C, ...*, а элементы маленькими буквами: *a, b, c, ...*

Тот факт, что элемент ***x* принадлежит множеству *A*,** записывается так: $x \in A$, в противном случае пишем $x \notin A$.

Множество, которое содержит конечное количество элементов, именуют **конечным множеством** (множество студентов ФВТ, множество дней недели). Если множество содержит бесконечное количество элементов, его называют **бесконечным** (множество целых чисел).

Для однозначного описания некоторого множества мы будем пользоваться следующими способами [4]:

а) задавать список элементов, входящих в данное множество. Если множество ***A*** состоит из элементов $a_1, \ldots, a_n$, то будем писать $A = \{a_1, \ldots, a_n\}$. Например, множество ***M*** арабских цифр можно задать **перечислением** $M = \{0, 1, 2, \ldots, 9\}$. Порядок, в котором записываются элементы множества, значения не имеет;

б) мы не можем выписать все элементы очень больших, в особенности бесконечных, множеств. В этом случае множества определяются с помощью подходящих **предикатов** (*предикат — это* то, что утверждается о субъекте), задающих общее свойство элементов, принадлежащих множеству ***A***. Будем писать $A = \{x \mid P(x)\}$, что означает «множество всех ***x***, таких, что выполнено $P(x)$». Здесь $P(x)$ означает свойство, характеризующее в точности все элементы множества ***A***. Например, множество ***M*** арабских цифр можно задать посредством свойства $A = \{x \mid x$ — арабская цифра$\}$;

в) указывать порождающую процедуру, т. е. способ получения элементов множества ***A***. Например, множество чисел Фибоначчи первого порядка можно задать с помощью следующей порождающей процедуры: $F = \{f_i \mid f_1 = 0, f_2 = 1, f_i = f_{i-1} + f_{i-2}\}$;

г) четвертый способ — задание множеств с помощью операций над уже известными множествами (будут рассмотрены ниже).

Перечислением элементов можно задавать только конечные множества. Бесконечные множества задаются характеристическим предикатом или порождающей процедурой.

Заметим, что элементы множества сами могут являться множествами. Например, множество групп студентов состоит из элементов (групп), которые, в свою очередь, состоят из студентов.

Принято использовать следующие **обозначения для числовых множеств** [5]:

– N или ω — множество натуральных чисел,

– Z — множество целых чисел,

– Q — множество рациональных чисел,

– R — множество вещественных чисел,

– C — множество комплексных чисел.

Число элементов множества A обозначается как $|A|$ и называется **мощностью** (размером, нормой, длиной и др.) множества A.

Два множества, имеющие одинаковую мощность, называются **равномощными**. Например: A — множество цветов радуги, B — множество нот $|A| = |B| = 7$.

Множество, которое не содержит ни одного элемента, называется **пустым множеством** и обозначается как $\varnothing$. Так, например, может оказаться, что множество студентов, получивших оценку «отлично» на экзамене по ПДСД, пусто.

Универсальное множество (обозначается U) есть множество всех рассматриваемых в данной задаче элементов.

Обозначения стандартных множеств

При работе с множествами часто **используются следующие обозначения** [6]:

– **символ** $\Rightarrow$ — импликация, следование. $A \Rightarrow B$ означает «если A верно, то B также верно». Пример: $x=2 \Rightarrow x^2=4$;

– **символ** $\Leftrightarrow$ — «если и только если», или «равносильно». $A \Leftrightarrow B$ означает «***A*** верно тогда и только тогда, когда ***B*** верно». Пример: $x+5 = y+2 \Leftrightarrow x+3 = y$;

– **символ** {|} вместо слов «таких, что»; $\{x \mid P(x)\}$ означает множество всех x таких, что верно $P(x)$, т. е. множество элементов, удовлетворяющих условию;

– **символы &, ∧** для обозначения связки И;

– **символы |, ∨** для обозначения связки ИЛИ;

– **квантор общности** $\forall x$ (для всех x). $\forall x, P(x)$ обозначает «$P(x)$ верно для всех x»;

– **квантор существования** $\exists x$ (существует x). $\exists x, P(x)$ означает «существует хотя бы один x такой, что верно $P(x)$».

Мультимножества

В множестве все элементы различны, а значит, входят в множество ровно один раз. В некоторых случаях оказывается полезным рассматривать совокупности элементов, в которые элементы входят по несколько раз.

Пусть $X = \{x_1, \ldots, x_n\}$ — некоторое (конечное) множество, и пусть $a_1, a_2, \ldots, a_n$ — неотрицательные целые числа. Тогда мультимножеством $\hat{X}$ (над множеством ***X***) называется совокупность элементов множества ***X***, в которую элемент x_i входит a_i раз, $a_i \geq 0$. Мультимножество обозначается одним из следующих способов:

$$\hat{X} = \left[x_1^{a_1}, \ldots, x_n^{a_n}\right] = \langle \underbrace{x_1, \ldots, x_1}_{a_1}; \ldots; \underbrace{x_n, \ldots, x_n}_{a_n} \rangle = \langle a_1(x_1), \ldots, a_n(x_n) \rangle$$

Пример 1

Пусть $X = \{a, b, c\}$. Тогда

$$\hat{X} = [a^0, b^3, c^4] = \langle b, b, b; c, c, c, c \rangle = \langle 0(a), 3(b), 4(c) \rangle.$$

Число a_i называется **показателем** элемента x_i, множество ***X*** — **носителем** мультимножества $\hat{X}$, число $m = a_1 + \ldots + a_n$ — **мощностью** мультимножества $\hat{X}$, а множество $\underline{\hat{X}} = \{x_i \in X \mid a_i > 0\}$ называется **составом** мультимножества $\hat{X}$.

Пример 2

Пусть $\hat{X} = [a^0, b^3, c^4]$ — мультимножество над множеством $X = \{a, b, c\}$. Тогда $\underline{\hat{X}} = \{b, c\}$.

2.1.2. Операции над множествами. Диаграммы Эйлера — Венна

Самого по себе понятия множества ещё недостаточно: нужно определить способы конструирования новых множеств из уже имеющихся, то есть определить операции над множествами [2].

Операции над множествами будем иллюстрировать с помощью диаграмм Эйлера — Венна.

Диаграммы Эйлера — Венна — это геометрическое представление множеств. Множество U изображается прямоугольником, рассматриваемые множества — окружностями. Для выделения результата выполнения операций над множествами применяются штриховка или закраска цветом [5].

Подмножество

Множество A называют подмножеством множества B (обозначается $A \subseteq B$), если всякий элемент множества A является элементом множества B (рис. 10):

$$A \subseteq B \Leftrightarrow \forall x(x \in A \Rightarrow x \in B).$$

Говорят: A содержится в B, или B включает A.

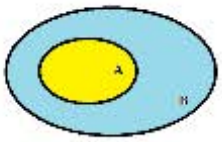


Рис. 10. Диаграмма «Подмножество»

Если $A \subseteq B$ и $A \neq B$, то **A называется строгим (собственным) подмножеством B**. Обозначается: $A \subset B$.

Подмножества A и $\varnothing$ множества A называются **несобственными подмножествами** множества A. Любое другое подмножество называется **собственным подмножеством** этого множества.

Множество всех подмножеств некоторого множества A называется его **булеаном**.

Обозначение: ***B(A).***

Например: ***A = {a; b}. B(A) = {∅; {a}; {b}; {a; b}}.***

$|B(A)| = 2^{|A|}$, т. е. множество, содержащее ***n*** элементов, имеет 2^n подмножеств.

Равенство множеств

Множества *A* и *B* равны ($A = B$) тогда и только тогда, когда $A \subseteq B$ и $B \subseteq A$, т. е. элементы множеств *A* и *B* совпадают.

$$A = B \Leftrightarrow (\forall a \in A \ \exists b \in B, a = b) \ \& \ (\forall b \in B \ \exists a \in A, b = a)$$

Операция включения (подмножества) обладает следующими свойствами:

$\forall A(A \subset A)$;

$\forall A, B(A \subset B \ \& B \subset A \Rightarrow A = B)$;

$\forall A, B, C(A \subset B \ \& \ B \subset C \Rightarrow A \subset C)$.

Дополнение

Для множества *A* множество *B* называется дополнением *A*, если в *B* включены те и только те элементы, которые не принадлежат *A* (обозначается как $B = \backsim A$ или $B = \overline{A}$ или $B = \neg A$). **Эту операцию еще называют НЕ**, т. е. говорят *B* равно НЕ *A* (рис. 11).

$$\overline{A} = \{x \in U \mid x \notin A\}$$

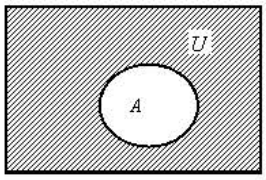


Рис. 11. Диаграмма «Дополнение»

Операция дополнения подразумевает, что задан некоторый **универсум** *U*:

$$\overline{A} = U \backslash A.$$

В противном случае операция дополнения не определена.

Очевидно:

$\overline{\emptyset} = U$	$\overline{U} = \emptyset$	$\overline{\overline{A}} = A$	$A \cup \overline{A} = U$	$A \cap \overline{A} = \emptyset$

Дополнение — одноместная операция.

Операции

Обычно рассматриваются следующие операции над множествами.

Объединение множеств

Объединением множеств A и B называется множество $A \cup B$, каждый элемент которого принадлежит либо множеству A, либо множеству B (рис. 12).

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

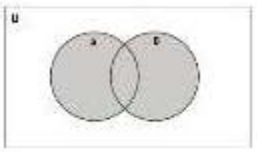


Рис. 12. Диаграмма «Объединение»

1) Если A — произвольное множество, то

$A \cup A = A$	$A \cup \emptyset = A$	$A \cup U = U$

2) Если A и B — произвольные множества, то $A \cup B = B \cup A$.

3) Если $B \subset A$, то $B \cup A = A$

Пересечение множеств

Пересечением множеств A и B называется множество $A \cap B$, каждый элемент которого принадлежит и множеству A, и множеству B (рис. 13).

$$A \cap B = \{x \mid x \in A \,\&\, x \in B\}$$

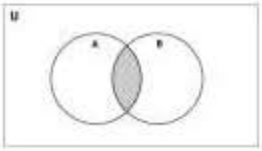


Рис. 13. Диаграмма «Пересечение»

1) Если A — произвольное множество, то

$A \cap A = A$	$A \cap \emptyset = \emptyset$	$A \cap U = A$

2) Если A и B — произвольные множества, то $A \cap B = B \cap A$.

3) Если $B \subset A$, то $B \cap A = B$.

Разность множеств

Разностью множеств A и B называется множество $A \setminus B$, каждый элемент которого принадлежит множеству A и не принадлежит множеству B (рис. 14).

$$A \setminus B = \{x \mid x \in A \,\&\, x \notin B\}$$

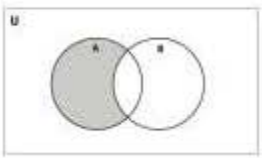


Рис. 14. Диаграмма «Разность»

Если A — произвольное множество, то

$A \setminus A = \emptyset$	$A \setminus \emptyset = A$	$A \setminus U = \emptyset$

В отличие от обьединения и пересечения, разность строго двуместна.

$$A \setminus B \neq B \setminus A$$

Симметричная разность

Симметричной разностью множеств A и B называется множество $A \triangle B$, каждый элемент которого принадлежит или множеству A, или множеству B, но не обеим множествам сразу (рис. 15).

$$A \triangle B - (A \cup B) \setminus (A \cap B) \{x \mid (x \in A \,\&\, x \notin B) \vee (x \notin A \,\&\, x \in B)\}$$

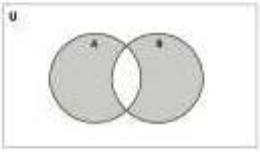


Рис. 15. Диаграмма «Симметричная разность»

Если A — произвольное множество, то

$A \Delta A = \emptyset$	$A \Delta \emptyset = A$	$A \Delta U = \bar{A}$

Пример 3.

Пусть A = {1, 2, 3}, B = {3, 4, 5}. Тогда A U B = {1, 2, 3, 4, 5}, A ∩ B = {3}, A\B = {1, 2}, B\A = {4, 5}, A Δ B = {1, 2, 4, 5}. Если U = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, то $\overline{A} - \{0,4,5,6,7,8,9\}$, $\overline{B} - \{0,1,2,6,7,8,9\}$.

2.1.3. Свойства операций над множествами

Операции над множествами обладают целым рядом важных свойств, которые рассматриваются в этом подразделе [7]. Пусть задан универсум U. Тогда $\forall A, B, C \subset U$ выполняются следующие равенства:

1) идемпотентность:

$$A \cup A = A, \;\; A \cap A = A;$$

2) коммутативность:

$$A \cup B = B \cup A, \;\; A \cap B = B \cap A;$$

3) ассоциативность:

$$A \cup (B \cup C) = (A \cup B) \cup C, \;\; A \cap (B \cap C) = (A \cap B) \cap C;$$

4) дистрибутивность (рис. 16):

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C), \;\; A \cap (B \cup C) = (A \cap B) \cup (A \cap C);$$

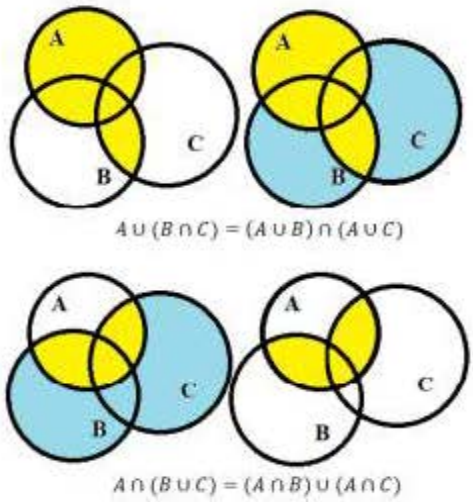


Рис. 16. Операция «Дистрибутивность»

5) поглощение:

$$(A \cap B) \cup A = A, \ \ (A \cup B) \cap A = A;$$

6) свойство нуля:

$$A \cup \emptyset = A, \ \ A \cap \emptyset = \emptyset;$$

7) свойство единицы:

$$A \cup U = U, \ \ A \cap U = A;$$

8) инволютивность:

$$\overline{\overline{A}} = A;$$

9) законы де Моргана (рис. 17):

$$\overline{A \cap B} = \overline{A} \cup \overline{B}, \overline{A \cup B} = \overline{A} \cap \overline{B};$$

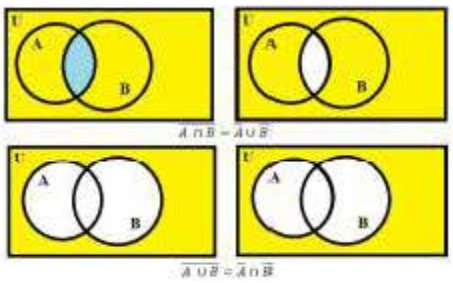


Рис. 17. Законы де Моргана

10) свойства дополнения:

$$A \cup \overline{A} = U, \;\; A \cap \overline{A} = \emptyset;$$

11) выражение для разности:

$$A \backslash B = A \cap \overline{B}.$$

В справедливости перечисленных равенств можно убедиться различными способами. Например, можно нарисовать диаграммы Венна для левой и правой частей равенства и убедиться, что они совпадают, или же провести формальное рассуждение для каждого равенства.

2.2. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ № 2 СОЗДАНИЕ БИБЛИОТЕКИ ФУНКЦИЙ, РЕАЛИЗУЮЩИХ ОПЕРАЦИИ НАД МНОЖЕСТВАМИ

Продолжаем наполнять функциями библиотеку, созданную в лабораторной работе № 1 (файл SetLab1_2_<Фамилия>.cpp.).

В библиотеку добавить следующие функции:

1. **F9. Подмножество A, B.** Входные параметры: два указателя на первые элементы списков — исходных множеств ***A*** и ***B***. Выходные параметры: логическое значение True, если ***A*** является подмножеством ***B***. Использовать **F2**.

2. **F10. Равенство двух множеств A, B.** Входные параметры: два указателя на первые элементы списков — исходных множеств ***A*** и ***B***. Выходные параметры: логическое значение True, если ***A*** равно ***B***. Использовать **F9.**

3. **F11. Объединение двух множеств A, B**. Входные параметры: два указателя на первые элементы списков — исходных множеств. Выходные параметры: указатель на первый элемент списка-результата (третье множество). Использовать **F2, F4.**

4. **F12. Пересечение двух множеств A, B.** Входные параметры: два указателя на первые элементы списков — исходных множеств. Выходные параметры: указатель на первый элемент списка-результата (третье множество). Использовать **F2, F4.**

5. **F13. Разность множеств A, B.** Входные параметры: два указателя на первые элементы списков — исходных множеств. Выходные параметры: указатель на первый элемент списка-результата (третье множество). Использовать **F2, F4.**

6. **F14. Симметричная разность A, B.** Входные параметры: два указателя на первые элементы списков — исходных множеств. Выходные параметры: указатель на первый элемент списка-результата (третье множество). Использовать **F11, F12** и **F13.**

В основной программе SetLab1_2_<Фамилия>.cpp протестировать функции **F9, F10, F11, F12, F13** и **F14**.

Для функции **F9** протестировать $A \subset B$ и $A \subset A$.

Для функции **F10** протестировать $A=B$ и $A=A$.

Для функции **F13** получить как $A-B$, так и $B-A$.

Значения элементов множеств выбирать согласно перечисленным ниже вариантам (по номеру в списке группы). В соответствии с вариантом задания модифицировать функцию **F5** (создание множества) — добавить новый параметр, указывающий, какое множество — ***A*** или ***B*** — будет создано. Мощность множеств — от 6 до 9 элементов (меняется от запуска к запуску). Значения множеств двузначные (от 10 до 99). Правую границу интервала значений подобрать так, чтобы с достаточной вероятностью операция пересечения не давала пустое множество.

Весь вывод осуществлять **только** в главной программе.

Вывод результатов каждой операции **только с пояснительным текстом**, т. е. название операции, результат.

Варианты заданий

1. Множество ***A*** — множество чисел, кратных 4-м. Множество ***B*** — множество четных чисел.

2. Множество ***A*** — множество чисел, последняя цифра которых больше 3-х. Множество ***B*** — множество чисел, последняя цифра которых меньше 8-и.

3. Множество ***A*** — множество чисел, кратных 3-м. Множество ***B*** — множество нечетных чисел.

4. Множество A — множество чисел, последняя цифра которых меньше 7-и. Множество B — множество чисел, последняя цифра которых меньше 4-х.

5. Множество A — множество чисел, кратных 5-и. Множество B — множество нечетных чисел.

6. Множество A — множество чисел, последняя цифра которых больше 7-и. Множество B — множество чисел, последняя цифра которых больше 5-и.

7. Множество A — множество чисел, кратных 7-и. Множество B — множество нечетных чисел.

8. Множество A — множество чисел, последняя цифра которых меньше 8-и. Множество B — множество чисел, последняя цифра которых больше 3-х.

9. Множество A — множество чисел, кратных 3-м. Множество B — множество чисел, кратных 9-и.

10. Множество A — множество чисел, кратных 6-и. Множество B — множество четных чисел.

11. Множество A — множество чисел, кратных 3-м. Множество B — множество чисел, кратных 6-и.

12. Множество A — множество чисел, кратных 5-и. Множество B — множество чисел, кратных 10-и.

13. Множество A — множество чисел, кратных 4-м. Множество B — множество чисел, кратных 8-и.

14. Множество A — множество чисел, кратных 4-м. Множество B — множество чисел, кратных 6-и.

15. Множество A — множество чисел, кратных 6-и. Множество B — множество чисел, кратных 9-и.

16. Множество A — множество чисел, последняя цифра которых больше 3-х. Множество B — множество чисел, последняя цифра которых меньше 8-и.

17. Множество A — множество чисел, последняя цифра которых больше 5-и. Множество B — множество чисел, последняя цифра которых больше 7-и.

18. Множество A — множество чисел, последняя цифра которых меньше 4-х. Множество B — множество чисел, последняя цифра которых меньше 7-и.

19. Множество A — множество чисел, последняя цифра которых меньше 8-и. Множество B — множество чисел, последняя цифра которых больше 3-х.

20. Множество A — множество четных чисел. Множество B — множество чисел, кратных 4-м.

21. Множество A — множество нечетных чисел. Множество B — множество чисел, кратных 3-м.

22. Множество A — множество нечетных чисел. Множество B — множество чисел, кратных 5-и.

23. Множество A — множество нечетных чисел. Множество B — множество чисел, кратных 7-и.

24. Множество A — множество чисел, кратных 9-и. Множество B — множество чисел, кратных 3-м.

25. Множество A — множество четных чисел. Множество B — множество чисел, кратных 6-и.

26. Множество A — множество чисел, кратных 6-и. Множество B — множество чисел, кратных 3-м.

27. Множество A — множество чисел, кратных 10-и. Множество B — множество чисел, кратных 5-и.

28. Множество A — множество чисел, кратных 8-и. Множество B — множество чисел, кратных 4-м.

29. Множество A — множество чисел, кратных 6-и. Множество B — множество чисел, кратных 4-м.

30. Множество A — множество чисел, кратных 9-и. Множество B — множество чисел, кратных 6-и.

3. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Цель лабораторной работы — разработка функций, реализующих базовые операции над множествами с использованием объектно-ориентированного программирования.

3.1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

3.1.1. Основные сведения [8]

Введение понятия класса является естественным развитием идей модульности. В классе структуры данных и функции их обработки объединяются. Класс используется только через его интерфейс, детали реализации для пользователя класса несущественны.

Идея классов отражает строение объектов реального мира, ведь каждые предмет или процесс обладают набором характеристик или отличительных черт, иными словами, свойствами и поведением. Программы часто предназначены для моделирования предметов, процессов и явлений реального мира, поэтому в языке программирования удобно иметь адекватный инструмент для представления моделей.

Класс является типом данных, определяемым пользователем. В классе задаются свойства и поведение каких-либо предмета или процесса в виде полей данных и функций для работы с ними. Создаваемый тип данных обладает практически теми же свойствами, что и стандартные типы. Существенным свойством класса является то, что детали его реализации скрыты от пользователей класса за интерфейсом. Интерфейсом класса являются заголовки его методов. Таким образом, класс как модель объекта реального мира является черным ящиком, замкнутым по отношению к внешнему миру.

Идея классов является основой объектно-ориентированного программирования (ООП). ООП — это не просто набор новых средств, добавленных в язык. Объектно-ориентированная программа строится в терминах объектов и их взаимосвязей.

Конкретные величины типа данных «класс» называются экземплярами класса, или объектами. Объекты взаимодействуют между собой, посылая и получая сообщения. Сообщение — это запрос на выполнение действия, содержащий набор необходимых параметров. Механизм сообщений реализуется с помощью вызова соответствующих функций.

3.1.2. Свойства ООП

Основными свойствами ООП являются инкапсуляция, наследование и полиморфизм.

Объединение данных с функциями их обработки в сочетании со скрытием ненужной для использования этих данных информации называется **инкапсуляцией**. Инкапсуляция повышает степень абстракции программы: данные класса и реализация его функций находятся ниже уровня абстракции, и для написания программы информация о них не требуется. Кроме того, инкапсуляция позволяет изменить реализацию класса без модификации основной части программы, если интерфейс остался прежним.

Инкапсуляция позволяет использовать класс в другом окружении и быть уверенным, что он не испортит не принадлежащие ему области памяти, а также создавать библиотеки классов для применения во многих программах.

Наследование — это возможность создания иерархии классов, когда потомки наследуют все свойства своих предков, могут их изменять и добавлять новые. Свойства при наследовании повторно не описываются, что сокращает объем программы. Выделение общих черт различных классов в один класс-предок является мощным механизмом абстракции, ведь и любая наука начинается с абстрагирования и классификации, которые помогают справиться со сложностью рассматриваемой предметной области.

Иерархия классов представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные — на ветвях и листьях.

Третьим китом, на котором стоит ООП, является **полиморфизм** — возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы. Понятие полиморфизма используется в C++ весьма

широко. Простым примером полиморфизма может служить перегрузка функций, когда из нескольких вариантов выбирается наиболее подходящая функция по соответствию ее прототипа передаваемым параметрам.

Другой пример — использование шаблонов функций, когда один и тот же код видоизменяется в соответствии с типом, переданным в качестве параметра.

3.1.3. Описание класса

Класс является абстрактным типом данных, определяемым пользователем, и представляет собой модель реального объекта в виде данных и функций для работы с ними. Данные класса называются **полями**, а функции класса — **методами**. Поля и методы называются **элементами** класса. Описание класса выглядит так:

```
class <имя>{
  [ private: ]
  <описание скрытых элементов>
  public:
  <описание доступных элементов>
}; // Описание заканчивается точкой с запятой
```

Спецификаторы доступа **private** и **public** управляют видимостью элементов класса. Элементы, описанные после служебного слова **private**, видимы только внутри класса. Этот вид доступа принят в классе по умолчанию.

Интерфейс класса описывается после спецификатора **public**. Действие любого спецификатора распространяется до следующего спецификатора или до конца класса. Можно задавать несколько секций private и public, порядок их следования значения не имеет.

Поля класса:

– могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот класс);

– могут быть описаны с модификатором const, при этом они инициализируются только один раз (с помощью конструктора) и не могут изменяться;

– могут быть описаны с модификатором static, но не как auto, extern и register.

Инициализация полей при описании не допускается.

В каждом классе есть хотя бы один метод, имя которого совпадает с именем класса. Он называется **конструктором** и вызывается автоматически при создании объекта класса. Конструктор предназначен для инициализации объекта. Автоматический вызов конструктора позволяет избежать ошибок, связанных с использованием неинициализированных переменных.

3.1.4. Описание объектов

Конкретные переменные типа «класс» называются **экземплярами** класса, или объектами.

При создании каждого объекта выделяется память, достаточная для хранения всех его полей, и автоматически вызывается **конструктор**, выполняющий их инициализацию. Методы класса не тиражируются. При выходе объекта из области действия он уничтожается, при этом автоматически вызывается **деструктор**.

Доступ к элементам объекта аналогичен доступу к полям структуры. Для этого используются операция «.» (точка) при обращении к элементу через имя объекта и операция -> при обращении через указатель.

Получить или изменить значения элементов со спецификатором private можно столько через обращение к соответствующим методам.

3.1.5. Конструкторы

Конструктор предназначен для инициализации объекта и вызывается автоматически при его создании. Ниже перечислены основные свойства конструкторов:

– конструктор не возвращает значение, даже типа void. Нельзя получить указатель на конструктор;

– класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки);

– конструктор, вызываемый без параметров, называется конструктором по умолчанию;

– параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов;

– если программист не указал ни одного конструктора, компилятор создает его автоматически. Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов. В случае когда класс содержит константы или ссылки, при попытке создания объекта класса будет выдана ошибка, поскольку их необходимо инициализировать конкретными значениями, а конструктор по умолчанию этого делать не умеет;

– конструкторы не наследуются;

– конструкторы нельзя описывать с модификаторами const, virtual и static.

– конструкторы глобальных объектов вызываются до вызова функции main. Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).

3.1.6. Деструкторы

Деструктор — это особый вид метода, применяющийся для освобождения памяти, занимаемой объектом. Деструктор вызывается автоматически, когда объект выходит из области видимости:

– для локальных объектов — при выходе из блока, в котором они объявлены;

– для глобальных — как часть процедуры выхода из main:

– для объектов, заданных через указатели, деструктор вызывается неявно при использовании операции delete.

Имя деструктора начинается с тильды (~), непосредственно за которой следует имя класса.

Деструктор:

– не имеет аргументов и возвращаемого значения;

– не может быть объявлен как const или static;

– не наследуется;

– может быть виртуальным

Если деструктор явным образом не определен, компилятор автоматически создает пустой деструктор.

Описывать в классе деструктор явным образом требуется в случае, когда объект содержит указатели на память, выделяемую динамически, иначе при уничтожении объекта память, на которую ссылались его поля-указатели, не будет помечена как свободная. Указатель на деструктор определить нельзя.

3.1.7. Рекомендации по составу класса

Как правило, класс как тип, определенный пользователем, должен содержать скрытые (private) поля и следующие функции:

– конструкторы, определяющие, как инициализируются объекты класса;

– набор методов, реализующих свойства класса;

– набор операций, позволяющих копировать, присваивать, сравнивать объекты и производить с ними другие действия;

– класс исключений, используемый для сообщений об ошибках с помощью генерации исключительных ситуаций.

Если существуют функции, которые работают с классом или несколькими классами через интерфейс (то есть доступ к скрытым полям им не требуется), можно описать их вне классов, чтобы не перегружать интерфейсы, а для обеспечения логической связи поместить их в общее с этими классами пространство имен.

3.2. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ № 3 РАЗРАБОТКА КЛАССА «МНОЖЕСТВО» НА ОСНОВЕ ОДНОСВЯЗНОГО СПИСКА

Разработать класс «Множество» на основе связного списка. Класс будет содержать один параметр — указатель на голову списка, а также 14 методов, соответствующих функциям, разработанным в лабораторных работах № 1 и № 2 (F1 — конструктор, F8 — деструктор).

Все требования, описанные в лабораторных работах № 1 и № 2, остаются. Файл с реализацией методов назвать SetLab3_<Фамилия>.cpp, присоединить к

основной тестирующей программе с помощью заголовочного h-файла. Основной файл — OOP_Set_<Фамилия>.cpp. Пишем также в консольном приложении на C++.

В основной программе OOP_Set_<Фамилия>.cpp протестировать функции F5 (A и B), F6 (A и B), F7 (A и B), F9 (A и B, A и A), F10 (A и B, A и A), F11, F12, F13 (A и B, B и A), F14 и F8 (A и B).

Значения элементов множеств выбирать согласно перечисленным в лабораторной работе № 2 вариантам.

Весь вывод осуществлять **только** в главной программе.

Вывод результатов каждой операции **только с пояснительным текстом**, т. е. название операции, результат.

4. КОНТЕЙНЕРНЫЙ КЛАСС LIST

4.1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

4.1.1. Общие сведения

Контейнерные классы — это классы, предназначенные для хранения данных, организованных определенным образом [8]. Примерами контейнеров могут служить массивы, линейные списки или стеки. Для каждого типа контейнера определены методы для работы с его элементами.

Использование контейнеров позволяет значительно повысить надежность программ, их переносимость и универсальность, а также уменьшить сроки их разработки. Эти преимущества не даются даром: универсальность и безопасность использования контейнерных классов не могут не отражаться на быстродействии программы. Снижение быстродействия в зависимости от реализации компилятора может быть весьма значительным. Кроме того, для эффективного использования контейнеров требуется затратить усилия на освоение библиотеки.

Существуют несколько подходов к классификации контейнерных классов. Один из них приведен на рис. 18.

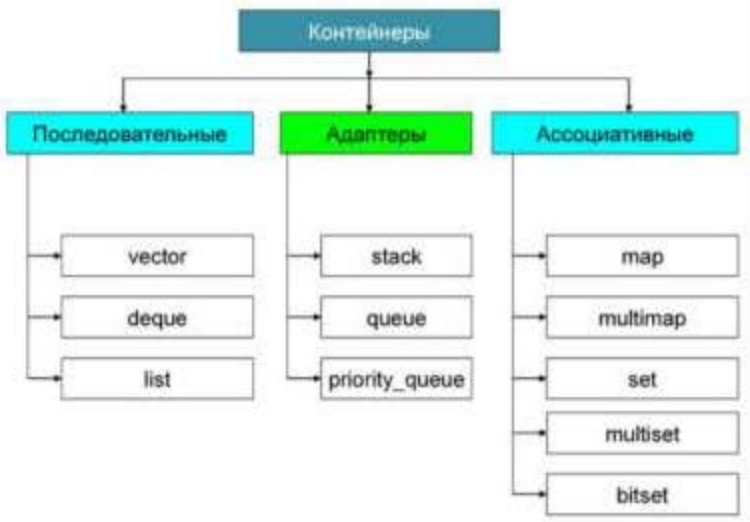


Рис. 18. Классификация контейнерных классов

Последовательные контейнеры обеспечивают хранение однотипных величин в виде непрерывной последовательности, т. е. определены понятия «начальный элемент», «конечный элемент», «предыдущий элемент», «последующий элемент». Порядок элементов в контейнере зависит от порядка и способа добавления элементов в контейнер, а не от значений элементов. Другими словами, пользователь сам выбирает, в каком порядке элементы будут храниться, указывая, в какое место контейнера он хочет записать или добавить элемент.

К последовательным контейнерам относятся векторы (vector), двусторонние очереди (deque), списки (list) и односвязные списки (forward_list), а также так называемые **адаптеры**, то есть варианты контейнеров, включая стеки (stack), очереди (queue) и очереди с приоритетами (priority_queue). Еще одним видом контейнера с ограниченным набором операций является массив (array).

Каждый вид контейнера обеспечивает свой набор действий над данными. Выбор вида контейнера зависит от того, что требуется делать с данными в программе. Например, при необходимости часто вставлять и удалять элементы из середины последовательности следует использовать список, а если включение элементов выполняется главным образом в конец или начало — двустороннюю очередь.

Все рассматриваемые последовательные контейнеры допускают вставку новых элементов в любую позицию и удаление элементов из любой позиции. Векторы оптимизированы для быстрого (за константное время) выполнения операций вставки и удаления, связанных с концом последовательности элементов (функции-члены push_back и pop_back), а деки — для операций, связанных как с началом, так и с концом последовательности (функции-члены push_back и pop_back, push_front и pop_front) [9].

В то же время векторы обладают рядом особенностей, отсутствующих у деков; в частности, они имеют такую характеристику, как емкость, которая доступна и для чтения (функция-член capacity), и для изменения (функция-член reserve).

Текстовые строки string обладают возможностями, аналогичными возможностям векторов с символьными элементами.

Списки позволяют выполнять быструю вставку и удаление элементов для любой позиции, однако доступ к элементу списка по его номеру требует линейного времени (т. е. зависит от текущего размера списка). По этой причине для списков не реализована операция индексирования, а связанные со списками

итераторы являются двунаправленными (а не итераторами произвольного доступа, как для всех остальных последовательных контейнеров).

Использование **ассоциативных контейнеров** предполагает, что в хранимых данных выделяются ключ, который определяет конкретный элемент хранимых данных, и неключевая информация. Таким образом, ассоциативные контейнеры обеспечивают быстрый доступ к данным по значению ключа.

Эти контейнеры построены на основе сбалансированных деревьев. Существуют пять типов ассоциативных контейнеров: словари (map), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset).

Все рассматриваемые ассоциативные контейнеры хранят последовательности своих элементов в отсортированном виде. Сортировка выполняется по ключу, причем для множеств и мультимножеств ключами выступают сами элементы (типа T), а в отображениях и мультиотображениях хранятся пары типа pair, первый компонент которых считается ключом (key), а второй — значением (value).

Мультимножества и мультиотображения, в отличие от множеств и отображений, позволяют хранить набор элементов с эквивалентными ключами (ключи считаются эквивалентными, если ни один из них не является меньшим, чем другой).

Вставка новых элементов в любой ассоциативный контейнер сохраняет его упорядоченность. И операция вставки, и операция удаления для ассоциативных контейнеров требуют логарифмического времени, если для этих операций указывается параметр-ключ. За это же время выполняется и поиск элементов по ключу, для реализации которого в ассоциативных контейнерах предусмотрен целый набор функций-членов. Указанные свойства ассоциативных контейнеров делают их удобным механизмом для группировки и объединения наборов данных по ключу.

Практически в любом контейнерном классе определены поля перечисленных ниже типов [8]:

value_type — тип элемента контейнера;

size_type — тип индексов, счетчиков элементов и т. д.;

iterator — итератор;

const_iterator — константный итератор;

reverse_iterator — обратный итератор;

const_reverse_iterator — константный обратный итератор;

reference — ссылка на элемент;

const_reference — константная ссылка на элемент;

key_type — тип ключа (для ассоциативных контейнеров);

key_compare — тип критерия сравнения (для ассоциативных контейнеров).

Итератор является аналогом указателя на элемент. Он используется для просмотра контейнера в прямом или обратном направлениях. Все, что требуется от итератора, — уметь ссылаться на элемент контейнера и реализовывать операцию перехода к его следующему элементу.

Константные итераторы используются тогда, когда значения соответствующих элементов контейнера не изменяются. При помощи итераторов можно просматривать контейнеры, не заботясь о фактических типах данных, используемых для доступа к элементам. Для этого в каждом контейнере определено несколько методов:

iterator begin(), const_iterator begin() const — указывают на первый элемент;

iterator end(), const_iterator end() const — указывают на элемент, следующий за последним;

reverse_iterator rbegin(), const_reverse_iterator rbegin() const — указывают на первый элемент в обратной последовательности;

reverse_iterator rend(), const_reverse_iterator rend() const — указывают на элемент, следующий за последним, в обратной последовательности.

Во всех контейнерах определены методы, позволяющие получить сведения о размере контейнеров:

size() — возвращает число элементов;

max_size() — возвращает максимальный размер контейнера (порядка миллиарда элементов);

empty() — булевская функция, показывающая, пуст ли контейнер;

swap(other) — меняет местами содержимое данного контейнера и контейнера other того же типа;

clear() — удаляет все элементы контейнера.

Ниже приведены функции-члены последовательных контейнеров [9].

void assign(n, x) — удаляет все элементы контейнера и копирует в него новые данные (n копий значения x);

void assign(InIterFirst, InIterLast) — удаляет все элементы контейнера и копирует в него новые данные (из диапазона [InIterFirst, InIterLast]);

void assign(init_list) — удаляет все элементы контейнера и копирует в него новые данные (из списка инициализации init_list;

reference operator[](n) — возвращает ссылку на элемент с индексом n (0 <= n < size());

reference at(n) — возвращает ссылку на элемент с индексом n (0 <= n < size());

reference back() — возвращает ссылку на последний элемент контейнера;

reference front() — возвращает ссылку на первый элемент контейнера;

size_type capacity() const — возвращает текущую емкость контейнера;

iterator erase(pos), iterator erase(first, last) — удаляет элемент на позиции pos или все элементы в диапазоне (first, last) и возвращает итератор, указывающий на элемент, следующий за последним удаленным элементом (или итератор end(), если были удалены конечные элементы контейнера);

iterator insert(pos, x), void insert(pos, n, x), void insert(pos, InIterFirst, InIterLast) — вставляет в контейнер новые данные, начиная с позиции pos (соответственно, одно или n значений x или элементы из диапазона InIterFirst, InIterLast);

void pop_back() — удаляет последний элемент;

void pop_front() — удаляет первый элемент;

void push_back(x) — добавляет x в конец контейнера.

void push_front(x) — добавляет x в начало контейнера;

void resize(n, x = T()) — изменяет размер контейнера, делая его равным n. Если n > size(), то в конец контейнера добавляется требуемое число копий x. Если n < size(), то удаляется требуемое количество конечных элементов контейнера.

4.1.2. Дополнительные функции-члены класса list

Контейнер list представляет двухсвязный список, то есть такой список, где каждый элемент имеет указатели на предыдущий и последующий элементы с возможностью добавления (и удаления) элементов и в начало, и в конец, и в середину списка за время O(1), благодаря чему мы можем перемещаться по списку как вперед, так и назад [9].

Минусом двусвязного списка является отсутствие возможности произвольного доступа, которое есть у массива и вектора.

Для использования списка необходимо подключить библиотеку «list»:

#include <list>

Ниже приведен пример создания списка:

list<int> A — создание пустого списка для хранения элементов типа **int**;

list<int> A(n, value) — создание список из n элементов со значением value.

В отличие от других контейнеров, для типа list не определены операция обращения по индексу или функция at(), которая выполняет похожую задачу.

Тем не менее для контейнера list можно использовать функции front() и back(), которые возвращают первый и последний элементы соответственно.

Чтобы обратиться к элементам, которые находятся в середине (после первого и до последнего элемента), придется выполнять перебор элементов с помощью циклов или итераторов.

Ниже приведен пример последовательного обхода списка:

```
list<double> A;
list<double>::iterator it;
for(it=A.begin(); it!=A.end(); it++)
{
sum+=(*it); //доступ к элементам по итератору
}
```

Некоторые методы класса list приведены ниже:

front() — первый элемент списка;

back() — последний элемент списка;

push_back(value) — добавляет элемент в конец списка;

push_front(value) — добавляет элемент в начало списка;

insert(it, value) — вставка некоторого значения value перед it;

insert(i, n, value) — вставка n копий элементов со значением value перед i;

pop_front() — удаляет первый элемент списка;

pop_back() — удаляет последний элемент (не возвращает значение);

erase(it) — удаляет элемент, на который указывает данный итератор;

erase(start, fin) — удалению подлежат все элементы между start и fin, но fin не удаляется;

clear() — удаляет все элементы списка;

size() — выдает количество элементов списка;

empty() — возвращает истину, если вызывающий список пуст.

К дополнительным функциям класса **list** также относятся:

emplace(Where, val) — вставляет элемент в указанное положение в списке. **Where** — позиция в целевом объекте, **val** — элемент, добавляемый в list;

merge(lst[, comp]) — выполняет операцию слияния текущего списка и списка lst того же типа (оба списка должны быть предварительно отсортированы). При слиянии элементы сравниваются с помощью операции < или предиката comp, если он явно указан (и эти же операция или предикат должны быть ранее использованы для сортировки списков). Слияние является устойчивым, т. е. относительный порядок следования элементов исходных списков не нарушается. Если «одинаковые» элементы присутствуют как в текущем списке, так и в списке lst, то элемент из lst помещается после элемента, уже присутствующего в текущем списке. В результате слияния список lst становится пустым;

remove(x), remove_if(pred) — удаляет из списка, соответственно, все вхождения элемента **x** или все элементы, для которых предикат **pred** возвращает значение true;

reverse() — изменяет порядок элементов списка на обратный;

sort([comp]) — выполняет сортировку списка, используя операцию < или предикат **comp**, если он явно указан. Сортировка является устойчивой, т. е. относительный порядок элементов с одинаковыми ключами сортировки не изменяется;

splice(pos, lst), splice(pos, lst, pos_lst), splice(pos, lst, first_lst, last_lst) — перемещает элементы из списка **lst** в текущий список (элементы размещаются, начиная с позиции **pos**). Перемещаются, соответственно, все элементы списка **lst**, или элемент списка **lst**, расположенный на позиции **pos_lst**, или элементы списка **lst** из диапазона [**first_lst, last_lst**);

unique([pred]) — удаляет соседние «одинаковые» элементы списка, оставляя первый из набора «одинаковых» элементов. Для сравнения элементов используются операция == или предикат **pred**, если он явно указан.

4.2. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ № 4
РАЗРАБОТКА КЛАССА «МНОЖЕСТВО»
НА ОСНОВЕ КОНТЕЙНЕРА LIST

Разработать класс «Множество» на основе контейнера list. Класс будет содержать один параметр — целочисленный список, а также 14 методов, соответствующих функциям, разработанным в лабораторных работах № 1 и № 2 (F1 — конструктор).

Все требования, описанные в лабораторных работах № 1 и № 2, остаются в силе. Добавление нового элемента в список необходимо производить строго в начало списка. Файл с реализацией методов назвать SetLab4_<Фамилия>.cpp, присоединить к основной тестирующей программе с помощью заголовочного h-файла. Основной файл — Set_Cont_List_<Фамилия>.cpp. Пишем также в консольном приложении на C++. В методах максимально используем методы контейнера list. Для перемещения по списку используем итераторы.

В основной программе Set_Cont_List_<Фамилия>.cpp протестировать функции F5, F6, F7, F8, F9, F10, F11, F12, F13 и F14.

Значения элементов множеств выбирать согласно перечисленным в лабораторной работе № 2 вариантам.

5. КОНТЕЙНЕРНЫЙ КЛАСС SET

5.1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Контейнерный класс set стандартной библиотеки C++ используется для хранения и извлечения данных из коллекции. Значения элементов в наборе set уникальны и служат ключевыми значениями, в соответствии с которыми данные автоматически упорядочены. После добавления некоторого количества элементов они всегда будут упорядочены по возрастанию. При попытке добавить уже существующий в контейнере элемент не возникнет ошибки, но элемент добавлен не будет. Значение элемента в set не может изменяться напрямую. Вместо этого необходимо удалить старые значения и вставить элементы с новыми значениями.

В основе множеств лежат бинарные красно-черные деревья. Операции поиска, удаления и вставки имеют логарифмическую сложность.

Для использования множеств необходимо подключить библиотеку «set»:

#include <set>

Ниже приведен пример создания множества:

set <int> s0; — создание пустого множества **s0** целочисленного типа.

Некоторые методы класса set приведены ниже [10]:

begin() — возвращает итератор, обращающийся к первому элементу в множестве;

clear() — стирает все элементы в наборе;

contains() — проверяет, есть ли элемент с указанным ключом в элементе set;

count() — возвращает число элементов в наборе, ключи которых соответствуют ключу, заданному параметром. Возвращаемое значение — 1, если набор содержит элемент, ключ сортировки которого совпадает с ключом параметра, значение 0, если набор не содержит элемент с соответствующим ключом;

emplace() — вставляет элемент в множество. Возвращает пару: логическое значение true, если вставка была выполнена, и значение false, если множество уже содержало элемент, и адрес, куда был вставлен новый элемент (если

первое значение — true) или где уже находился элемент (если первое значение — false);

empty() — проверяет, пуст ли набор. Возвращает логическое значение;

end() — возврат итератора после последнего элемента множества. Используется для проверки того, прошел ли итератор до конца множества;

erase() — удаляет элемент или диапазон элементов в наборе с заданных позиций или удаляет элементы, соответствующие заданному ключу;

find() — возвращает итератор, ссылающийся на элемент в наборе, ключ которого эквивалентен заданному ключу. Возвращаемое значение — итератор, ссылающийся на расположение элемента с указанным ключом или на расположение элемента после последнего элемента в наборе (set::end()), если для ключа не найдено совпадений;

insert() — вставляет элемент или диапазон элементов в набор. Возвращаемое значение — возвращает пару, первый компонент которой bool имеет значение true, если вставка была выполнена, и значение false, если множество уже содержало вставляемый элемент. Второй компонент итератора пары возвращаемых значений указывает на вставленный элемент, если значение первого компонента true, или на существующий элемент, если значение первого компонента false;

max_size() — возвращает максимальную длину множества;

size() — возвращает количество элементов в наборе;

Кроме того, контейнерный класс set поддерживает различные операторы. Для наших лабораторной работы и курсовой интерес представляет оператор проверки равенства.

Оператор == проверяет равенство объекта-множества слева от оператора объекту-множеству справа от оператора. Значение **true**, если набор в левой части оператора равен набору справа от оператора, **false** — в противном случае. Сравнение между объектами-множествами основывается на попарном сравнении элементов этих множеств. Два множества равны, если они содержат одинаковое количество элементов, а их соответствующие элементы имеют одинаковые значения. В противном случае они неравны.

Пример [10]:

```
set <int> s1, s2, s3;
int i;

for ( i = 0 ; i < 3 ; i++ )
{
  s1.insert ( i );
  s2.insert ( i * i );
  s3.insert ( i );
}

if ( s1 == s2 )
          cout << "Множества s1 и s2 равны." << endl;
  else
  cout << "Множества s1 и s2 не равны." << endl;

if ( s1 == s3 )
  cout << "Множества s1 и s3 равны." << endl;
else
  cout << " Множества s1 и s3 не равны." << endl;
```

5.2. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ № 5 РАЗРАБОТКА КЛАССА «МНОЖЕСТВО» НА ОСНОВЕ КОНТЕЙНЕРА SET

Разработать класс «Множество» на основе контейнера Set. Класс будет содержать один параметр — целочисленное множество, а также 14 методов, соответствующих функциям, разработанным в лабораторных работах № 1 и № 2 (F1 — конструктор).

Все требования, описанные в лабораторных работах № 1 и № 2, остаются в силе. Файл с реализацией методов назвать SetLab5_<Фамилия>.cpp, присоединить к основной тестирующей программе с помощью заголовочного h-файла. Основной файл — Set_Cont_Set_<Фамилия>.cpp. Пишем также в консольном приложении на C++. В методах максимально используем методы контейнера Set. Для перемещения по множеству используем итераторы.

В основной программе Set_Cont_Set_<Фамилия>.cpp протестировать функции F5, F6, F7, F8, F9, F10, F11, F12, F13 и F14.

Значения элементов множеств выбирать согласно перечисленным в лабораторной работе № 2 вариантам.

6. ОБЗОР КОНТЕЙНЕРНЫХ КЛАССОВ

6.1. КОНТЕЙНЕРНЫЙ КЛАСС VECTOR

6.1.1. Назначение контейнерного класса vector

Вектор — это структура данных, которая хранит элементы, подобно обычному массиву «Класс вектора» стандартной библиотеки C++ — это шаблон класса для контейнеров последовательности. Вектор хранит элементы заданного типа в линейном расположении и обеспечивает быстрый случайный доступ к любому элементу. Вектор — это предпочтительный контейнер для последовательности, если приоритетной является производительность операции случайного доступа.

Для использования этого контейнерного класса необходимо подключить библиотеку «vector»:

#include <vector>;

6.1.2. Методы контейнерного класса vector

Для работы с размером и объемом вектора определены следующие методы:

- empty() — возвращает true, если вектор пуст;
- size() — возвращает количество элементов в векторе;
- max_size() — возвращает максимально возможное количество элементов в векторе;
- reserve() — устанавливает минимально возможное количество элементов в векторе;
- capacity() — возвращает количество элементов, которое может содержать вектор до того, как ему потребуется выделить больше места;
- shrink_to_fit() — уменьшает количество используемой памяти за счет освобождения неиспользованной.

Для вектора определены следующие модификаторы:

- clear() — очищает контейнер;
- insert() — вставляет элементы;

- emplace() — создает элементы на месте (в позиции итератора). Не используются копирование или перемещение. В определенных задачах может использоваться вместо insert();
- emplace_back() — создает элементы на месте в конце массива;
- erase() — удаляет элементы;
- push_back() — добавляет элемент в конец;
- pop_back() — удаляет последний элемент;
- resize() — изменяет количество хранимых элементов;
- swap() — обменивает содержимое;
- assign() — заменяет содержимое вектора на новое.

Для контейнера vector определены операции сравнения ==,!=,<,<=,>,>=. Сравнение двух массивов осуществляется поэлементно.

6.1.3. Пример использования контейнерного класса vector

В качестве примера использования контейнерного класса vector рассмотрим программу поиска минимума и максимума.

```
#include <iostream>
#include <vector>
int main()
{
   std::vector<int> v;
   for (int i = 0; i < 10; i++)
   {
      v.push_back(rand() % 1000);
   }
   for (int i = 0; i < v.size(); i++)
   {
      std::cout << v[i] << ", ";
   }
   std::cout << std::endl;
   int min = 1000;
   for (int i = 0; i < v.size(); i++) {
      if (v[i] < min)
      {
```

```
            min = v[i];
        }
    }
    std::cout << "min=" << min << std::endl;
    int max = 0;
    for (int i = 0; i < v.size(); i++) {
        if (v[i] > max)
        {
            max = v[i];
        }
    }
    std::cout << "max=" << max << std::endl;
}
```

Программа демонстрирует подключение соответствующей библиотеки, задание вектора целочисленных значений и его инициализацию путем добавления новых значений в конец вектора, вывод элементов вектора на экран, а также поиск минимального и максимального значений вектора с выводом результатов. Результат работы программы приведен на рис. 19.

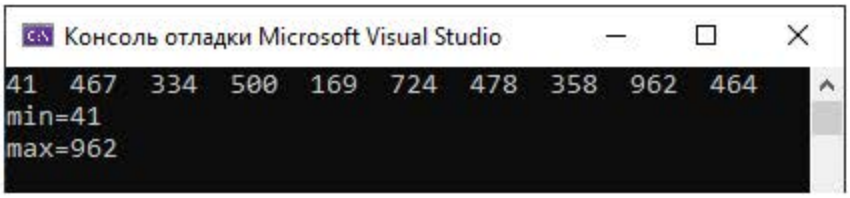


Рис. 19. Пример использования контейнерного класса vector

6.2. КОНТЕЙНЕРНЫЙ КЛАСС STACK

6.2.1. Назначение контейнерного класса stack

В языке программирования C++ stack представляет собой контейнер, реализующий структуру данных «стек» (или «очередь с ограниченным доступом»). В стеке элементы добавляются и удаляются только с одного конца, называемого вершиной стека. Этот контейнер предоставляет удобный интерфейс для работы с данными по принципу «последний вошел, первый вышел» (LIFO).

Для использования этого контейнерного класса необходимо подключить библиотеку «stack»:

#include <stack>

6.2.2. Методы контейнерного класса stack

Для контейнерного класса «stack» определены следующие методы:

- top() — доступ к верхнему элементу stack;
- empty() — возвращает true, если stack пуст;
- size() — количество элементов в стеке;
- push() — добавляет элемент в вершину стека;
- emplace() — создание элемента на месте с добавлением в вершину стека;
- pop() — удаляет элемент с вершины стека.

6.2.3. Пример использования контейнерного класса stack

В качестве примера использования контейнерного класса stack рассмотрим программу изменения порядка следования элементов в последовательности на обратный.

```
#include <iostream>
#include <stack>

int main()
{
   std::stack<int> num;
   for (int i = 0; i < 10; i++)
   {
      num.push(i);
      std::cout << i << ", ";
   }
   std::cout << std::endl;

   while (!num.empty())
   {
      std::cout << num.top() << ", ";
      num.pop();
   }
   std::cout << std::endl;

}
```

Программа демонстрирует подключение соответствующей библиотеки, задание стека целочисленных значений и его инициализацию путем добавления последовательности значений от 0 до 9, вывод элементов стека на экран с помощью обращения к его вершине с последующей выборкой элемента из вершины стека. Результат работы программы приведен на рис. 20.

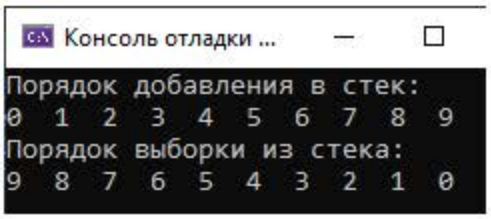


Рис. 20. Пример использования контейнерного класса stack

6.3. КОНТЕЙНЕРНЫЙ КЛАСС QUEUE

6.3.1. Назначение контейнерного класса queue

Класс queue представляет очередь — контейнер, который работает по принципу FIFO («первым вошел — первым вышел»): первым всегда извлекается первый добавленный элемент.

Для использования необходимо подключить библиотеку «queue»:

#include <queue>

6.3.2. Методы контейнерного класса queue

Для контейнерного класса «queue» определены следующие методы:

- front() — доступ к первому элементу;
- back() — доступ к последнему элементу;
- empty() — возвращает true, если queue пуст;
- size() — количество элементов в очереди;
- push() — вставляет элемент в конец очереди;
- emplace() — создаёт элемент на месте с добавлением в конец очереди;
- pop() — удаляет первый элемент.

6.3.3. Пример использования контейнерного класса queue

В качестве примера использования контейнерного класса queue и в целях отражения отличия очереди от стека рассмотрим тот же самый код, что и в разделе 6.2.3, но не для стека, а для очереди.

```
#include <iostream>
#include <queue>

int main()
{
   setlocale(LC_ALL, "Russian");
   std::queue<int> num;
   std::cout << "Порядок добавления в очередь: " << std::endl;
   for (int i = 0; i < 10; i++)
   {
      num.push(i);
      std::cout << i << "  ";
   }
   std::cout << std::endl;
   std::cout << "Порядок выборки из очереди: " << std::endl;
   while (!num.empty())
   {
      std::cout << num.front() << "  ";
      num.pop();
   }
   std::cout << std::endl;

}
```

Как видно из кода, для добавления в очередь очередного элемента также используется метод push(). Однако, в отличие от стека, добавление выполняется в конец очереди. А вот для выборки элемента используется метод front(). И выборка осуществляется с начала очереди. Для удаления первого элемента очереди также используется метод pop(). Результат работы программы приведен на рис. 21.

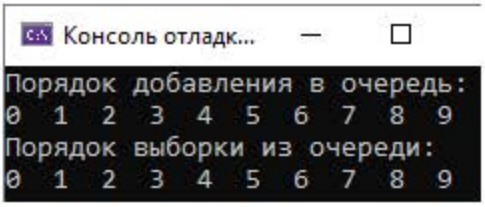


Рис. 21. Пример использования контейнерного класса queue

6.4. КОНТЕЙНЕРНЫЙ КЛАСС DEQUE

6.4.1. Назначение контейнерного класса deque

Деком называется структура данных, которая позволяет добавлять элементы и в конец, и в начало, а также удалять элементы из конца и из начала. Все эти операции реализованы за O(1).

Контейнер deque очень похож на контейнер vector. Так же как и vector, deque является динамическим массивом. Разница между ними состоит лишь в том, что в деках динамический массив открыт с двух сторон. Это и позволяет очень быстро добавлять новые элементы как в конец, так и в начало контейнера. В векторах элементы можно добавлять лишь в конец массива.

Для использования необходимо подключить библиотеку «deque»:

#include < deque >

6.4.2. Методы контейнерного класса deque

Для контейнерного класса «deque» определены следующие методы.

Для получения элементов дека можно использовать операцию [] и ряд функций:

- [index] — получает элемент по индексу;
- at(index) — возвращает элемент по индексу;
- front() — возвращает первый элемент;
- back() — возвращает последний элемент.

Чтобы добавить элементы в deque, можно применять следующие функции:

- push_back(val) — добавляет значение val в конец дека;
- push_front(val) — добавляет значение val в начало дека;

- emplace_back(val) — добавляет значение val в конец дека;
- emplace_front(val) — добавляет значение val в начало дека;
- emplace(pos, val) — вставляет элемент val на позицию, на которую указывает итератор pos. Возвращает итератор на добавленный элемент;
- insert(pos, val) — вставляет элемент val на позицию, на которую указывает итератор pos, аналогично функции emplace(). Возвращает итератор на добавленный элемент;
- insert(pos, n, val) — вставляет n элементов val, начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если n = 0, то возвращается итератор pos;
- insert(pos, begin, end) — вставляет, начиная с позиции, на которую указывают итератор pos, элементы из другого контейнера из диапазона между итераторами begin и end. Возвращает итератор на первый добавленный элемент. Если между итераторами begin и end нет элементов, то возвращается итератор pos;
- insert(pos, values) — вставляет список значений values, начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если values не содержит элементов, то возвращается итератор pos.

6.4.3. Пример использования контейнерного класса deque

В качестве примера использования контейнерного класса deque и в целях отражения отличия дека от очереди и стека рассмотрим тот же самый код, что и в разделах 6.2.3 и 6.3.3, но с демонстрацией работы с последовательностью как с начала, как и с конца.

```
#include <iostream>
#include <deque>

int main()
{
  setlocale(LC_ALL, "Russian");
  std::deque<int> num;
  std::cout << "Порядок добавления в дек: " << std::endl;
  for (int i = 0; i < 10; i=i+2)
  {
```

```
        num.push_back(i);
        num.push_front(i+1);
    }
    for (int n : num)
        std::cout << n << "  ";
    std::cout << std::endl;

    std::cout << "Вывод значений дека в прямом порядке: " << std::endl;
    for (unsigned i{}; i < num.size(); i++)
        std::cout << num[i] << "  ";
    std::cout << std::endl;

    std::cout << "Вывод значений дека в обратном порядке: " << std::endl;
    for (auto iter = num.end(); iter != num.begin(); iter--)
        std::cout << *(iter-1) << "  ";
    std::cout << std::endl;
}
```

Как следует из кода программы, формирование дека осуществляется поочередно с конца (метод push_back()) и с начала (метод push_front()) дека. Кроме того, пример демонстрирует три различных варианта вывода значений последовательности. Результат работы программы приведен на рис. 22.

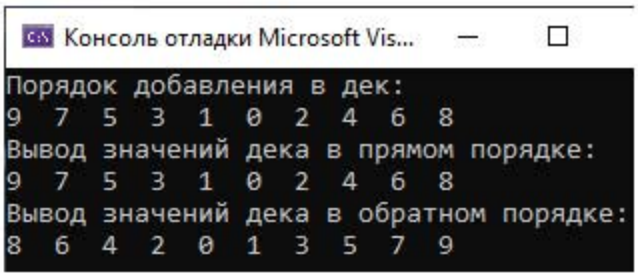


Рис. 22. Пример использования контейнерного класса deque

6.5. КОНТЕЙНЕРНЫЙ КЛАСС PRIORITY_QUEUE

6.5.1. Назначение контейнерного класса priority_queue

В языке программирования C++ priority_queue представляет собой контейнер, реализующий структуру данных «очередь приоритетов», который, как и стандартная очередь, работает по принципу FIFO. Контейнер обеспечивает по-

стоянное время поиска наибольшего (по умолчанию) элемента за счет логарифмической вставки и извлечения. При добавлении элементов в очередь приоритетов применяется функция компаратора, которая сравнивает добавляемые элементы и располагает их в очереди в определенном порядке. По умолчанию для сравнения данных применяется функция, которая располагает первыми элементами, которые условно «больше».

Таким образом, главное отличие между priority_queue и queue состоит в том, что в priority_queue наибольшее значение перемещается в начало очереди. Внутреннее различие в том, что по умолчанию классом, лежащим в основе priority_queue, является vector. Операцию сравнения, используемую для определения того, что должно находиться в голове очереди, можно изменить, передавая необязательный аргумент конструктору.

Для использования этого контейнерного класса необходимо подключить библиотеку «queue»:

```
#include <queue>
```

6.5.2. Методы контейнерного класса priority_queue

Для контейнерного класса «priority_queue» определены следующие методы:

- top() — доступ к элементу в вершине очереди;
- empty() — возвращает true, если очередь пуста;
- size() — количество элементов в очереди;
- push() — вставляет элемент с сортировкой очереди;
- emplace() — создает элемент на месте с сортировкой очереди;
- pop() — удаляет элемент в вершине очереди.

6.5.3. Пример использования контейнерного класса priority_queue

В качестве примера использования контейнерного класса priority_queue рассмотрим программу инициализации очереди с приоритетами случайными значениями с последующим выводом значений очереди в порядке убывания приоритетов.

```
#include <iostream>
#include <queue>

int main()
{
   setlocale(LC_ALL, "Russian");
   std::priority_queue<int> num;
   std::cout << "Порядок добавления в очередь: " << std::endl;
   for (int i = 0; i < 10; i++)
   {
      int a = rand() % 100;
      num.push(a);
      std::cout << a << "  ";
   }
   std::cout << std::endl;

   std::cout << "Вывод значений очереди в порядке убывания приоритетов: " << std::endl;

   while (!num.empty())
   {
      std::cout << num.top() << "  ";
      num.pop();
   }
   std::cout << std::endl;
}
```

Первый цикл программы выполняет заполнение очереди с приоритетом случайными значениями в диапазоне от 0 до 99. При этом метод push() сразу сортирует получаемую последовательность согласно значениям приоритета. При выводе значений очереди используется цикл while с методом empty(), проверяющим наличие элементов в очереди. Результат работы программы приведен на рис. 23.

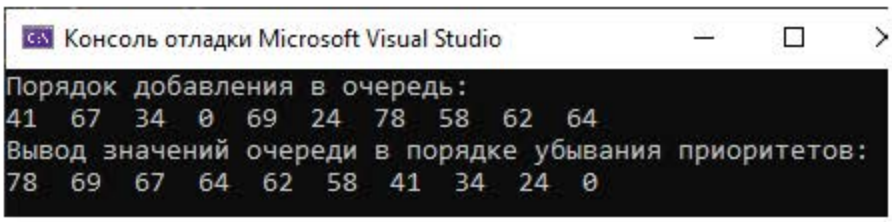


Рис. 23. Пример использования контейнерного класса priority_queue

6.6. КОНТЕЙНЕРНЫЙ КЛАСС MAP

6.6.1. Назначение контейнерного класса map

В языке программирования C++ map представляет собой контейнер, который используется для хранения и извлечения данных из коллекции, в которой каждый элемент является парой, состоящей из значения данных и ключа сортировки. Значение ключа уникально и применяется для автоматической сортировки данных. Примером такого контейнера может служить словарь, где каждому слову ставятся в соответствие его перевод или объяснение. Поэтому такие структуры еще называют словарями.

Значение элемента в паре можно изменять напрямую. Значение ключа является константой и не может быть изменено. Для изменения значения ключа, связанного со старым элементом, необходимо сначала его удалить, а затем вставить новое значения ключа для нового элемента.

Для использования этого контейнерного класса необходимо подключить библиотеку «map»:

#include <map>

6.6.2. Методы контейнерного класса map

Для контейнерного класса «map» определены следующие методы:

– at() — находит элемент с указанным значением ключа;

– contains() — проверяет, есть ли элемент с указанным ключом в элементе map;

– emplace() — вставляет элемент в объект map;

– equal_range() — возвращает пару итераторов. Первый итератор в паре указывает на первый элемент в map с ключом, который больше указанного ключа. Второй итератор в паре указывает на первый элемент в map с ключом, который больше или равен данному ключу;

– find() — возвращает итератор, указывающий на расположение элемента в map элементе с ключом, равным указанному ключу;

– insert() — вставляет элемент или диапазон элементов в map в заданной позиции;

– lower_bound() — возвращает итератор первому элементу в map, который имеет значение ключа, равное или больше заданного ключа.

6.6.3. Пример использования контейнерного класса map

В качестве примера использования контейнерного класса map рассмотрим программу создания и обработки словаря со списком сотрудников. Каждая запись списка состоит из кода или табельного номера сотрудника и его фамилии.

```
#include <iostream>
#include <map>

int main()
{
   setlocale(LC_ALL, "Russian");
   std::map<unsigned, std::string> sotrudnik;

   sotrudnik[1000] = "Иванов";
   sotrudnik[1001] = "Петров";
   sotrudnik[1006] = "Сидоров";
   sotrudnik[1010] = "Абрамов";
   sotrudnik[1100] = "Викторов";
   std::cout << "Исходный словарь: " << std::endl;
   for (const auto& element : sotrudnik)
      std::cout << element.first << "\t" << element.second << std::endl;
   sotrudnik.erase(1006);
   std::cout << std::endl;
   std::cout << "Словарь после удаления пары с ключем 1006: " << std::endl;
   for (const auto& element : sotrudnik)
      std::cout << element.first << "\t" << element.second << std::endl;
   std::cout << std::endl;
   std::cout << "Результаты поиска ключа 1010: " << std::endl;
   std::cout << "1010\t" << sotrudnik.count(1010) << std::endl;
   std::cout << std::endl;
   std::cout << "Результаты поиска ключа 1006: " << std::endl;
   std::cout << "1006\t" << sotrudnik.count(1006) << std::endl;
   std::cout << std::endl;
   std::cout << "Добавление элемента с ключем 1009: " << std::endl;
   sotrudnik.insert(std::make_pair(1009, "Мухин"));
   for (const auto& element : sotrudnik)
      std::cout << element.first << "\t" << element.second << std::endl;
}
```

Как следует из кода программы, сначала создается словарь из пяти пар элементов. Далее из словаря удаляется элемент с ключом 1006. В программе также демонстрируются удачный (ключ 1010) и неудачный (ключ 1006) поиски элементов в словаре. И, наконец, в программе демонстрируется операция добавления элемента с ключом 1009 в словарь. Результат работы программы приведен на рис. 24.

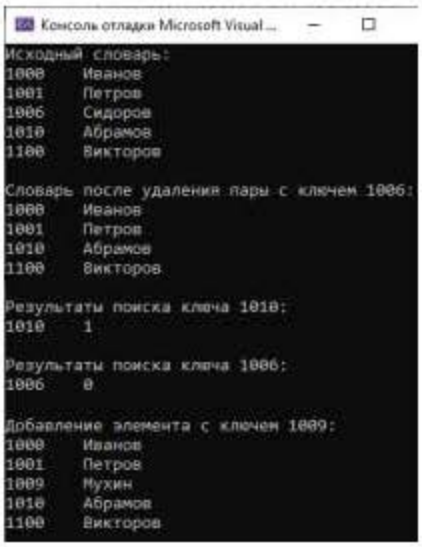


Рис. 24. Пример использования контейнерного класса map

6.7. КОНТЕЙНЕРНЫЙ КЛАСС BITSET

6.7.1. Назначение контейнерного класса bitset

В языке программирования C++ bitset — это специальный класс-контейнер, который предназначен для хранения битовых значений, т. е. элементы этого контейнера могут иметь значения 0 или 1, истина или ложь.

Класс bitset очень похож на обычный массив, но, в отличие от массива, под каждый элемент объекта типа bitset отводится всего один бит. Это позволяет использовать память в восемь раз меньше, чем наименьший элементарный тип данных в C++ — char.

Каждый элемент (каждый бит) может быть доступен по отдельности: например, у нас есть объект типа bitset с именем mybitset, необходимо получить доступ к четвертому биту. Для этого после имени массива в квадратных ско-

бочках указывается индекс бита, как если бы мы хотели получить доступ к элементу массива — mybitset [3].

Для использования этого контейнерного класса необходимо подключить библиотеку «bitset»:

#include < bitset >

6.7.2. Методы контейнерного класса bitset

Для контейнерного класса «bitset» определены следующие методы:

– set() — метод устанавливает биты объекта bitset в 1. Версия без параметров инициализирует все биты объекта bitset единицами. Параметризованная версия функции сохраняет новое значение, указанное в параметре val, бита, чей индекс передается через параметр position;

– reset() — метод выполняет сброс битов, то есть инициализирует их нулевыми значениями. Версия метода без параметров сбрасывает все биты в объекте bitset (устанавливает все биты в 0). Параметризованная версия устанавливает бит в 0, индекс которого передается через параметр position;

– flip() — метод преобразует битовую маску в обратный код. То есть все 1-цы в битовой маске заменяет на 0, а все нули — на 1. Версия flip без параметров переворачивает все биты в битовой маске, т. е. меняет все 0-ли на 1-цы и все 1-цы на 0-ли. Функция flip с параметром инвертирует только один бит, позиция которого передается через параметр pos;

– to_string() — метод преобразует объект типа bitset в объект-строку, представляющий преобразуемый объект bitset в виде последовательности нулей и единиц;

– count() — метод возвращает количество единичных битов (то есть битов, имеющих значение 1) объекта bitset;

– size() — метод возвращает количество битов, которые содержатся в объекте типа bitset;

– test() — метод возвращает значение бита, чей индекс передается через параметр position;

– any() — метод проверяет, есть ли в объекте типа bitset хотя бы один единичный бит, и возвращает true, если единичный бит найден;

– none() — метод проверяет, есть ли в объекте типа bitset хотя бы один нулевой бит, и возвращает true, если нулевой бит найден.

6.7.3. Пример использования контейнерного класса bitset

В качестве примера использования контейнерного класса bitset рассмотрим программу тестирования основных методов данного контейнерного класса.

```
#include <iostream>
#include <bitset>

int main()
{
   setlocale(LC_ALL, "Russian");
   std::bitset<16> bit;
   std::cout << std::endl << "Исходные биты: " << bit.to_string() << ", размер=" << sizeof(bit) << "
байт";
   bit.set(10);
   std::cout << std::endl << "С измененным 10-м битом: " << bit.to_string() ;
   bit.set();
   std::cout << std::endl << "Все биты в 1: " << bit.to_string();
   bit.reset();
   std::cout << std::endl << "Все биты в 0: " << bit.to_string();
   bit[5] = 1;
   bit[7] = 1;
   bit[9] = 1;
   bit[11] = 1;
   std::cout << std::endl << "5, 7, 9, 11 биты в 1: " << bit.to_string();
   bit.flip();
   std::cout << std::endl << "После инверсии: " << bit.to_string();
   std::cout << std::endl << "Количество единиц: " << bit.count();
   std::cout << std::endl << "Есть ли единицы?: " << bit.any();
}
```

Как следует из кода программы, сначала создается объект типа bitset из 16 битов. Далее с помощью метода set() изменяем 10-й бит на 1. С помощью того же метода set(), но без параметров, все биты устанавливаются в 1. Далее с помощью метода reset() все биты устанавливаются в 0. Далее демонстрируется способ обращения к отдельным битам объекта типа bitset по индексам. Таким образом изменяются на единицу 5-й, 7-й, 9-й и 11-й биты. И в конце программы с помощью метода count() определяется количество единиц, а с помощью метода any() — их наличие. Результат работы программы приведен на рис. 25.

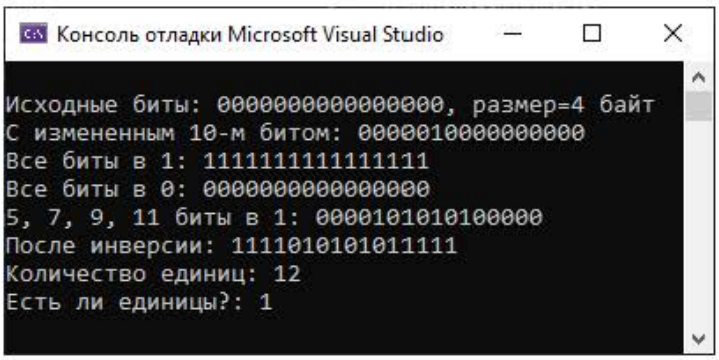


Рис. 25. Пример использования контейнерного класса bitset

6.8. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ № 6 СРАВНЕНИЕ ПОДХОДОВ К РЕАЛИЗАЦИИ СТРУКТУРЫ ДАННЫХ «МНОЖЕСТВО»

Для реализованных в лабораторных работах 1-5 (односвязный список, класс «Список», контейнерный класс List, контейнерный класс Set) операций над множествами выполнить сравнение времени выполнения следующих операций:

– создание множества из N элементов ($N \geq 1000$, задается в диалоге). Диапазон значений множеств — от 0 до $10 \cdot N$;

– мощность множества;

– подмножество $A \subset A$;

– подмножество $B \subset A$;

– равенство $A = A$;

– равенство $B = A$;

– объединение множеств A и B;

– пересечение множеств A и B;

– разность A-B;

– разность B-A;

– симметричная разность A и B.

В консоль вывести данные в табличном виде. Подобрать N таким образом, чтобы бóльшая часть значений таблицы была бы ненулевой.

7. ДЕРЕВЬЯ И ИХ СВОЙСТВА

7.1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

7.1.1. Деревья. Основные понятия и определения

Определим **дерево** с базовым типом T — это либо

1) пустое дерево; либо

2) некоторая вершина типа T с конечным числом связанных с ней отдельных деревьев с базовым типом T, называемых **поддеревьями**.

Дадим второе, нерекурсивное, определение дерева, которое более полно отражает его основные свойства.

Определение: Деревом называется конечное множество, состоящее из одного или более элементов, называемых узлами, таких, что:

– между узлами имеет место отношение типа «исходный-порожденный»;

– есть только один узел, не имеющий исходного. Он называется **корнем**;

– все узлы, за исключением корня, имеют только один исходный; каждый узел может иметь несколько порожденных;

– отношение «исходный-порожденный» действует только в одном направлении, т. е. ни один потомок некоторого узла не может стать для него предком.

Отсюда видно, что **список** есть дерево, в котором каждая вершина имеет не более одного поддерева, поэтому список называют **вырожденным деревом**.

Если в дереве между порожденными узлами, имеющими общий исходный узел, считается существенным их порядок, то дерево называется **упорядоченным**. В задачах поиска почти всегда рассматриваются упорядоченные деревья.

Конечное множество непересекающихся деревьев называется **лесом**.

Элемент типа T называется **узлом** дерева. Связи между узлами дерева называются **ветвями**. Дерево, являющееся частью другого дерева, называется **поддеревом**. Самый верхний узел называется **корнем**.

Вершина *y*, находящаяся непосредственно ниже вершины *x*, называется **непосредственным потомком** *x*. И наоборот, вершину x называют **непосредственным предком** *y*.

Если A — корень бинарного дерева и B — корень его левого или правого поддерева, то говорят, что A — **отец** B, а B — левый или правый **сын** A.

Если каждый узел бинарного дерева, не являющийся листом, имеет непустые правые и левые поддеревья, то дерево называется **строго бинарным деревом.**

Свойство 1. Строго бинарное дерево с n листами всегда содержит 2^n–1 узлов.

Максимальный уровень какой-либо из вершин дерева называется его **глубиной,** или **высотой**.

Если элемент не имеет потомков, то его называют **терминальной вершиной,** или листом. Нетерминальная вершина называется **внутренней**.

Число непосредственных потомков внутренней вершины называется ее **степенью**. Максимальная степень всех вершин есть **степень дерева**.

Число ветвей или ребер, которые нужно пройти от корня к вершине x, называется **длиной пути** к x. Корень имеет путь длины 0, его прямые потомки — пути длины 1 и т. д.

7.1.2. Двоичные (бинарные) деревья

Очень интересными с точки зрения программирования являются упорядоченные деревья второй степени. Их называют **двоичными (или бинарными) деревьями**.

Определим упорядоченное двоичное дерево как конечное множество элементов, которое либо пусто, либо состоит из корня с двумя двоичными деревьями, которые называются **левым** и **правым** поддеревом этого корня.

Пример бинарного дерева приведен ниже (рис. 26).

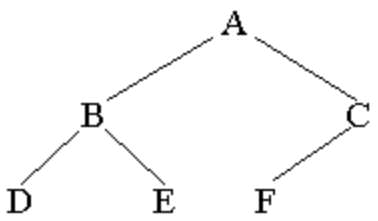


Рис. 26. Двоичное дерево

Знакомые для нас примеры двоичного дерева: генеалогическое дерево, схема теннисного турнира, арифметическое выражение с бинарными операциями, где каждому оператору соответствует вершина, а операнды — поддеревья (пример).

Рассмотрим представление бинарных деревьев. Каждая вершина такого дерева имеет следующую структуру (рис. 27).

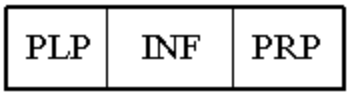


Рис. 27. Структура вершины бинарного дерева:
PLP — указатель на левое поддерево,
INF — информационное поле,
PRP — указатель на правое поддерево

Информационное поле в общем случае может представлять собой запись, содержащую несколько полей.

Введем обозначения дополнительных переменных, которые потребуются для описания соответствующих алгоритмов. Пусть

Start — указатель на начало дерева;

Work — рабочий указатель на узел дерева.

Если Work — указатель на узел дерева, то

Work->PLP — значение поля указателя на левое поддерево,

Work->INF — значение информационного поля,

Work->PRP — значение указателя на правое поддерево.

Если у текущего узла нет левого или правого поддерева, то в поле соответствующего указателя будет храниться значение пустого указателя, т. е. 0. Если дерево пусто, то Start = 0.

7.1.3. Обход бинарного дерева

Пусть необходимо выполнить некоторую операцию O над всеми элементами дерева. Эту операцию можно выполнить в рамках более общей задачи — задачи **«посещения»** всех вершин дерева, или, как ее обычно называют, **«обхода»** дерева. **Обход двоичного дерева** — это процесс, при котором каждый узел посещается точно только один раз.

Обход дерева обычно осуществляется в некотором порядке, поэтому можно считать, что дерево линейно упорядочено. Существуют три принципа упорядочения, естественно вытекающих из структуры дерева: **сверху вниз, слева направо, снизу вверх**.

Для работы с деревьями чаще всего используют рекурсивные алгоритмы. Использование рекурсивных алгоритмов менее эффективно, поскольку многократный вызов программы расходует системные ресурсы. Тем не менее в данном случае использование рекурсивных функций является оправданным, поскольку нерекурсивные алгоритмы для работы с деревьями гораздо сложнее и для написания, и для восприятия кода программы.

Рассмотрим схемы обхода бинарного дерева (рис. 28).

При обходе дерева его узлы посещаются в определенном порядке и могут считаться расположенными линейно. Можно говорить о переходе к следующему узлу, имея в виду некоторое упорядочение.

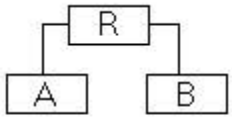


Рис. 28. Схемы обхода

Существуют три принципа упорядочения, которые естественно вытекают из структуры представленного на рисунке дерева.

Сверху вниз: R, A, B (посетить корень, обойти левое поддерево, обойти правое поддерево).

Слева направо: A, R, B (обойти левое поддерево, посетить корень, обойти правое поддерево).

Снизу вверх: A, B, R (обойти левое поддерево, обойти правое поддерево, посетить корень).

Рассмотрим более подробно алгоритмы этих методов обхода.

Алгоритм обхода методом сверху вниз (в прямом порядке, в глубину):

1. Обработка или просмотр корня дерева или поддерева.
2. Обход методом сверху вниз левого поддерева обработанного корня.
3. Обход методом сверху вниз правого поддерева обработанного корня.

Процедура WN (проход дерева сверху вниз):

```
Procedure WN (Work)
начало_процедуры
Если Work<>0 то
Печать Work->INF
WN (Work->PLP)
WN (Work->PRP)
конец_если
конец_процедуры
```

Алгоритм обхода методом слева направо (симметричный):

1. Обход методом слева направо левого поддерева.
2. Обработка или просмотр корня дерева или поддерева.
3. Обход методом слева направо правого поддерева.

Процедура LP (проход дерева слева направо):

```
Procedure LP (Work)
начало_процедуры
Если Work<>0 то
LP (Work->PLP)
Печать Work->INF
LP (Work->PRP)
конец_если
конец_процедуры
```

Алгоритм обхода методом снизу вверх (в обратном порядке):

1. Обход методом снизу вверх левого поддерева.
2. Обход методом снизу вверх правого поддерева.
3. Обработка или просмотр корня дерева или поддерева.

Процедура NW (проход дерева снизу вверх):

```
Procedure NW (Work)
начало_процедуры
Если Work<>0 то
NW (Work->PLP)
NW (Work->PRP)
Печать Work->INF
    конец_если
конец_процедуры
```

На первый взгляд эти три алгоритма отличаются только порядком следования шагов. Это действительно так. Но следует заметить, что эти алгоритмы **рекурсивные**. Проход левого и правого поддеревьев в каждом алгоритме осуществляется, во-первых, своим методом, а во-вторых, для каждого из поддеревьев необходимо последовательно выполнить 1-й, 2-й и 3-й шаги алгоритма.

Рассмотрим работу этих алгоритмов на примере (рис. 29).

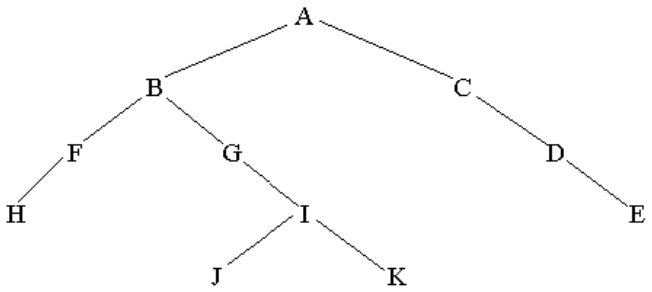


Рис. 29. Бинарное дерево

Обход дерева методом сверху вниз дает результат ABFHGIJKCDE, слева направо — HFBGJIKACDE, снизу вверх — HFJKIGBEDCA.

Если рассматривать задачи, требующие сплошного обхода дерева, то для части из них порядок обхода в целом неважен, например, подсчет числа узлов дерева, числа листьев / не листьев, элементов, обладающих заданной информацией, и т. д. Однако такая задача, как уничтожение бинарного дерева с освобождением памяти, требует использования только обхода снизу вверх (в обратном порядке).

Нерекурсивный алгоритм обхода дерева методом «сверху вниз»

Пусть T — указатель на бинарное дерево; A — стек, в который заносятся адреса еще не пройденных вершин; TOP — вершина стека; P — рабочая переменная.

1. Начальная установка: TOP:=0; P:=T.
2. Если P=nil, то перейти на 4. {конец ветви}
3. Вывести P^.info. Вершину заносим в стек: TOP:=TOP+1; A[TOP]:=P; шаг по левой ветви: P:=P^.llink; перейти на 2.
4. Если TOP=0, то КОНЕЦ.
5. Достаем вершину из стека: P:=A[TOP]; TOP:=TOP-1; шаг по правой связи: P:=P^.rlink; перейти на 2.

7.2. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ № 7
РАЗРАБОТКА КЛАССА «БИНАРНОЕ ДЕРЕВО ПОИСКА»

Разработать класс «Бинарное дерево поиска» на основе любого из контейнерных классов.

Структура программы должна включать:

– заголовочный файл;

– файл реализации — Tree_Lab7_<Фамилия>.cpp;

– главную программу Lab_7<Фамилия>.cpp.

Класс «Бинарное дерево поиска» должен включать следующие методы:

– создание пустого дерева;

– проверка пустоты;

– добавление нового узла в дерево;

– создание дерева (параметры: интервал допустимых значений, количество элементов);

– обход дерева сверху вниз (вывод результата в виде стоки):

– обход дерева слева направо (вывод результата в виде стоки);

– обход дерева снизу вверх (вывод результата в виде стоки);

– удаление дерева (с помощью обхода снизу вверх).

Пишем в консольном приложении на C++.

В основной программе протестировать все вышеперечисленные методы.

8. КУРСОВАЯ РАБОТА

8.1. ПРИМЕРНЫЕ ТЕМЫ КУРСОВОЙ РАБОТЫ

1. Реализация структуры данных «Множество» с использованием контейнерных классов priority_queue и queue.

2. Реализация структуры данных «Множество» с использованием контейнерных классов unordered_multiset и queue.

3. Реализация структуры данных «Множество» с использованием контейнерных классов forward_list и queue.

4. Реализация структуры данных «Множество» с использованием контейнерных классов forward_list и queue.

5. Реализация структуры данных «Множество» с использованием контейнерных классов unordered_map и queue.

6. Реализация структуры данных «Множество» с использованием контейнерных классов unordered_set и queue.

7. Реализация структуры данных «Множество» с использованием контейнерных классов vector и forward_list.

8. Реализация структуры данных «Множество» с использованием контейнерных классов multiset и forward_list.

9. Реализация структуры данных «Множество» с использованием контейнерных классов unordered_multimap и forward_list.

10. Реализация структуры данных «Множество» с использованием контейнерных классов stack и forward_list.

11. Реализация структуры данных «Множество» с использованием контейнерных классов deque и forward_list.

12. Реализация структуры данных «Множество» с использованием контейнерных классов multimap и forward_list.

13. Реализация структуры данных «Множество» с использованием контейнерных классов string и map.

14. Реализация структуры данных «Множество» с использованием контейнерных классов array и map.

15. Реализация структуры данных «Множество» с использованием контейнерных классов bitset и map.

16. Реализация структуры данных «Множество» с использованием контейнерных классов unordered_multiset и map.

17. Реализация структуры данных «Множество» с использованием контейнерных классов priority_queue и map.

18. Реализация структуры данных «Множество» с использованием контейнерных классов unordered_map и map.

19. Реализация структуры данных «Множество» с использованием контейнерных классов unordered_set и stack.

20. Реализация структуры данных «Множество» с использованием контейнерных классов vector и stack.

21. Реализация структуры данных «Множество» с использованием контейнерных классов multiset и stack.

22. Реализация структуры данных «Множество» с использованием контейнерных классов unordered_multimap и stack.

23. Реализация структуры данных «Множество» с использованием контейнерных классов multimap и stack.

24. Реализация структуры данных «Множество» с использованием контейнерных классов string и stack.

25. Реализация структуры данных «Множество» с использованием контейнерных классов array и deque.

26. Реализация структуры данных «Множество» с использованием контейнерных классов bitset и deque.

27. Реализация структуры данных «Множество» с использованием контейнерных классов unordered_multiset и deque.

28. Реализация структуры данных «Множество» с использованием контейнерных классов priority_queue и deque.

29. Реализация структуры данных «Множество» с использованием контейнерных классов unordered_map и deque.

30. Реализация структуры данных «Множество» с использованием контейнерных классов unordered_set и deque.

8.2. СТРУКТУРА КУРСОВОЙ РАБОТЫ

Титульный лист

Задание

Реферат

Содержание

Введение (около 2-х страниц)

– актуальность рассматриваемой темы

– цель разрабатываемой программы

– перечень подзадач, которые необходимо решить для достижения поставленной цели

– применяемые инструментальные средства

1. Постановка задачи и анализ предметной области
   1.1. Основные понятия и определения
   1.2. Постановка задачи
   1.3. Анализ требований
      1.3.1. Требования к интерфейсу пользователя
      1.3.2. Требования к структуре данных
      1.3.3. Требования к программным средствам
2. Проектирование программы
   2.1. Модель интерфейса
   2.2. Проектирование структур данных
   2.3. Структура программного обеспечения
3 Реализация программы
   3.1. Кодирование
   3.2. Диаграмма компонентов
4 Тестирование программы
   4.1 Функциональное тестирование
   4.2 Тестирование приложения
5. Анализ результатов

Заключение

Список использованных источников

Приложения

8.3. РАЗДЕЛЫ КУРСОВОЙ РАБОТЫ

8.3.1. Задание на раздел 1

1. Постановка задачи и анализ предметной области

1.1. Основные понятия и определения

Дать основные понятия и операции теории множеств (рассмотреть все операции из задания). Рассмотреть контейнерные классы List, Set и ваши из курсовой (назначение, основные сведения — один или два абзаца для каждой структуры данных).

1.2. Постановка задачи

Ваше задание на курсовую работу.

1.3. Анализ требований

1.3.1. Требования к интерфейсу пользователя

Текст данного подраздела должен основываться на том, что приложение должно позволять вводить мощность каждого из множеств, а также выводить время выполнения различными методами каждой из перечисленных в задании операций в виде таблицы. После перечисления возможностей интерфейса должен быть примерно такой текст: «Интерфейс приложения представлен в форме консольного приложения. Вывод временных характеристик осуществляется в таблицу, заголовки столбцов которой — структуры данных, а заголовки строк — наименования основных операций над ними».

1.3.2. Требования к структурам данных

В курсовой работе используются следующие структуры данных: (перечислить). Далее должно следовать описание каждой из структур данных: односвязный список (объявление, основные операции), класс «Список» (объявление, основные методы), контейнерные классы List, Set и ваши (объявление, основные методы). **ВНИМАНИЕ!!!** В этом подразделе код только для объявления. Методы в виде таблицы со столбцами: название метода, назначение.

1.3.3. Требования к программным средствам

Описать, какие функции должна выполнять разрабатываемая программа (основываясь на задании); исходя из функций разработать **диаграмму вариантов использования**, **привести описание спецификации прецедента** (3-4 таблицы).

Основные варианты использования, связанные с актером: **Ввести мощность множеств, Вычислить время операций, Вывести результат**. Ну а эти варианты использования уже раскрываете другими вариантами использования при необходимости. В этих вариантах использования обязательно должны быть перечислены используемые в курсовой структуры данных и выполняемые операции.

8.3.2. Задание на раздел 2

2. Проектирование программы

2.1. Модель интерфейса

Привести скриншоты окон консольного приложения: окно для ввода мощности каждого из множеств, окно/окна для вывода результатов. Вы должны сделать скриншот каждого окна, в графическом редакторе каждой значимой компоненте окна присвоить номер, вставить в курсовую работу этот рисунок и прокомментировать каждый номер на рисунке. Например (рис. 30):

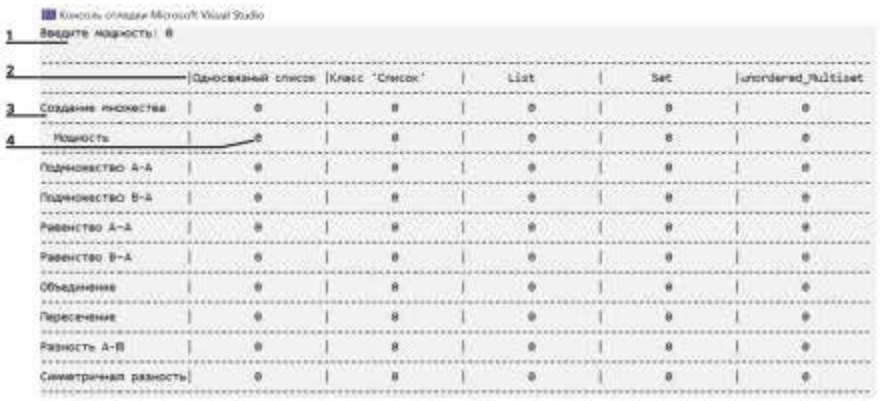


Рис. 30. Пример окна консольного приложения

2.2. Проектирование структур данных

В курсовой работе используются следующие структуры данных (перечисление). Ниже должно быть приведено описание структуры данных: «Односвязный список» и функции «Название». Далее следуют код описания односвязного списка и код функции, например пересечения. Далее пояснительный текст к коду как описания, так и функции. И так для каждой используемой в курсовой работе структуры данных. Для каждой структуры функция/метод должны быть разными.

2.3. Структура программного обеспечения

Структура программного обеспечения описана диаграммой классов, представленной на рисунке 999[1]. Ниже должно быть представлено краткое описание спецификаций каждого класса (Спецификация класса — это краткое описание класса, которое включает: название класса, назначение класса, описание членов класса с указанием наименования и типа, описание методов класса с указанием наименования и параметров.)

8.3.3. Задание на раздел 3

3. Реализация программы

3.1. Кодирование

В ходе выполнения курсовой работы было разработано приложение «Название», код которого приведен в приложении А.

Для алгоритма операции, например пересечения (у вас может быть алгоритм любой другой операции, но состоящей из достаточно большого числа строк), была разработана диаграмма деятельности (рисунок 999).

ВНИМАНИЕ!!! Диаграмма деятельности — это не код соответствующей функции, а слова с минимальным использованием переменных. Код должен создаваться на основе диаграмм деятельности, а не наоборот.

Под рисунком комментарии к диаграмме, например:

«На диаграмме используются следующие переменные:

– Start — указатель на голову односвязного списка;

– Uk — рабочий указатель…»

и т. п.

3.2. Диаграмма компонентов

В процессе выполнения курсовой работы была составлена диаграмма компонентов, которая отображает разбиение программной системы на структурные компоненты и связи между компонентами (рисунок 999).

За рисунком необходимо прокомментировать назначение каждого компонента диаграммы (можно в виде таблицы). Например:

CourseWork.exe — исполняемый файл;

CourseWork.h — файл заголовка и т. д.

[1] Дан нереальный номер рисунка, чтобы студенты поставили свой реальный.

8.3.4. Задание на раздел 4

4. Тестирование программы

4.1. Функциональное тестирование

На половину странички теория о функциональном (например) тестировании.

4.2. Тестирование приложения

Начинаем так: В курсовой работе было выполнено функциональное (например) тестирование разработанного программного обеспечения. Результаты тестирования приведены в таблице 1.

Таблица 1. Результаты тестирования

Состав теста	**Ожидаемый результат**	**Наблюдаемый результат**

В наблюдаемых результатах обязательно ссылка на рисунок со скриншотом этого результата. Сами скриншоты приводите сразу за таблицей. Перед рисунками обязательно пояснительный текст. (Например: «Ниже, на рисунках 991-999, приведены скриншоты, отражающие результаты работы программы в процессе тестирования».)

Следует учесть, что все рисунки со скриншотами должны иметь номер и название, причем **названия рисунков не должны совпадать**.

Ну и в конце завершающие слова: «В ходе выполнения тестирования несовпадения ожидаемого и наблюдаемого результатов не выявлены. Следовательно, можно сделать вывод, что программа работает корректно».

Необходимо привести не менее 12-ти тестов. Среди 12-ти тестов как минимум три должны быть на нестандартные, граничные ситуации: задание отрицательного числа элементов в множестве, выполнение, например, пересечения, когда одно или оба множества пустые, и т. п.

8.3.5. Задание на раздел 5

1. Это должна быть полностью оформленная курсовая работа.

2. Введение строго по плану

Введение (около 2-х страниц):

– актуальность рассматриваемой темы;

– цель разрабатываемой программы;

– перечень подзадач, которые необходимо решить для достижения поставленной цели;

– применяемые инструментальные средства.

3. В основном тексте по всей работе проверить:

a. Выравнивание по ширине.

b. Красная строка.

c. Выход рисунков за границы листа.

d. Наличие висячих заголовков.

e. Отступы между подразделами.

f. Наличие интернетовских символов типа неразрывного пробела, перевода строки и т. п. (убрать).

g. На все рисунки должны быть ссылки в тексте.

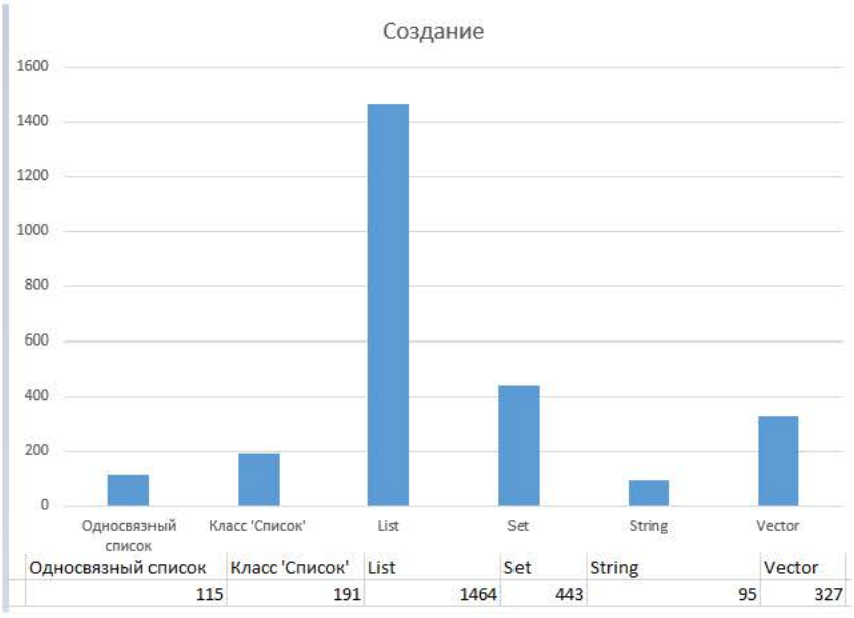
h. Все рисунки должны иметь номер и название.

i. То же самое по таблицам. В таблицах номер и название над таблицей.

4. Новый раздел 5. Анализ результатов.

Для каждой из приведенных в задании операций над множествами строим с помощью MS Excel табличное представление данных о времени выполнения операции методами различных структур данных, а также гистограмму. Для анализируемой операции подобрать такую мощность множеств, которая минимизирует число нулевых значений.

Например (рис. 31):



Односвязный список	Класс 'Список'	List	Set	String	Vector
115	191	1464	443	95	327

Рис. 31. Пример табличного и графического представления данных

После этого анализируете табличное и графическое представление результатов: какая реализация операции лучше, хуже, или они одинаковы.

В конце раздела общий анализ по всем операциям.

5. Заключение. После 5-го раздела должно быть заключение. Общая мысль такая. В курсовой работе была поставлена такая-то задача. Для ее реализации были решены следующие подзадачи (см. разделы курсовой), получены следующие результаты (по разделу 5), по которым сделаны следующие выводы (также раздел 5).

6. Список использованных источников. 5-7 источников. В тексте **ОБЯЗАТЕЛЬНО** ссылки на эти источники. Ссылки [], а не сноски (в работе[1]). Ссылки в работе идут в порядке их следования в списке. Или, наоборот, список составляете по мере появления очередной ссылки.

7. Приложение А — Листинг программы. Обязательно титульный для приложения. Далее просмотрите, чтобы весь код в приложении был черным.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Самуйлов, С.В.* Алгоритмы и структуры обработки данных : учебное пособие) / С.В. Самуйлов. — Саратов : Вузовское образование, 2016. — 132 с.

2. *Новиков Ф.А.* Дискретная математика для программистов : учебник для вузов / Ф.А. Новиков. — 3-е изд. — Санкт-Петербург : Питер, 2009. — 384 с.

3. Дискретная математика. Конспект лекций. — URL: https://studfile.net/preview/4003389/ (дата обращения: 10.07.2025).

4. *Битюцкий В.П.* Основы дискретной математики. Часть 1 : учебное пособие по дисциплине «Дискретная математика» / В.П. Битюцкий, С.С. Соколов. — Екатеринбург : ГОУ ВПО УГТУ-УПИ, 2005. — 96 с.

5. *Судоплатов С.В.* Дискретная математика : учебник / С.В. Судоплатов, Е.В. Овчинникова. — Москва : ИНФРА-М ; Новосибирск : Изд-во НГТУ, 2005.

6. *Кривцова И.Е.* Основы дискретной математики. Часть 1 : учебное пособие / И.Е. Кривцова, И.С. Лебедев, А.В. Настека. — Санкт-Петербург : Университет ИТМО, 2016. — 92 с.

7. Теория множеств. Дискретная математика. — URL: https://siblec.ru/informatika-i-vychislitelnaya-tekhnika/diskretnaya-matematika/1-teoriya-mnozhestv (дата обращения: 10.07.2025).

8. *Павловская Т.А.* C/C++. Программирование на языке высокого уровня / Т.А. Павловская. — Санкт-Петербург : Питер, 2003. — 461 с.

9. *Абрамян М.Э.* Введение в стандартную библиотеку шаблонов C++. Описание, примеры использования, учебные задачи : учебник / М.Э. Абрамян ; Южный федеральный университет. — Ростов-на-Дону ; Таганрог : Изд-во ЮФУ, 2017. — 177 с.

10. C и C++ в Visual Studio. — URL: https://learn.microsoft.com/ru-ru/cpp/overview/visual-cpp-in-visual-studio?view=msvc-170 (дата обращения: 10.07.2025).