МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. Н.И. ЛОБАЧЕВСКОГО»

Институт информационных технологий, математики и механики

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Учебно-методическое пособие

Рекомендовано методической комиссией ИИТММ для студентов ННГУ, обучающихся по направлениям подготовки

01.03.02 Прикладная математика и информатика,

02.03.02 Фундаментальная информатика и информационные технологии,

09.03.04 Программная инженерия

Нижний Новгород

2017

УДК 004.4

Лабораторный практикум. Составители:Барышева И.В., Мееров И.Б., Сысоев А.В., Шестакова Н.В. Под редакцией Гергеля В.П. Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2017. – 105с.

Рецензент: кандидат физико-математических наук, Д.Е.Шапошников

Пособие предназначено для студентов ИИТММ направлений подготовки «Прикладная математики и информатика», «Фундаментальная информатика и информационные технологии», «Программная инженерия», изучающих дисциплину «Алгоритмы и структуры данных».

В пособии даны описания 8 лабораторных работ для выполнения студентами в ходе учебной практики. Приведен необходимый теоретический материал, рассмотрены вопросы реализации основных алгоритмов, даны вопросы для самоконтроля.

УДК 004.4

СОДЕРЖАНИЕ

Введение

В учебном пособии содержится описание лабораторного практикума по основной части общего учебного курса «Алгоритмы и структуры данных», изучаемого студентами Института информационных технологий, математики и механики (ИТММ)[1] в рамках направлений подготовки «Прикладная математика и информатика», «Фундаментальная информатика и информационные технологии», «Программная инженерия».

Цель учебного курса и, соответственно, лабораторного практикума состоит в освоении моделей и методов компьютерного представления сложных математических моделей, используемых при моделировании исследуемых объектов или явлений в разных проблемно-ориентированных областях. В рамках курса и лабораторного практикума осваиваются:

- методы конструирования математических моделей и их компьютерного представления для разнообразных задач обработки данных;
- методы разработки математических структур, соответствующих сложным объектам (таблицам, текстам, чертежам и т.п.) и операций над этими структурами;
- методы распределения компьютерных ресурсов между динамически изменяемыми структурами данных и конкурирующими вычислительными процессами;
- принципы разработки масштабных программных систем, создаваемых для компьютерного представления сложных объектов и явлений;
- методы оценки вычислительной сложности разрабатываемых алгоритмов и программ как основы для создания высокопроизводительного программного обеспечения с максимально достижимыми показателями по эффективности.

Изучаемый в учебном курсе теоретический материал и получаемые при выполнении лабораторных работ практические навыки составляют важный обязательный уровень подготовки специалистов по созданию сложных проблемно-ориентированных человеко-машинных систем, автоматизирующих процесс построения и анализа сложных математических моделей объектов и явлений. Выполнение лабораторных работ направлено на получение практических навыков в области программирования и развитие компетенций в сфере профессиональной деятельности.

Излагаемый в пособии учебно-методический материал основан на опыте преподавания общего учебного курса «Алгоритмы и структуры данных» и проводимого для его поддержки лабораторного практикума на факультете вычислительной математики и кибернетики Нижегородского госуниверситета. Данный учебный курс читается на факультете ВМК с 1979 г. и был разработан проф. Р.Г. Стронгиным; дальнейшее развитие учебного курса выполнялось проф. В.П. Гергелем. Основные принципы организации лабораторного практикума и начальный состав лабораторных работ были разработаны под руководством проф. А.О. Грудзинского).

Лабораторные работы, представленные в пособии, разработаны коллективом преподавателей кафедры математического обеспечения ЭВМ факультета ВМК (ныне кафедры математического обеспечения и суперкомпьютерных технологий института ИТММ) и являются дальнейшим развитием учебно-методического комплекса [8-14]. Тематика работ охватывает широкий диапазон заданий от задач по реализации структур хранения для типовых динамических структур данных (стеки, очереди и таблицы) до

[1] Институт ИТММ создан в Нижегородском университете в 2015 г. на базе факультета вычислительной математики и кибернетики (ВМК), механико-математического факультета и НИИ прикладной математика и кибернетики. Создание института в рамках Программы повышения международной конкурентоспособности ННГУ в соответствии с Национальным проектом «5-100»

постановок работ по созданию сложных программных систем для поддержки процессов компьютерной обработки наиболее распространенных видов проблемно-ориентированных данных (тексты и графическая информация), в т.ч.

- Выполнение теоретико-множественных операций над множествами (битовые поля);
- Обработка матриц специального вида (ленточные и верхне-треугольные матрицы);
- Вычисление арифметических выражений (стеки);
- Имитационное моделирование системы обслуживания потока заданий (очереди);
- Аналитические преобразования полиномов от нескольких переменных (списки);
- Редактирование текстов (связные списки);
- Обработка геометрических объектов на ЭВМ (плексы);
- Статистическая обработка данных (таблицы);

В рамках практикума предлагается 8 лабораторных работ, расположенных в порядке возрастания сложности разрабатываемого программного обеспечения. В описании каждой лабораторной работы приведены основные теоретические положения, используемые структуры данных, алгоритмы решения задач, возможный порядок разработки. Даются вопросы и задания для самоконтроля, а также требования к результатам выполнения работы. В каждой лабораторной работе рассматриваются разные уровни сложности, что позволяет учитывать уровень подготовленности студентов и объем времени, выделяемый на учебную практику. После выполнения и сдачи преподавателю готовой программы, студент готовит отчет. Составление отчета является как формой рефлексии, позволяя студенту проанализировать процесс и итог работы, так и формой подготовки к профессиональной деятельности по составлению программной и сопроводительной документации.

Лабораторные работы 1-2 разработаны Мееровым И.Б. и Сысоевым А.В., лабораторные работы 3-8 подготовлены в результате существенной переработки предшествующих вариантов Шестаковой Н.В. План лабораторного практикума подготовлен Барышевой И.В. Общая редакция описания лабораторных работ выполнена Гергелем В.П.

Программа общего курса «Алгоритмы и структуры данных»

1. Цели и задачи курса и его место в учебном процессе

1.1. *Цель преподавания курса*

Усложнение решаемых человеком научно-технических и управленческих задач ведет к возрастанию сложности математических средств, развиваемых для анализа таких проблем. Основой для эффективного использования усложняющихся математических методов специалистами из проблемных областей является создание проблемно-ориентированных человеко-машинных систем, автоматизирующих процесс построения и анализа сложной математической модели объекта или явления (по описанию, представленному в терминах соответствующей области приложений). Проблемно-ориентированный специалист может эффективно пользоваться такой системой, не вдаваясь в вопросы программного воплощения соответствующих математических моделей и методов, как при написании программы на языке высокого уровня можно не знать способов трансляции этой программы в машинные команды. По существу, такие системы создают некоторую новую проблемно-ориентированную (виртуальную) машину для прикладного специалиста, приспособленную для удобного описания объектов проблемной области и операций над этими объектами.

Цель данного курса состоит в изучении основных способов компьютерного представления математических методов моделирования и анализа в виде таких виртуальных машин.

1.2. *Задачи изучения курса*

Изучение курса включает освоение методов компьютерного представления сложных математических моделей (отображающих проблемно-ориентированные объекты и операции над ними), обеспечивающих создание виртуальных машин, в т.ч.

- методы конструирования математических моделей и их компьютерного представления для разнообразных задач обработки данных;
- методы разработки математических структур, соответствующих сложным объектам (таблицам, текстам, чертежам и т.п.) и операций над этими структурами;
- методы распределения компьютерных ресурсов между динамически изменяемыми структурами данных и конкурирующими вычислительными процессами;
- принципы разработки масштабных программных систем, создаваемых для компьютерного представления сложных объектов и явлений;
- методы оценки вычислительной сложности разрабатываемых алгоритмов и программ как основы для создания высокопроизводительного программного обеспечения с максимально достижимыми показателями по эффективности.

1.3. *Дисциплины, освоение которых необходимо при изучении данного курса*

Курс опирается на материал вводного курса «Языки и методы программирования», изучаемого в 1-2 семестрах и направленного на освоение компьютерных систем как инструмента автоматизации исполнения алгоритмов обработки информации (общее представление о компьютерных системах, понятие алгоритма, способы разработки алгоритмов, программа на языке высокого уровня, выполнение программы, тестирование и отладка, модульное и структурное программирование, основы объектно-ориентированного подхода к разработке программ).

В курсе используются основные понятия математической логики (логические переменные и операции двоичной логики), ряд понятий алгебры (алгебраические операции, циклическая группа), теория графов (орографы и их подграфы), дискретной математики (рекурсивные описания, конечные автоматы), понятия функций и математической структуры.

2. Содержание курса

1. Структура действия и структуры данных.

↳1.1. Структуры данных.

1.1.1. Разложение действия на элементарные части (структура действия). Порождение структуры операндов структурой действия.

1.1.2. Рекурсия как средство повышения эффективности программирования и определяемая ею собственная структура операндов (векторы, матрицы и др., примеры структур).

1.1.3. Структура алгоритмов и структура данных. Связь с математическим понятием структуры. Графический образ структуры.

1.1.4. Переменные величины и схемы структур. Значения переменных структур и экземпляры схем. Элементы структуры, имена, значения. Основные и вспомогательные базисные множества и отношения в структуре.

↳1.2. Структуры хранения.

1.2.1. Структуры хранения, представляющие структуры программ.

1.2.2. Структура машинной памяти. Примеры структур хранения данных. Вектор памяти. Массивы. Адресная арифметика как средство задания отношений в структуре хранения. Структуры хранения, операции над структурами и типы.

↳1.3. Динамические структуры.

1.3.1. Переработка информации как преобразование структур данных. Преобразования, приводящие к рекурсивным отношениям исходных и результирующих структур.

1.3.2. Динамические структуры - класс структур с частичным упорядочением (по включению) структур данных, примеры динамических структур (стеки, очереди, деки).

↳1.4. Динамические структуры и структуры хранения.

1.4.1. Динамические структуры и распределение памяти; средства поддержания динамической структуры. Выражение отношений программными средствами. Пример: структура типа стека и ее структура хранения.

1.4.2. Сравнение структур хранения и хранения динамических структур.

1.4.3. Хранение динамических структур при ограниченной памяти. Степень использования памяти. Управление размещением. Пример: организация хранения очереди; введение циклических структур. Пример: хранение двух стеков. Хранение нескольких динамических структур и необходимость перераспределения памяти в процессе обработки информации.

↳1.5. Динамическое распределение памяти.

1.5.1. Статическое и динамическое распределение памяти. Управление памятью.

1.5.2. Управление памятью путем перепаковки структур хранения, представляющих отношения адресной арифметикой. Пример системы управления памятью путем перепаковки. Хранение нескольких стеков в общем массиве памяти (начальное распределение памяти; переполнение стека; оценка наличия свободной памяти; гипотеза о росте потребности в памяти; перераспределение свободной памяти; перепаковка памяти).

1.5.3. Роль гипотез о росте структур при разработке систем управления памятью. Пример использования гипотезы о сохранении тенденции роста с момента

последней перепаковки. Система управления памятью и математическая модель распределения ресурса.

↳1.6. Распределение памяти для структур хранения, представляющих основные отношения с помощью адресных указателей.

1.6.1. Представление основных отношений с помощью адресных указателей (сцепление). Задание линейных структур сцеплением (ссылки, кванты памяти; звенья; указатель структуры и признак конца). Линейный список.

1.6.2. Хранение динамических структур с использованием сцепления. Стек свободной памяти. Исключение операций перепаковки. Пример системы хранения стеков, очередей и деков.

2. Динамические структуры и конструирование математических моделей (алгебр: объекты и операции).

↳2.1. **Пример 1**: система для арифметических действий над полиномами.

2.1.1. Упорядочение мономов по степеням переменных (случай одной переменной). Представление полинома вектором коэффициентов при упорядоченных мономах. Арифметические действия над полиномами как операции над векторами (линейными структурами); рекурсия при выполнении операций.

2.1.2. Умножение полиномов и рост структур. Полином как линейная структура в стеке. Использование системы управления стеками для работы с полиномами. Недостатки представления полинома вектором коэффициентов (расход памяти и времени на хранение и обработку значительного числа нулевых коэффициентов).

↳2.2. **Пример 2**: система для арифметических действий над многочленами от нескольких переменных.

2.2.1. Исключение хранения нулевых коэффициентов. Упорядочение мономов по степеням переменных. Индексы мономов. Многочлен как линейная структура, элементы которой имеют значения, определяемые несколькими величинами.

2.2.2. Представление многочленов стеками и проблема перепаковки памяти. Представление многочленов списками. Проблема нулевых многочленов. Общее представление многочленов циклическими списками (понятие циклического списка). Список многочлена, тождественно равного нулю.

2.2.3. Алгоритм сложения многочленов, содержащий операции управления памятью, включения элементов в середину списка, исключения элемента из списка.

↳2.3. **Пример 3**: редактирование текстов.

2.3.1. Текст как линейная структура, элементами которой являются символы. Представление текста линейным списком. Текст как линейная структура, элементами которой являются слова, значения которых есть линейные структуры (последовательности символов). Выражение связи элемента и значения с помощью адресных указателей. Текст как линейная структура, элементами которой являются строки, значения которых есть линейные структуры (последовательности слов). Текст как иерархия линейных структур, образом которой является структура типа дерева, Представление структуры текста связным списком.

2.3.2. Представление текста связным списком из однотипных звеньев. Атомы. Связный список общего вида. Звено как представитель подсписка связного списка. Операция расчленения списка и объединения списков. Свойства основных операций над списком. Пример использования основных операций (выделение

списка фамилий из списка пар фамилия-имя; слияние списков имен и фамилий в список пар фамилия-имя).

2.3.3. Обработка списков (как модель обработки текстов). Обход списка; операция "первый атом". Замена атома списка другим атомом или подсписком. Копирование списка. Управление памятью при работе со связными списками (сборка мусора; необходимость маркировки занятых звеньев). Языки для обработки списков.

2.4. **Пример 4**: структуры хранения геометрических объектов (случай плоского чертежа, содержащего точки и отрезки прямых линий).

2.4.1. Наличие нескольких основных базисных множеств в структуре (точки, линии и т.п.). Представление разнотипных элементов структуры звеньями одинакового формата (использование сцепления для выражения принадлежности точек линиям). Плексы. Различия чертежа и графа представляющего его плекса.

2.4.2. Алгоритм обхода плекса. Плекс как представление выражения (операторы, операнды, значения). Вычисление выражения, представленного плексом (построение рисунка или чертежа). Плексы как представление арифметических выражений. Общее выражение, представляемое плексом.

3. Организация доступа по имени к структурам данных.

3.1. Табличная форма задания соответствия имени и адреса.

3.1.1. Имя как средство выделения и указания элемента структуры. Адрес как указатель для аппаратного доступа к звену, являющемуся машинным представлением элемента структуры. Необходимость построения вычислимого соответствия имени и адреса.

3.1.2. Понятие таблицы (ключ, тело, запись). Таблицы имен и адресов.

3.1.3. Операции над таблицей: поиск, включение и исключение записей. Таблица как динамическая структура данных.

3.2. Просмотровые (неупорядоченные) таблицы. Поиск записи по ключу (просмотр). Оценка эффективности поиска. Программная реализация.

3.3. Упорядоченные таблицы.

3.3.1. Использование упорядоченности для ускорения поиска (двоичный поиск). Оценка эффективности поиска.

3.3.2. Сортировка записей в целях упорядочения их по числовым значениям ключа (сортировка включением; ускорение сортировки - сортировка слиянием). Временная и пространственная сложность алгоритмов сортировки.

3.3.3. Включение новых записей в сортированную таблицу.

3.4. Представление таблиц в виде деревьев поиска.

3.4.1. Понятие дерева поиска. Выполнение операций поиска, включения и исключения записей. Возможность выполнения операций без перепаковки памяти.

3.4.2. Оценка эффективности выполнения операций. Анализ балансировки деревьев (возможность вырождения дерева поиска в линейный упорядоченный список). Сбалансированные и идеально сбалансированные деревья.

3.4.3. Алгоритм вставки с сохранением балансировки дерева поиска.

3.5. Таблицы с вычислимыми адресами.

3.5.1. Функции перемешивания (понятие и требования к способам построения). Возможность ситуаций относительного переполнения (коллизий) при выполнении операций вставки записей.

3.5.2. Запись и поиск в случае переполнения. Открытое перемешивание (связь с циклическими группами используемой схемы построения адресов имен). Оценка эффективности.

3.5.3. Использование линейных списков в переполняемой строке (таблицы с переменной длиной строки).

3.6. Сравнительная характеристика способов организации таблиц.

4. Представление графовых моделей на ЭВМ.

4.1. Понятие графа и выбор структуры хранения графов (матрицы смежности, множества смежных вершин, списки исходящих дуг).

4.2. Реализация структуры хранения графов.

4.3. Алгоритмы обхода графов (поиск в глубину и в ширину).

4.4. **Пример:** Поиск кратчайшего пути в графе. Алгоритм Дейкстры.

Учебный курс «Алгоритмы и структуры данных» (2 курс, 3-4 семестры)

План практических и лабораторных занятий

1 семестр

№	Тематика лекционных занятий	Тематика практических и лабораторных занятий	Лабораторный практикум
1	Введение	**Объектно-ориентированное программирование**: примеры использования	
2	Структуры данных. Структуры хранения данных	**Пример использования ООП**: Структуры хранения множеств	**ЛР 1 «Структура хранения множества»**
3	Примеры структур: множества и матрицы		
4	Основы технологии разработки ПО: этапность и системы контроля версий	**Пример использования ООП**: Матрицы	**ЛР 2 «Структуры хранения матриц специального вида»**
5	Основы технологии разработки ПО: тестирование		
6	Динамические структуры данных: стеки и очереди Очереди	**Практика использования динамических структур данных: стеки и очереди. Примеры**:	**ЛР 3 «Вычисление арифметических выражений»**

7		1. Трансляция арифметических выражений 2. Система обслуживания с приоритетной очереди заданий 3. Профилирование вычислений	**ЛР 4 «Имитационное моделирование системы обслуживания потока заданий на ЭВМ»**
8	Динамическое распределение памяти. Система N стеков.		
9		**Практика использования динамического распределения памяти.** **Примеры**: 1. Система обслуживания с несколькими очередями заданий 2. Поразрядная сортировка **Системная динамически распределяемая память**	
10	Структуры хранения с использованием указателей Списки		
11		**Практика реализации списковой памяти.** **Примеры**: 1. Реализация стеков и очередей с использованием списков 2. Разработка общего представления списков 3. Реализация двунаправленных списков	
12			
13	Пример 1: Аналитические вычисления (полиномы)		**ЛР 5 «Аналитические преобразования полиномов от нескольких переменных»**
14		**Практика использования списковой памяти.** **Примеры**: 1. Разработка расширенного набора операций над полиномами 2. Структуры хранения разреженных матриц	
15	Пример 2: Обработка текстов		
16			

		Библиотека шаблонов. Примеры использования	

2 семестр

№	Тематика лекционных занятий	Тематика практических и лабораторных занятий	Лабораторный практикум
1	Основы технологии разработки ПО: командная разработка	**Практика работы с текстами.** **Примеры**: 1. Выбор и разработка базовых операция над текстами 2. Алгоритмы обхода 3. Разработка расширенного набора операций над текстами 4. Визуализация текста 5. Разработка экранного интерфейса для редактора текста	**ЛР 6 «Редактирование текстов»**
2			
3	Пример 3: Редактирование геометрических объектов (плексы)		
4			
5	Организация доступа по имении (таблицы) 1. Просматриваемые таблицы 2. Упорядоченные таблицы 3. Таблицы на основе деревьев поиска 4. Таблицы с вычислимым входом	**Практика работы с геометрическими объектами.** **Примеры**: 1. Выбор и разработка базовых операция над плексами 2. Разработка расширенного набора операций над плексами 3. Визуализация плексов 4. Разработка экранного интерфейса для редактора чертежей	
6			**ЛР 7 «Обработка геометрических объектов на ЭВМ»**
7			

8			
9		**Расширенная практика работы с библиотекой шаблонов**	
10			
11	Графы	**Практика работы с графами.** **Примеры**: 1. Выбор и разработка базовых операция над графами 2. Разработка операций над графами	**ЛР 8 «Статистическая обработка результатов экзаменационной сессии»**
12			
13	Введение в параллельное программирование	**Практика работы с таблицами.** **Примеры**: 1. Выбор и разработка базовых операция над таблицами. 2. Разработка операций над графами. Проведение вычислительных экспериментов для сравнения разных способов организации таблиц.	
14			

Лабораторная работа №1
Структура хранения множества

Введение

Теория множеств – учение об общих свойствах множеств – преимущественно бесконечных. Явным образом понятие *множества* подверглось систематическому изучению во второй половине XIX века в работах немецкого математика Георга Кантора[2].

Влияние теории множеств на развитие современной математики очень велико. Прежде всего, теория множеств явилась фундаментом ряда новых математических дисциплин (теории функций действительного переменного, общей топологии, общей алгебры, функционального анализа и др.).

Постепенно теоретико-множественные методы находят всё большее применение и в классических частях математики. Например, в области математического анализа они широко применяются в качественной теории дифференциальных уравнений, вариационном исчислении, теории вероятностей и др.

Активное применение аппарата теории множеств в современной науке приводит к необходимости создания соответствующих программных решений. Вместе с тем лишь в отдельных языках программирования предусмотрены встроенные средства для работы с множествами (примером может служить язык Pascal в реализации фирмы Borland).

Программная реализация *множества* может выполняться различными способами (в соответствии с требованиями конкретной задачи или с общих позиций) и обычно тесно связана с использованием битовых операций в выбранном языке программирования.

Данная работа посвящена изучению одного из возможных подходов к хранению и обработке множеств.

1. Постановка учебной задачи

1.1. Основные понятия и определения

Понятие *множества*, или *совокупности*, принадлежит к числу простейших математических понятий; оно не определяется, но может быть пояснено при помощи примеров. Так, можно говорить о множестве всех книг, составляющих данную библиотеку, множестве всех точек данной линии, множестве всех решений данного уравнения.

Книги данной библиотеки, точки данной линии, решения данного уравнения являются *элементами* соответствующего множества.

Чтобы *определить* множество, достаточно *указать характеристическое свойство элементов*, т. е. такое свойство, которым обладают все элементы этого множества и только они.

[2] Кантор Георг (1845 – 1918) – немецкий математик. Разработал теорию бесконечных множеств и теорию трансфинитных чисел. Доказал несчётность множества всех действительных чисел, сформулировал (1878) общее понятие мощности множества. Ввёл понятия предельной точки, производного множества, построил пример совершенного множества, развил одну из теорий иррациональных чисел, сформулировал одну из аксиом непрерывности.

Может случиться, что данным свойством не обладает вообще ни один предмет; тогда говорят, что это свойство определяет *пустое* множество.

То, что данный предмет **X** есть элемент множества **M**, записывают так: $\mathbf{X} \in \mathbf{M}$ (читают: **X** принадлежит множеству **M**).

Если каждый элемент множества **A** является в то же время элементом множества **B**, то множество **A** называется *подмножеством*, или частью, множества **B**. Это записывают так: $\mathbf{A} \subset \mathbf{B}$ или $\mathbf{B} \supset \mathbf{A}$. Таким образом, подмножеством данного множества **B** является и само множество **B**. Пустое множество, по определению, считают подмножеством всякого множества.

Одной из определяющих характеристик множества является его *мощность*. В рамках данной работы рассматриваются множества, содержащие конечное число элементов. В этом случае понятие *мощности* определяется как *количество элементов* множества.

Для таких множеств в математике принята следующая форма записи: $A = \{a_1, a_2, \ldots, a_n\}$, где A – множество, a_i – элементы множества, n – мощность множества.

Множество всех возможных элементов называется *Универс* и обычно обозначается **U**.

1.2. Операции над множествами

Пусть заданы два множества $A = \{a_1, a_2, \ldots, a_n\}$ и $B = \{b_1, b_2, \ldots, b_m\}$. Рассмотрим следующие основные операции над множествами:

- Включение элемента в множество: $A \cup \{b\} = \{a_1, a_2, \ldots, a_n, b\}$
- Исключение элемента из множества: $A \setminus \{a_j\} = \{a_1, a_2, \ldots, a_{j-1}, a_{j+1}, \ldots, a_n\}$.
- Сумма (объединение) множеств.
- Суммой множеств A и B называется множество C, каждый элемент которого есть элемент хотя бы одного из множеств-слагаемых A и B.
- Пересечение множеств:
- Пересечением множеств A и B называется множество C, каждый элемент которого принадлежит обоим множествам A и B.
- Разность (дополнение) множеств:
- Разностью множеств A и B называется множество C, каждый элемент которого является элементом A и не является элементом B.
- Вычисление мощности множества.

1.3. Требования к лабораторной работе

В рамках лабораторной работы ставится задача создания программных средств, поддерживающих эффективное хранение множеств, удовлетворяющих указанным выше допущениям, и выполнение основных операций над множествами:

- включение элемента в множество;
- исключение элемента из множества;
- проверка наличия элемента в множестве;
- сложение множеств;
- пересечение множеств;
- разность множеств;
- копирование множества;

- вычисление мощности множества.

Программные средства должны содержать:

- класс Множество;
- тестовое приложение, демонстрирующее использование основных операций с множествами.

1.4. Условия и ограничения

Сделаем следующие основные допущения:

1. Условимся рассматривать в дальнейшем конечные (см. выше) множества, состоящие из элементов произвольного типа.
2. Элементы множества проиндексированы (каждому элементу соответствует уникальный индекс).
3. Множество индексов элементов составляет непрерывный диапазон целых значений.
4. Будем считать размер множества конечным числом, не превышающим 2^{31}.

2. Метод решения

2.1. Структуры данных

Как известно, *структура данных* есть модель данных в виде *математической структуры* $S = (M_1, \ldots, M_k, p_1,\ldots,p_n)$, где $M_1, \ldots, M_k$ – базисные множества, $p_1,\ldots,p_n$ – отношения между элементами базисных множеств.

Поскольку построение математической структуры основано на понятии множества, само множество не может быть определено, как структура данных.

В дальнейших рассуждениях мы будем опираться на следующее описание множества, вытекающее из сделанных выше допущений.

Каждому множеству $A = \{a_1, a_2, \ldots, a_n\} \subset U = \{u_1, \ldots, u_k\}$ поставим в соответствие *характеристический вектор* $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_k\}$, где

a) k – мощность U;

b) $\alpha_i = \begin{cases} 1, \text{ если } u_i \in A; \\ 0, \text{ если } u_i \notin A. \end{cases}$

c) $\sum_{i=1}^{k} \alpha_i = n$.

Все операции над множествами могут быть заменены в таком случае на операции над характеристическими векторами. Таким образом, в дальнейшем в работе мы будем решать задачу хранения и обработки именно характеристических векторов.

Важный вопрос хранения самих элементов множеств вида A, а также Универса U, в данной работе не рассматривается. Отметим только, что введение характеристического вектора для представления множества позволяет организовать хранение лишь Универса (например, в виде вектора элементов) и избежать многократного дублирования данных, содержащихся в конкретных множествах вида A.

Поскольку каждый элемент характеристического вектора принимает значения из множества {0, 1}, наиболее эффективной (с точки зрения расхода памяти) является его

реализация через битовое поле – непрерывный участок памяти (количество бит в котором достаточно для представления Универса), где каждый бит соответствует одному элементу вектора α.

Реализацию битового поля целесообразно вынести в отдельный класс, скрывающий детали, не существенные для представления и работы с множествами (некоторые из возникающих нюансов будут рассмотрены далее).

2.2. Алгоритмы

2.2.1. Битовое поле

Для работы с Битовым полем предлагается реализовать следующие операции:

- установить бит (в единицу);
- очистить бит (в ноль);
- получить значение бита;
- сравнить два битовых поля;
- выполнить операцию “логическое или” для двух битовых полей;
- выполнить операцию “логическое и” для двух битовых полей;
- выполнить операцию “логическое отрицание” для битового поля.

2.2.2. Множество

Для работы с Множеством предлагается реализовать следующие операции:

- включение элемента в множество;
- исключение элемента из множества;
- проверка наличия элемента в множестве;
- сравнение множеств
- сложение множеств;
- пересечение множеств;
- разность множеств;
- копирование множества;
- вычисление максимальной мощности множества.

Алгоритмически все они опираются на формулы, приведенные в разделе 1. Необходимо отметить, что, операции сложения, пересечения и разности должны создавать новые экземпляры структуры данных *Множество*.

3. Разработка программного комплекса

3.1. Структура

С учетом сформулированных выше предложений к реализации целесообразной представляется следующая модульная структура программы:

- BitField.h, BitField.cpp – модуль с классом, реализующим операции над *Битовыми полями*;
- Set.h, Set.cpp – модуль с классом, реализующим обработку *Множеств*;
- Main.cpp – модуль программы тестирования.

3.2. Спецификации классов

С учетом предложенных к реализации алгоритмов можно сделать следующие объявления классов.

Класс *Битовое поле* (файл BitField.h):

```
typedef unsigned int TELEM;
class TBitField
{
private:
  int  BitLen; // длина битового поля - макс. к-во битов
  TELEM *pMem; // память для представления битового поля
  int  MemLen; // к-во эл-тов Мем для представления бит.поля
  // методы реализации
  int   GetMemIndex(const int n) const; // индекс в pMem для бита n
  TELEM GetMemMask (const int n) const; // битовая маска для бита n
public:
  TBitField(int len);
  TBitField(const TBitField &bf);
  ~TBitField();
  // доступ к битам
  int GetLength(void) const;      // получить длину (к-во битов)
  void SetBit(const int n);       // установить бит
  void ClrBit(const int n);       // очистить бит
  int  GetBit(const int n) const; // получить значение бита
  // битовые операции
  int operator==(const TBitField &bf);       // сравнение
  TBitField& operator=(const TBitField &bf); // присваивание
  TBitField  operator|(const TBitField &bf); // операция "или"
  TBitField  operator&(const TBitField &bf); // операция "и"
  TBitField  operator~(void);                // отрицание
  friend istream &operator>>(istream &istr, TBitField &bf);
  friend ostream &operator<<(ostream &ostr, const TBitField &bf);
};
```

Класс *Множество* (файл Set.h)

```
class TSet
{
private:
  int MaxPower;       // максимальная мощность множества
  TBitField BitField; // битовое поле для хранения хар-го вектора
public:
  TSet(int mp);
  TSet(const TSet &s);       // конструктор копирования
  TSet(const TBitField &bf); // конструктор преобразования типа
  operator TBitField();      // преобразование типа к битовому полю
  // доступ к битам
  int GetMaxPower(void) const;     // максимальная мощность множества
  void InsElem(const int n);       // включить элемент в множество
  void DelElem(const int n);       // удалить элемент из множества
  int IsMember(const int n) const; // проверить наличие элемента в
                                   //  множестве
  // теоретико-множественные операции
  int operator== (const TSet &s); // сравнение
  TSet& operator=(const TSet &s); // присваивание
  TSet operator+ (const int n);   // включение элемента в множество
  TSet operator- (const int n);   // удаление элемента из множества
  TSet operator+ (const TSet &s); // объединение
  TSet operator* (const TSet &s); // пересечение
  TSet operator~ (void);          // дополнение
  friend istream &operator>>(istream &istr, TSet &bf);
  friend ostream &operator<<(ostream &ostr, const TSet &bf);
```

```
};
```

3.3. Этапы разработки

Предлагается следующая последовательность разработки и реализации требуемых программных средств:

- объявление класса `TBitField`;
- создание пустой заготовки программы тестирования;
- реализация конструкторов и деструктора;
- реализация методов ввода/вывода;
- объявление битового поля в программе тестирования, вызов метода вывода битового поля, запуск тестирующей программы;
- объявление класса `TSet`;
- реализация конструкторов;
- реализация методов ввода/вывода;
- объявление множества в программе тестирования, вызов метода вывода множества, запуск тестирующей программы;
- реализация оставшихся методов в классе `TBitField`.
- реализация оставшихся методов в классе `TSet`, их проверка в тестирующей программе.

3.4. Рекомендации по разработке

С учетом сделанных выше объявлений выпишем возможные реализации логической операции “и” (метод `operator&`) для битового поля и операции “пересечение” для множества.

```
TBitField TBitField::operator&(const TBitField &bf)
{
  int i, len = BitLen;
  if (bf.BitLen > len)
    len = bf.BitLen;
  TBitField temp(len);
  for (i = 0; i < MemLen; i++)
    temp.pMem[i] = pMem[i];
  for (i = 0; i < bf.MemLen; i++)
    temp.pMem[i] &= bf.pMem[i];
  return temp;
}

TSet TSet::operator*(const TSet &s)
{
  TSet temp(BitField & s.BitField);
  return temp;
}
```

Комментарии:

- результаты операций – новые экземпляры классов *Битовое поле* и *Множество*;
- длина нового битового поля (мощность нового множества) – максимум из длин (мощностей) аргументов;
- основная часть метода «и» – цикл, копирующий элементы поля `pMem` (динамического массива) в новое битовое поле (каждый элемент в данном случае

имеет тип `int`, то есть содержит 32 бита) и цикл, выполняющий логическое «и» с каждым элементом второго аргумента операции;

- реализации метода «пересечение» сводится к вызову метода «и» для соответствующих битовых полей.

3.5. Тестирование

Для тестирования созданных классов можно использовать следующую программу (в выбранной среде программирования необходимо создать консольное приложение).

```
#include "tset.h"

void main()
{
  int n, m, k, count;

  setlocale(LC_ALL, "Russian");
  cout << "Тестирование программ поддержки множества" << endl;
  cout << "               Решето Эратосфена" << endl;
  cout << "Введите верхнюю границу целых значений - ";
  cin  >> n;
  TSet s(n + 1);
  // заполнение множества
  for (m = 2; m <= n; m++)
    s.InsElem(m);
  // проверка до sqrt(n) и удаление кратных
  for (m = 2; m * m <= n; m++)
    // если м в s, удаление кратных
    if (s.IsMember(m))
      for (k = 2 * m; k <= n; k += m)
       if (s.IsMember(k))
         s.DelElem(k);
  // оставшиеся в s элементы - простые числа
  cout << endl << "Печать множества некратных чисел" << endl
       << s << endl;
  cout << endl << "Печать простых чисел" << endl;
  count = 0;
  k = 1;
  for (m = 2; m <= n; m++)
    if (s.IsMember(m))
    {
      count++;
      cout << setw(3) << m << " ";
      if (k++ % 10 == 0)
        cout << endl;
    }
  cout << endl;
  cout << "В первых " << n << " числах " << count << " простых" << endl;
}
```

4. Возможные темы дополнительных заданий

Для развития навыков практического программирования рекомендуются следующие направления расширения постановки задачи:

- Реализация представления множеств с хранением элементов Универса. Переход от визуализации множеств через характеристические вектора к визуализации элементов.

- Организация множеств с использованием шаблонов.
- Реализация дополнительных операций над множествами.

5. Критерии оценивания выполнения лабораторной работы

Лабораторная работа считается успешно выполненной в случае, когда реализованы классы битовое поле и множество, в программе тестирования демонстрируется работоспособность методов классов.

6. Вопросы и задания для самоконтроля

- Как определить количество элементов массива для хранения битового поля из N элементов?
- Что называется битовой маской?
- Какие предположения позволяют использовать класс битовое поле при реализации класса множество?

Лабораторная работа №2
Структуры хранения матриц специального вида

Введение

Понятие *Матрица* в европейской науке было введено в работах У. Гамильтона[3] и А. Кэли[4] в середине XIX века.

Матричные обозначения широко распространены в современной математике и её приложениях. *Матрица* – полезный аппарат для исследования многих задач теоретической и прикладной математики. Так, одной из важнейших является задача нахождения решения систем линейных алгебраических уравнений.

Следствием разнообразия областей применения матричного аппарата в современной науке является наличие в любом из больших математических программных комплексов (Mathcad, Mathematica, Derive, Mapple) подсистем, выполняющих операции над матрицами, а также существование специальных программных библиотек (ScalaPack, PlaPack), рассчитанных на обработку огромных (десятки и сотни тысяч строк) матриц, в том числе с использованием распределенных (параллельных) вычислений.

Помимо матриц общего вида, для которых наиболее естественной и наиболее часто используемой представляется программная реализация в виде двумерного массива, в математических приложениях выделяются различные матрицы специальных видов (треугольные, диагональные, …). Для таких матриц предпочтительно создание собственных способов хранения и обработки, учитывающих специфику их структуры, и потому более эффективных. Изучению некоторых из них посвящена данная работа.

1. Постановка учебной задачи

1.1. Основные понятия и определения

Матрица – в математике – прямоугольная таблица каких-либо *элементов* a_{ij} (чисел, математических выражений), состоящая из m строк и n столбцов:

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}.$$

Над матрицей можно производить действия по правилам матричной алгебры. При m=n матрица называется *квадратной*, а число n – её *порядком*.

Набор элементов матрицы (a_{11}, a_{22}, …, a_{nn}) называется *главной диагональю*.

Верхнетреугольной называется матрица, в которой все элементы под главной диагональю равны нулю:

---

[3] Гамильтон Уильям Роуан (1805-1865) – ирландский математик, иностранный член-корреспондент Петербургской АН (1837). Дал точное формальное изложение теории комплексных чисел. Построил систему чисел-кватернионов. В механике дал общий принцип наименьшего действия.

[4] Кэли Артур (1821-1895) – английский математик, иностранный член-корреспондент Петербургской АН (1870). Его авторству принадлежат труды по теории алгебраических квадратичных форм, проективной геометрии, математическому анализу, астрономии.

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ 0 & \dots & \dots \\ 0 & 0 & a_{mn} \end{pmatrix}.$$

Определение матрицы возможно также через понятие *Вектор*.

Вектор – в математике – набор a_i (чисел, математических выражений), состоящий из n элементов.

Тогда *Матрица* из m строк и n столбцов может быть определена как *Вектор* из n элементов, где каждый элемент, в свою очередь, является вектором из m элементов.

1.2. Операции над векторами

Пусть заданы два вектора $A = (a_1, a_2, \dots, a_n)$ и $B = (b_1, b_2, \dots, b_n)$. Рассмотрим следующие основные операции над векторами:

- Сравнение (A = B). Вектора считаются равными тогда и только тогда, когда $a_i = b_i$ при всех i = 1..n.
- Прибавление скаляра (A + a). Результатом сложения вектора A и скаляра a называется вектор $A' = (a_1 + a, a_2 + a, \dots, a_n + a)$.
- Вычитание скаляра (A – a). Результатом вычитания вектора A и скаляра a называется вектор $A' = (a_1 - a, a_2 - a, \dots, a_n - a)$.
- Умножение на скаляр (A * a). Результатом умножения вектора A на скаляр a называется вектор $A' = (a_1 * a, a_2 * a, \dots, a_n * a)$.
- Вычисление длины. Длиной вектора A называется скалярная величина $d = \sqrt{\sum_{i=1}^{n} a_i^2}$.
- Сложение векторов (A + B). Результатом сложения векторов A и B называется вектор $C = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n)$.
- Вычитание векторов (A – B). Результатом вычитания векторов A и B называется вектор $C = (a_1 - b_1, a_2 - b_2, \dots, a_n - b_n)$.
- Скалярное произведение векторов (A * B). Скалярным произведением векторов A и B называется скалярная величина $c = \sqrt{\sum_{i=1}^{n} a_i \times b_i}$.

1.3. Операции над матрицами

Пусть заданы две матрицы $A = (a_{ij})$ и $B = (b_{ij})$ и вектор $V = (v_j)$, где i = 1..m; j = 1..n. Рассмотрим следующие основные операции над матрицами:

- Сравнение (A = B). Матрицы считаются равными тогда и только тогда, когда $a_{ij} = b_{ij}$ при всех i = 1..m; j = 1..n.
- Умножение на скаляр (A * d). Результатом умножения матрицы A на скаляр d называется матрица $D = (d_{ij})$, где $d_{ij} = a_{ij} * d$, при всех i = 1..m; j = 1..n.
- Умножение на вектор (A * V). Результатом умножения матрицы A на вектор V называется вектор $D = (d_i)$, где $d_i = \sum_{j=1}^{n} a_{ij} \times v_j$, при всех i = 1..m; j = 1..n.

- Сложение матриц (A + B). Результатом сложения матриц A и B называется матрица $C = (c_{ij})$, где $c_{ij} = a_{ij} + b_{ij}$ при всех i = 1..m; j = 1..n.
- Вычитание матриц (A – B). Результатом вычитания матриц A и B называется матрица $C = (c_{ij})$, где $c_{ij} = a_{ij} – b_{ij}$ при всех i = 1..m; j = 1..n.
- Умножение матриц (A * B). Результатом умножения матриц $A = (a_{ij})$ и $B = (b_{jk})$ называется матрица $C = (c_{ik})$, где $c_{ij} = a_{ij} * b_{jk}$ при всех i = 1..m; j = 1..n; k = 1..n.

1.4. Требования к лабораторной работе

В рамках лабораторной работы ставится задача создания программных средств, поддерживающих эффективное хранение матриц специального вида (*верхнетреугольных*) и выполнение основных операций над ними:

- сложение/вычитание;
- умножение;
- копирование;
- сравнение.

Программные средства должны содержать:

- класс Вектор (на шаблонах);
- класс Матрица (на шаблонах);
- тестовое приложение, позволяющее задавать матрицы и осуществлять основные операции над ними.

1.5. Условия и ограничения

Сделаем следующие основные допущения:

- Условимся рассматривать в дальнейшем *верхнетреугольные* квадратные матрицы, состоящие из элементов произвольного типа.
- Будем считать размер матрицы конечным числом, не превышающим 2^{31}.

2. Метод решения

2.1. Структуры данных

Как известно, *структура данных* есть модель данных в виде *математической структуры* $S = (M_1, \ldots, M_k, p_1,\ldots,p_n)$, где $M_1, \ldots, M_k$ – базисные множества, $p_1,\ldots,p_n$ – отношения между элементами базисных множеств.

В соответствии с этим можно привести следующие определения:

1. Структура данных *Вектора* $V = (v_1, v_2, \ldots, v_n)$ есть

   $S_v = (M_v\ p_v)$, где

   $M_v = \{v_1,v_2, \ldots, v_n\}$ – базисное множество,

   $p_a\{v_i, v_j\} = \begin{cases} u, & j = i+1 \\ л, & j \neq i+1 \end{cases}$ – отношение следования.

2. Структура данных *Матрицы* $A = (a_{ij})$, где i = 1..n; j = 1..n есть

   $S_a = (M_a, p1_a, p2_a)$, где

   $M_a = \{a_{11}, a_{12}, \ldots, a_{nn-1},a_{nn}\}$ – базисное множество,

$$p1_a\{a_{ik}, a_{jk}\}_{1\le i,j\le n} = \begin{cases} u, & j = i+1 \\ л, & j \ne i+1 \end{cases} \quad k = 1..n$$

$$p2_a\{a_{ki}, a_{kj}\}_{1\le i,j\le n} = \begin{cases} u, & j = i+1 \\ л, & j \ne i+1 \end{cases} \quad k = 1..n$$

– отношения следования.

3. При определении *Матрицы* через *Вектор* (A = (v_i), где v_i – вектор из n элементов) структура данных примет вид:

$S1_a = (M1_a, p3_a)$, где

$M1_a = \{v_1, v_2, \ldots, v_n\}$ – базисное множество,

$$p3_a\{v_i, v_j\}_{1\le i,j\le n} = \begin{cases} u, & j = i+1 \\ л, & j \ne i+1 \end{cases}$$ – отношения следования.

4. Наконец, для *верхнетреугольной* матрицы имеет смысл задать структуру данных таким образом, чтобы исключить хранение нулевых элементов. Определение матрицы через вектор позволяет сделать это наилучшим образом:

$S2_a = (M2_a, p4_a)$, где

$M2_a = \{v_1, v_2, \ldots, v_n\}$ – базисное множество, где v_i есть вектор из i элементов.

$$p4_a\{v_i, v_j\}_{1\le i,j\le n} = \begin{cases} u, & j = i+1 \\ л, & j \ne i+1 \end{cases}$$ – отношения следования.

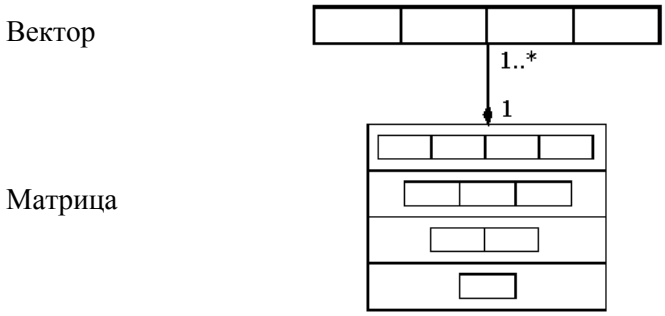


Рис. 1. Матрица как вектор векторных элементов

Очевидно сходство в задании структуры данных *Вектор*, как набора элементов, связанных отношением следования, и структуры данных *Матрица*, как набора элементов-векторов, связанных отношением следования. Этот факт позволяет единообразно организовать алгоритмы обработки векторов и матриц, а, следовательно, использовать при разработке требуемых классов механизм наследования.

2.2. Алгоритмы

2.2.1. Вектор

Для структуры данных *Вектор* предлагается реализовать следующие операции:

- вычисление длины;
- сравнение;
- прибавление/вычитание скаляра;
- умножение на скаляр;
- сложение/вычитание векторов;
- скалярное произведение векторов;
- создание копии.

2.2.2. Матрица

Для структуры данных *Матрица* предлагается реализовать следующие операции:

- сравнение:
- сложение/вычитание матриц:
- умножение матриц.

Как и в случае с вектором операции для структуры данных *Матрица* опираются на формулы, приведенные в разделе 2.

3. Разработка программного комплекса

3.1. Структура

С учетом предложенных к реализации структур данных целесообразной представляется следующая модульная структура программы:

- Vector.h, Vector.cpp – модуль, реализующий структуру данных *Вектор*;
- Matrix.h, Matrix.cpp – модуль, реализующий структуру данных *Матрица*;
- Main.cpp – модуль программы тестирования.

3.2. Спецификации классов

С учетом предложенных к реализации алгоритмов обработки выбранных структур данных можно сделать следующие объявления классов.

Класс *Вектор* (файл Vector.h):

```
template <class ValType>
class TVector
{
protected:
  ValType *pVector;
  int Size;       // размер вектора
  int StartIndex; // индекс первого элемента вектора
public:
  TVector(int s = 10, int si = 0);
  TVector(const TVector &v);                // конструктор копирования
  ~TVector();
  int GetSize()      { return Size;       } // размер вектора
  int GetStartIndex(){ return StartIndex; } // индекс первого элемента
  ValType& GetValue  (int pos);             // доступ с контролем индекса
  ValType& operator[](int pos);             // доступ
  int      operator==(const TVector &v);    // сравнение
  TVector& operator= (const TVector &v);    // присванивание
    // скалярные операции
  TVector  operator+ (const ValType &val);  // прибавить скаляр
  TVector  operator- (const ValType &val);  // вычесть скаляр
  TVector  operator* (const ValType &val);  // умножить на скаляр
    // векторные операции
  TVector  operator+ (const TVector &v);    // сложение
  TVector  operator- (const TVector &v);    // вычитание
  TVector  operator* (const TVector &v);    // скалярное произведение
    // ввод-вывод
  friend istream& operator>>(istream &in, TVector &v);
  friend ostream& operator<<(ostream &out, const TVector &v);
};
```

Класс *Матрица* (файл Matrix.h)

```
template <class ValType>
class TMatrix : public TVector<TVector<ValType> >
{
public:
  TMatrix(int s = 10);
  TMatrix(const TMatrix &mt);                    // копирование
  TMatrix(const TVector<TVector<ValType> > &mt); // преобразование типа
  TMatrix& operator==(const TMatrix &mt);        // сравнение
  TMatrix& operator= (const TMatrix &mt);        // присваивание
  TMatrix  operator+ (const TMatrix &mt);        // сложение
  TMatrix  operator- (const TMatrix &mt);        // вычитание
  TMatrix  operator* (const TMatrix &mt);        // умножение
  // ввод / вывод
  friend istream& operator>>(istream &in, TMatrix &mt);
  friend ostream & operator<<( ostream &out, const TMatrix &mt);
};
```

3.3. Этапы разработки

Предлагается следующая последовательность разработки и реализации требуемых программных средств:

1. объявление класса `TVector`;
2. создание пустой заготовки программы тестирования;
3. реализация конструкторов и деструктора;
4. реализация методов ввода/вывода;
5. объявление вектора в программе тестирования, вызов метода вывода вектора, запуск тестирующей программы;
6. объявление класса `TMatrix`;
7. реализация конструкторов и деструктора;
8. реализация методов ввода/вывода;
9. объявление матрицы в программе тестирования, вызов метода вывода матрицы, запуск тестирующей программы;
10. реализация скалярных операций в классе `TVector`, их проверка в тестирующей программе;
11. реализация оставшихся операций в классе `TVector`, их проверка в тестирующей программе;
12. реализация операций класса `TMatrix`, их проверка в тестирующей программе.

3.4. Рекомендации по разработке

С учетом сделанных выше объявлений выпишем возможную реализацию операций создания копии (метод `operator=`) для вектора и матрицы.

3.4.1. Вектор

```
template <class ValType>
TVector<ValType>& TVector<ValType>::operator=(const TVector &v)
{
  if (this != &v)
  {
    if (Size != v.Size)
    {
      delete[] pVector;
      pVector = new ValType[v.Size];
```

```
      }
      Size = v.Size;
      StartIndex = v.StartIndex;
      for (int i = 0; i < Size; i++)
        pVector[i] = v.pVector[i];
    }
    return *this;
  }
```

Комментарии:

- элементы вектора хранятся в динамически выделяемом непрерывном участке памяти `pVector`;
- при несовпадении размеров исходного и копируемого векторов необходимо освободить и выделить заново память;
- основная часть метода – цикл, поэлементно копирующий значения.

3.4.2. Матрица

```
template <class ValType>
TMatrix<ValType>& TMatrix<ValType>::operator=(
  const TMatrix<ValType> &mt)
{
  if (this != &mt)
  {
    if (Size != mt.Size)
    {
      delete[] pVector;
      pVector = new TVector<ValType>[mt.Size];
    }
    Size = mt.Size;
    StartIndex = mt.StartIndex;
    for (int i = 0; i < Size; i++)
      pVector[i] = mt.pVector[i];
  }
  return *this;
}
```

Комментарии:

- матрица хранится как вектор векторов переменной длины. В поле `pVector` каждая ячейка хранит указатель на объект класса *Вектор*;
- строка

  `pVector = new TVector<ValType>[mt.Size];`
- выделяет участок памяти для `mt.Size` объектов класса *Вектор* (при этом для каждого из них при создании используется конструктор с параметрами по умолчанию, устанавливающий размер вектора равный 10);
- строка в цикле

  `pVector[i] = mt.pVector[i];`

  для каждого созданного объекта-вектора вызывает его оператор присваивания, который установит правильный размер и скопирует элементы.

Нетрудно заметить, что реализация методов присваивания для вектора и матрицы совпадает с точностью до обозначений, за исключением одной строки, выделяющей память под новое количество элементов поля `pVector`.

Посмотрим теперь на реализацию метода сложения векторов.

```
template <class ValType>
TVector<ValType> TVector<ValType>::operator+(const TVector<ValType> &v)
{
  TVector temp(Size, StartIndex);
```

```
  for (int i = 0; i < Size; i++)
    temp.pVector[i] = pVector[i] + v.pVector[i];
  return temp;
}
```

Выбранный нами ранее способ определения матрицы как вектора векторных элементов позволяет сделать следующее

Утверждение

Пусть матрица A = (a_i), где a_i – вектор из i элементов, матрица B = (b_i), где b_i – вектор из i элементов, тогда результатом сложения матриц A и B будет матрица C = (c_i), где c_i – вектор из i элементов и $c_i = a_i + b_i$ при всех i = 1..n.

Истинность данного утверждения нетрудно показать. Также очевидно, что полученный способ вычисления суммы матриц совпадает с данным в разделе 2 определением суммы векторов. Таким образом, должны совпадать реализации методов сложения для класса *Вектор* и класса *Матрица*, с той лишь разницей, что операция

```
temp.pVector[i] = pVector[i] + v.pVector[i];
```

в случае сложения векторов суммирует скаляры, а в случае сложения матриц должна суммировать вектора.

В результате, реализация метода сложения матриц может быть записана следующим образом:

```
template <class ValType>
TMatrix<ValType> TMatrix<ValType>::operator+(const TMatrix<ValType> &mt)
{
  return TVector<TVector<ValType> >::operator+(mt);
}
```

3.5. Тестирование

Для тестирования созданных классов можно использовать следующую программу (в выбранной среде программирования необходимо создать консольное приложение).

```
#include <iostream>
#include "utmatrix.h"

void main()
{
  TMatrix<int> a(5), b(5), c(5);
  int i, j;

  setlocale(LC_ALL, "Russian");
  cout << "Тестирование программ поддержки треугольных матриц" << endl;
  for (i = 0; i < 5; i++)
    for (j = i; j < 5; j++ )
    {
      a[i][j] =  i * 10 + j;
      b[i][j] = (i * 10 + j) * 100;
    }
  c = a + b;
  cout << "Matrix a = " << endl << a << endl;
  cout << "Matrix b = " << endl << b << endl;
  cout << "Matrix c = a + b" << endl << c << endl;
}
```

4. Возможные темы дополнительных заданий

Для развития навыков практического программирования рекомендуются следующие направления расширения постановки задачи:

Реализация других способов представления матриц специального вида для проведения вычислительных экспериментов с целью сравнения эффективности хранения и обработки матриц (хранение без исключения нулевых элементов, плотное использование памяти – исключение элементов ниже главной диагонали).

Реализация дополнительных операций над матрицами – транспонирование, вычисление определителя, ранга, обратной матрицы, проверка на подобие.

Решение задачи приведения матрицы к треугольному виду. Решение систем линейных уравнений.

Разработка эффективных алгоритмов матричного умножения с использованием возможностей и принципов работы современных процессоров. Сравнение разных реализаций (проведение вычислительных экспериментов при больших порядках матриц).

5. Критерии оценивания выполнения лабораторной работы

Лабораторная работа считается успешно выполненной в случае, когда реализованы классы вектор и матрица, в программе демонстрируется работоспособность методов классов.

6. Вопросы и задания для самоконтроля

- Можно ли использовать разработанный класс вектор для решения задач аналитической геометрии?
- В чем преимущества описанного представления верхнетреугольных матриц?
- Какие примеры задач можно привести, для решения которых может потребоваться использование верхнетреугольных матриц?

Лабораторная работа №3
Вычисление арифметических выражений (стеки)

Введение

Лабораторная работа направлена на практическое освоение динамической структуры данных Стек. С этой целью в лабораторной работе изучаются различные варианты структуры хранения стеков и разрабатываются методы и программы решения ряда задач с использованием стеков. В качестве области приложений выбрана тема вычисления арифметических выражений, возникающей при трансляции программ на языке программирования высокого уровня в исполняемые программы.

При вычислении произвольных арифметических выражений возникают две основные задачи: проверка корректности введённого выражения и выполнение операций в порядке, определяемом их приоритетами и расстановкой скобок. Существует алгоритм, позволяющий реализовать вычисление произвольного арифметического выражения за один просмотр без хранения промежуточных результатов. Для реализации данного алгоритма выражение должно быть представлено в постфиксной форме. Рассматриваемые в данной лабораторной работе алгоритмы являются начальным введением в область машинных вычислений.

1. Постановка учебной задачи

1.1. Основные понятия и определения

Арифметическое выражение - выражение, в котором операндами являются объекты, над которыми выполняются арифметические операции. Например,

(1+2)/(3+4*6.7)-5.3*4.4

При такой форме записи (называемой *инфиксной*, где знаки операций стоят между операндами) порядок действий определяется расстановкой скобок и приоритетом операций. *Постфиксная* (или *обратная польская*) *форма* записи не содержит скобок, а знаки операций следуют после соответствующих операндов. Тогда для приведённого примера постфиксная форма будет иметь вид:

1 2+ 3 4 6.7*+/ 5.3 4.4* -

Обратная польская нотация была разработана австралийским ученым Чарльзом Хэмблином в середине 50-х годов прошлого столетия на основе польской нотации, которая была предложена в 1920 году польским математиком Яном Лукасевичем. Эта нотация лежит в основе организации вычислений для арифметических выражений. Известный ученый Эдсгер Дейкстра предложил алгоритм для перевода выражений из инфиксной в постфиксную форму. Данный алгоритм основан на использовании стека.

Стек (англ. stack), магазин – схема запоминания информации, при которой каждый вновь поступающий ее элемент как бы «проталкивает» вглубь отведенного участка памяти находящиеся там элементы (подобно патрону, помещаемому в магазин винтовки) и занимает крайнее положение (так называемую *вершину стека*). При выдаче информации из стека выдается элемент, расположенный в вершине стека, а оставшиеся элементы продвигаются к вершине; следовательно, элемент, поступивший последним, выдается первым [1]. Более строгое определение структуры дано в разделе 2 описания данной лабораторной работы.

1.2. Арифметические операции

В выражении, записанном в инфиксной форме, бинарные и унарные операции выполняются по единым правилам, и конкретный вид операции определяется ее положением в выражении. Для постфиксной формы попытка вычислить выражение, содержащее унарные операции, приведет к ошибке в связи с недостаточным количеством операций, т.к. знак операции стоит после операндов. При выполнении лабораторной работы можно предполагать, что унарные операции отсутствуют. Использование унарных операций может быть рассмотрено как тема для самостоятельного выполнения.

1.3. Требования к лабораторной работе

В рамках лабораторной работы ставится задача реализации программ, обеспечивающих поддержку стеков, и разработки программных средств, производящих обработку арифметических выражений, включая проверку правильности записи выражения, перевод в постфиксную форму и вычисление результата.

В начальной – самой простой постановке – можно предполагать, что проверка записи выражения состоит в контроле правильности расстановки скобок, перевод в постфиксную форму производится только для корректных выражений, а вычисление – для корректных выражений, содержащих только числовые операнды и допустимые знаки операций.

1.4. Синтаксический контроль расстановки скобок в арифметическом выражении

Полный синтаксический анализ арифметического выражения является сравнительно сложной задачей, и данная тема может быть заданием повышенной сложности для самостоятельного выполнения.

В рамках данной лабораторной работы предлагается ограничить контроль только правильной расстановкой скобок[5]. Таким образом, требуется проанализировать соответствие открывающих и закрывающих круглых скобок во введённом арифметическом выражении. Программа должна напечатать таблицу соответствия скобок, причем в таблице должно быть указано, для каких скобок отсутствуют парные им, а также общее количество найденных ошибок. Для идентификации скобок могут быть использованы их порядковые номера в выражении. Например, для арифметического выражения

1 2 3 4 5 6

(a+b1)/2+6.5)*(4.8+sin(x)

должна быть напечатана таблица вида:

СКОБКИ	
открывающая	закрывающая
1	2
-	3
5	6

[5] Данная простая постановка задания выбрана также и с учетом того, что рассматриваемая лабораторная работа является первой в расширенном лабораторном практикуме, который должен быть выполнен в течение одного учебного года.

4	-

Всего ошибок: 2.

Прочерки в таблице обозначают отсутствие соответствующей скобки. При отсутствии обнаруженных ошибок программа должна выдать соответствующее сообщение.

1.5. Перевод арифметического выражения из инфиксной формы записи в постфиксную

В рамках данного задания требуется разработать алгоритм и составить программу для перевода арифметического выражения из инфиксной формы записи в постфиксную. Инфиксная форма записи характеризуется наличием знаков операций между операндами. Например,

(1+2)/(3+4*6.7)-5.3*4.4

При такой форме записи порядок действий определяется расстановкой скобок и приоритетом операций. Постфиксная форма записи не содержит скобок, а знаки операций следуют после соответствующих операндов. Тогда для приведённого примера постфиксная форма будет иметь вид:

1 2+ 3 4 6.7*+/ 5.3 4.4* -

Так как при такой записи несколько операндов могут следовать подряд, то при выводе они разделяются пробелами.

Как результат программа должна напечатать постфиксную форму выражения или выдать сообщение о невозможности построения такой формы в случае обнаружения ошибок при расстановке скобок.

1.6. Вычисление арифметического выражения в постфиксной форме

Для выполнения данного задания необходимо разработать алгоритм и составить программу для вычисления арифметического выражения. Программа должна напечатать результат вычисления выражения или выдать сообщение о наличии нечисловых операндов.

1.7. Условия и ограничения

При выполнении лабораторной работы могут быть использованы следующие основные допущения:

- Можно предполагать, что арифметические выражения состоят не более чем из 255 символов.
- В качестве допустимых арифметических операций можно рассматривать только символы + (сложение), - (вычитание), * (умножение), / (деление).

2. Метод решения

2.1. Структуры данных

Как известно, *структура данных* есть модель данных в виде *математической структуры*

$S = (M_1, \ldots, M_k, p_1, \ldots, p_n)$,

где M_1, …, M_k – базисные множества, p_1,…,p_n – отношения между элементами базисных множеств.

Динамическая структура есть математическая структура, которой соответствует частично-упорядоченное (по включению) базовое множество M, элементы которого являются структурами данных. При этом отношения включения индуцируются операциями преобразования структуры данных.

Пусть p_1 – отношение следования, порождаемое операцией вставки, p_2 – отношение следования, порождаемое операцией удаления. Тогда *стек* есть структура

$S = (M, p_1, p_2)$,

в которой

- каждый элемент базисного множества есть структура,
- в любой момент существует только один конкретный элемент из M,
- элементы частично упорядочены по включению.

Таким образом, стек есть динамическая структура, операции вставки и удаления переводят стек из одного состояния в другое, а состояние стека характеризуется совокупностью хранимых элементов и положением вершины стека.

В качестве *структуры хранения* стека предлагается использовать одномерный (одноиндексный) массив, размещаемый в динамической области памяти. Для описания структуры хранения следует использовать следующие переменные:

- ***pMem*** – указатель на память, выделенную для хранения стека,
- ***MemSize*** – размер выделенной памяти,
- ***MaxMemSize*** – размер памяти, выделяемый по умолчанию, если при создании стека явно не указано требуемое количество элементов памяти,
- ***DataCount*** – количество запомненных в стеке значений,
- ***Hi*** – индекс элемента массива, в котором хранится последнее добавленное значение стека.

2.2. Алгоритмы

2.2.1. Стек

Для работы со стеком предлагается реализовать следующие операции:

- Метод ***Put*** – добавить элемент;

При добавлении элемента в стек необходимо переместить указатель вершины стека, записать элемент в соответствующую позицию динамического массива и увеличить количество элементов.

- Метод Get – удалить элемент;

При удалении элемента из стека необходимо возвратить значение из динамического массива по индексу вершины стека, переместить указатель вершины стека и уменьшить количество элементов.

- Метод IsEmpty – проверить стек на пустоту;

Стек пуст, если в нем нет ни одного элемента, т.е. когда количество элементов равно нулю.

- Метод IsFull – проверить стек на полноту.

Стек полон при исчерпании всей отведенной под хранение элементов памяти, т.е. когда значение DataCount совпадает со значением MemSize.

2.2.2. Проверка правильности расстановки скобок

На вход алгоритма поступает строка символов, на выходе должна быть выдана таблица соответствия номеров открывающихся и закрывающихся скобок и общее количество ошибок. Идея алгоритма, решающего поставленную задачу, состоит в следующем.

- Выражение просматривается посимвольно слева направо. Все символы, кроме скобок, игнорируются (т.е. просто производится переход к просмотру следующего символа).
- Если очередной символ – открывающая скобка, то её порядковый номер помещается в стек.
- Если очередной символ – закрывающая скобка, то производится выталкивание из стека номера открывающей скобки и запись этого номера в паре с номером закрывающей скобки в результирующую таблицу.
- Если в этой ситуации стек оказывается пустым, то вместо номера открывающей скобки записывается 0, а счетчик ошибок увеличивается на единицу.
- Если после просмотра всего выражения стек оказывается не пустым, то выталкиваются все оставшиеся номера открывающих скобок и записываются в результирующий массив в паре с 0 на месте номера закрывающей скобки, счетчик ошибок каждый раз увеличивается на единицу.

2.2.3. Перевод в постфиксную форму

Данный алгоритм основан на использовании стека.

На вход алгоритма поступает строка символов, на выходе должна быть получена строка с постфиксной формой.

Каждой операции и скобкам приписывается приоритет.

Знак операции	(	)	+ -	* /
Приоритет	0	1	2	3

Предполагается, что входная строка содержит синтаксически правильное выражение.

Входная строка просматривается посимвольно слева направо до достижения конца строки. Операндами будем считать любую последовательность символов входной строки, не совпадающую со знаками определённых в таблице операций. Операнды по мере их появления переписываются в выходную строку. При появлении во входной строке операции, происходит вычисление приоритета данной операции. Знак данной операции помещается в стек, если:

- Приоритет операции равен 0 (это « (»),
- Приоритет операции строго больше приоритета операции, лежащей на вершине стека,
- Стек пуст.

В противном случае из стека извлекаются все знаки операций с приоритетом больше или равным приоритету текущей операции. Они переписываются в выходную строку, после чего знак текущей операции помещается в стек.

Имеется особенность в обработке закрывающей скобки. Появление закрывающей скобки во входной строке приводит к выталкиванию и записи в выходную строку всех знаков операций до появления открывающей скобки. Открывающая скобка из стека выталкивается, но в выходную строку не записывается. Таким образом, ни открывающая, ни закрывающая скобки в выходную строку не попадают.

После просмотра всей входной строки происходит последовательное извлечение всех элементов стека с одновременной записью знаков операций, извлекаемых из стека, в выходную строку.

2.2.4. Вычисление арифметического выражения

Алгоритм вычисления арифметического выражения за один просмотр входной строки основан на использовании постфиксной формы записи выражения и работы со стеком. Входным данным служит строка символов, полученная в результате работы алгоритма из 2.2.3, выходным – результат вычисления выражения.

Выражение просматривается посимвольно слева направо. При обнаружении операнда производится перевод его в числовую форму и помещение в стек (если операнд не является числом, то вычисление прекращается с выдачей сообщения об ошибке.) При обнаружении знака операции происходит извлечение из стека двух значений, которые рассматриваются как операнд2 и операнд1 соответственно, и над ними производится обрабатываемая операция. Результат этой операции помещается в стек. По окончании просмотра всего выражения из стека извлекается окончательный результат.

3. Разработка программного комплекса

3.1. Структура

С учетом сформулированных выше предложений к реализации целесообразной представляется следующая модульная структура программы:

- TStack.h, TStack.cpp – модуль с классом, реализующим операции над стеком;
- TFormula.h, TFormula.cpp – модуль с классом, реализующим обработку арифметических выражений;
- StackTestkit.cpp – модуль программ тестирования.

3.2. Спецификации классов

С учетом предложенных к реализации алгоритмов можно сделать следующие объявления классов.

Класс TStack (файл TStack.h):

```
const int MaxMemSize = 25 // максимальный размер памяти для стека
typedef int TELEM;        // тип элемента СД
typedef TElem *PTElem     // тип указателя на СД

class TStack {
protected:  // поля
  PTElem pMem;   // указатель на массив элементов
 int MemSize;    // размер памяти для СД
 int DataCount;  // количество элементов в СД
 int Hi;         // индекс вершины стека
 virtual int GetNextIndex (int index); // получить следующий индекс

public:
    TStack (int Size = MaxMemSize);//конструктор
    ~Tstack();                     //деструктор
    int IsEmpty ( void ) const ;   // контроль пустоты
    int IsFull  ( void ) const ;   // контроль переполнения
    void  Put ( const TElem &Val );// добавить значение
    virtual TElem Get ( void ) ;   // извлечь значение
};
```

Тип *TElem* описывает тип значений элементов стека. В данном примере для типа элементов использован тип int, для обеспечения возможности хранения в стеке значений других типов (только стандартных) достаточно изменить определение типа TElem. Память

выделяется в динамической области памяти, размер стека можно задавать при инициализации объекта типа TStack. Для определения методов данного класса ниже приведен фрагмент файла TStack.cpp (модуль реализации).

```
#include ”TStack.h”
TStack :: TStack (int Size = MaxMemSize) {// конструктор
  DataCount = 0;
  if (Size=0) { Size = MaxMemSize; }
  MemSize = Size;
  pMem = new TElem [MemSize];
  Hi   = -1; }

TStack :: ~Tstack() {   //деструктор
  delete []pMem;
}

int TStack :: GetNextIndex (int index) { // получить следующий индекс
  return ++index ;
}

int TStack :: IsEmpty ( void ) const { return DataCount == 0;}

int TStack :: IsFull  ( void ) const { return DataCount == MemSize;}

void TStack :: Put ( const TElem &Val ) { // добавить значение
  Hi = GetNextIndex(Tail);
  pMem[Tail] = Val;
  DataCount++;
}

TElem TStack :: Get ( void ) {            // извлечь значение
  DataCount--;
  return pMem[Hi--];
}
```

В конструкторе выделяется память в динамически распределяемой области памяти, размер области памяти определяется как параметр конструктора (по умолчанию устанавливается MaxMemSize) и явно учитывает тип элемента стека (TElem).

Деструктор вызывает освобождение выделенной памяти.

Метод *IsEmpty* проверяет наличие запомненных в стеке значений и возвращает не нулевое значение в случае, когда в стеке элементов нет.

Метод *IsFull* оценивает возможность добавления в стек новых значений и возвращает не нулевое значение в случае, когда выделенная для хранения стека память заполнена полностью.

Метод *Put* используется для записи в стек значения Val. Перед обращением к этой функции необходимо проверять стек на полноту.

Метод *Get* обеспечивает получение значения, записанного в стек последним; выполнение операции выполняется с удалением возвращаемого значения из стека. Перед обращением к этой функции необходимо проверять стек на пустоту. Метод объявлен как виртуальный для последующей реализации схемы наследования.

Метод GetNextIndex используется для получения индекса следующего элемента. Данный метод используется для инкапсуляции отношения «Следующий элемент памяти» и необходим для последующей реализации схемы наследования.

Следует отметить следующие важные моменты:

- В приведенных примерах программ в качестве типа элементов стека выбран int (для смены типа надо переопределить обозначение TELEM. Может быть рекомендована разработка программ поддержки стека как шаблонов.

- В данных примерах программ отсутствует также проверка возможных ошибочных ситуаций (например, выборка значения из пустого стека). Введение таких проверок должно быть выполнено в обязательном порядке. При обнаружении ошибочных ситуаций может быть организовано завершение программ с выдачей аварийного кода завершения или, что более правильно, можно применить обработку исключений.
- Разработка конструктора копирования и перегрузку операции присваивания следует выполнить самостоятельно.

Класс TFormula (файл TFormula.h)

```
const int MaxLength = 255 // максимальный размер входной строки
class TFormula {
private:
  char Formula[MaxLength];       // входная строка
  char PostfixForm[MaxLength];   // постфиксная запись

public:
  TFTrans (char form[]);
  int FormulaChecker(int Brackets[], int size) ;// проверка правильности
  int FormulaConverter(void);     // перевод в постфиксную форму
  double FormulaCalculator(void); // вычисление результата
};
```

3.3. Этапы разработки

Важной частью лабораторной работы является организация поэтапной разработки программ. Предлагается следующая последовательность разработки:

Этап 1. Реализация программ поддержки стека.

Этап 2. Реализация программ работы с арифметическим выражением.

Этап 3. Выполнение дополнительных заданий лабораторной работы.

На каждом этапе рекомендуется последовательная разработка необходимых программ с обязательным тщательным тестированием правильности их работы. Подготовке средств проверки правильности (тестов) следует уделить особое внимание. Так, например, при разработке программ стека может быть предложена следующая последовательность реализации:

1. Определение необходимых классов, проектирование схемы наследования, разработка спецификаций классов;
2. Реализация конструктора и деструктора класса TStack, методов проверки пустоты и переполнения стека. Тестирование разработанных программ.
3. Реализация метода вставки в стек. Тестирование.
4. Реализация метода извлечения из стека. Тестирование.

Для более надежного тестирования может быть рекомендовано попарная разработка – разработчик разрабатывает программы, тестировщик готовит тесты и выполняет проверку. При этом успешность работы разработчика определяется безошибочным выполнением тестов, а качество подготовленных тестов состоит в количестве обнаруженных ошибочных ситуаций при выполнении разработанных программ. Кроме того, может быть рекомендована поставка явно ошибочных реализаций стека (без исходного кода) с тем, чтобы были подготовлены тесты, выявляющих имеющиеся ошибочные ситуации в работе стека.

Для этапа 2 может быть предложена следующая последовательность разработки:

1. Разработка спецификации класса TFormula, реализация конструктора;

2. Реализация метода вычисления значения арифметического выражения (для этого, можно, например, задавать выражение уже в приведенной постфиксной форме). Тестирование.
3. Реализация метода перевода арифметического выражения в постфиксную форму (выражение задается в синтаксически правильной форме). Тестирование.
4. Реализация метода проверки правильности расстановки скобок в арифметическом выражении. Тестирование.

3.4. Рекомендации по разработке

Для снижения сложности начальной разработки можно рассматривать в качестве операндов одноразрядные числа.

При формировании постфиксной формы необходимо после каждого операнда добавлять пробелы, что потребуется для вычисления результата.

4. Возможные темы дополнительных заданий

Для развития навыков практического программирования рекомендуются следующие направления расширения постановки задачи:

1. Разработать программы для полного синтаксического контроля правильности записи арифметического выражения.
2. Разработать программы для обработки расширенной формы задания арифметических выражений (задание операндов с использованием переменных, использование в выражениях расширенного набора операций, вычисление результатов последовательности арифметических выражений).
3. Разработать программы для решения задачи, широко известной под названием «Ханойская башня». Данная задача обязана своим происхождением индусской легенде, которая рассказывает, что в большом храме Бенареса бронзовая плита поддерживает три алмазных стержня, на один из которых бог нанизал во время сотворения мира 64 золотых диска. С тех пор день и ночь монахи, сменяя друг друга, каждую секунду перекладывают по одному диску согласно описанным ниже правилам. Конец мира наступит тогда, когда все 64 диска будут перемещены, на что потребуется чуть больше 584 миллиарда лет.

   В общем виде задача состоит в следующем.

   Пирамидка (башня) состоит из N дисков, положенных один на другой так, что чем выше находится диск, тем меньше его диаметр.

   Требуется переместить эту башню, соблюдая следующие правила:

   - за один раз можно перекладывать только один диск;
   - нельзя класть диск на диск меньшего размера;
   - можно пользоваться только одной резервной площадкой.

   При выполнении работы программа решения поставленной задачи может быть дополнена средствами визуализации на экране дисплея процесса перемещения дисков.

   Разработать программы для выполнения вычислений, задаваемых при помощи последовательности операторов некоторого подмножества алгоритмического языка – пример подобной возможной постановки задачи приведен в [3].

5. Критерии оценивания выполнения лабораторной работы

Лабораторная работа считается успешно выполненной в случае, когда реализована проверка правильности расстановки скобок, выполняется перевод записи выражения в постфиксную форму, правильно вычисляются корректно записанные арифметические выражения.

Набор тестов для проверки:

1. 1+2
   Ошибок 0

   Постфиксная форма 1 2 +

   Результат 3

2. 1+2*(3-2)-4
   Ошибок 0

   Постфиксная форма 1 2 3 2 -*+4 –

   Результат -1

3. (((1+23)*1-22)+5)*2-(7
   Ошибок 1

   Перевод и вычисление невозможно

4. 1+2/(3-3)
   Ошибок 0

   Постфиксная форма 1 2 3 3-/+

   Результат Деление на ноль невозможно

5. 1++1
   Ошибок 0

   Постфиксная форма 1 1 ++

   Результат Недостаточно операндов

В минимальном объеме должны быть реализованы тесты 1-3.

6. Вопросы и задания для самоконтроля

- Какие операции могут быть дополнительно определены для стека?
- Как можно организовать обработку унарных операций?
- Составить постфиксную форму для арифметического выражения (((1+23)*1-22)+5)*2-7
- Какие правила работы со стеком, реализация которого выполнена в библиотеке STL?

- Какие примеры задач можно привести, для решения которых может потребоваться использование стека?

Лабораторная работа №4
Имитационное моделирование системы обслуживания потока заданий на ЭВМ (очереди)

Введение

Лабораторная работа направлена на практическое освоение динамической структуры данных Очередь. С этой целью в лабораторной работе изучаются различные варианты структуры хранения очереди и разрабатываются методы и программы решения задач с использованием очередей. В качестве области приложений выбрана тема эффективной организации выполнения потока заданий на вычислительных системах.

Очередь характеризуется таким порядком обработки значений, при котором вставка новых элементов производится в конец очереди, а извлечение – из начала. Подобная организация данных широко встречается в различных приложениях. В качестве примера использования очереди предлагается задача разработки системы имитации однопроцессорной ЭВМ. Рассматриваемая в рамках лабораторной работы схема имитации является одной из наиболее простых моделей обслуживания заданий в вычислительной системе и обеспечивает тем самым лишь начальное ознакомление с проблемами моделирования и анализа эффективности функционирования реальных вычислительных систем.

1. Постановка учебной задачи

1.1. Основные понятия и определения

Имитационное моделирование реально существующих объектов и явлений – физических, химических, биологических, социальных процессов, живых и неживых систем, инженерных конструкций, конструируемых объектов – представляет собой построение математической модели, которая описывает изучаемое явление с достаточной точностью, и последующую реализацию разработанной модели на ЭВМ для проведения вычислительных экспериментов с целью изучения свойств моделируемых явлений. Использование имитационного моделирования позволяет проводить изучение исследуемых объектов и явлений без проведения реальных (натурных) экспериментов.

Очередь (англ. queue), – схема запоминания информации, при которой каждый вновь поступающий ее элемент занимает крайнее положение (*конец очереди*). При выдаче информации из очереди выдается элемент, расположенный в очереди первым (*начало очереди*), а оставшиеся элементы продвигаются к началу; следовательно, элемент, поступивший первым, выдается первым. Более строгое определение структуры дано в разделе 2 описания данной лабораторной работы.

1.2. Требования к лабораторной работе

Для вычислительной системы (ВС) с одним процессором и однопрограммным последовательным режимом выполнения поступающих заданий требуется разработать программную систему для имитации процесса обслуживания заданий в ВС. При построении

модели функционирования ВС должны учитываться следующие основные моменты обслуживания заданий:

- генерация нового задания;
- постановка задания в очередь для ожидания момента освобождения процессора;
- выборка задания из очереди при освобождении процессора после обслуживания очередного задания.

По результатам проводимых вычислительных экспериментов система имитации должна выводить информацию об условиях проведения эксперимента (интенсивность потока заданий, размер очереди заданий, производительность процессора, число тактов имитации) и полученные в результате имитации показатели функционирования вычислительной системы, в т.ч.

- количество поступивших в ВС заданий;
- количество отказов в обслуживании заданий из-за переполнения очереди;
- среднее количество тактов выполнения заданий;
- количество тактов простоя процессора из-за отсутствия в очереди заданий для обслуживания.

Показатели функционирования вычислительной системы, получаемые при помощи систем имитации, могут использоваться для оценки эффективности применения ВС; по результатам анализа показателей могут быть приняты рекомендации о целесообразной модернизации характеристик ВС (например, при длительных простоях процессора и при отсутствии отказов от обслуживания заданий желательно повышение интенсивности потока обслуживаемых заданий и т.д.).

1.3. Условия и ограничения

Сделаем следующие основные допущения:

- При планировании очередности обслуживания заданий возможность задания приоритетов не учитывается.
- Моменты появления новых заданий и моменты освобождения процессора рассматриваются как случайные события.

2. Метод решения

2.1. Структуры данных

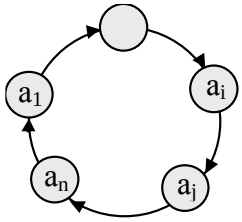
Напомним, что *динамическая структура* есть математическая структура, которой соответствует частично-упорядоченное (по включению) базовое множество M, операции вставки и удаления элементы которого являются структурами данных. При этом отношения включения индуцируются операциями преобразования структуры данных.

Таким образом, *очередь* есть динамическая структура, операции вставки и удаления переводят очередь из одного состояния в другое, при этом добавление новых элементов осуществляется в конец очереди, а извлечение – из начала очереди (дисциплина обслуживания «первым пришел – первым обслужен.

Важной задачей при реализации системы обслуживания очереди является выбор структуры хранения, обеспечивающей решение проблемы эффективного использования памяти без перепаковок и без использования связных списков (требующих дополнительных затрат памяти на указатели).

Как и в случае со стеком, в качестве *структуры хранения* очереди предлагается использовать одномерный (одноиндексный) массив, размещаемый в динамической области памяти. В связи с характером обработки значений, располагаемых в очереди, для указания хранимых в очереди данных необходимо иметь два указателя – на начало и конец очереди. Эти указатели увеличивают свое значение: один при вставке, другой при извлечении элемента. Таким образом, в ходе функционирования очереди может возникнуть ситуация, когда оба указателя достигнут своего наибольшего значения и дальнейшее пополнение очереди станет невозможным, несмотря на наличие свободного пространства в очереди. Одним из решений проблемы «движения» очереди является организация на одномерном массиве кольцевого буфера. *Кольцевым буфером* называется структура хранения, получаемая из вектора расширением отношения следования парой $p(a_n,a_1)$.

Рис. 2. Общая схема структуры хранения вида кольцевой буфер



Структура хранения очереди в виде кольцевого буфера может быть определена как одномерный (одноиндексный) массив, размещаемый в динамической области памяти и расположение данных в котором определяется при помощи следующего набора параметров:

pMem – указатель на память, выделенную для кольцевого буфера,

MemSize – размер выделенной памяти,

MaxMemSize – размер памяти, выделяемый по умолчанию, если при создании кольцевого буфера явно не указано требуемое количество элементов памяти,

DataCount – количество запомненных в очереди значений,

Hi – индекс элемента массива, в котором хранится последний элемент очереди,

Li – индекс элемента массива, в котором хранится первый элемент очереди.

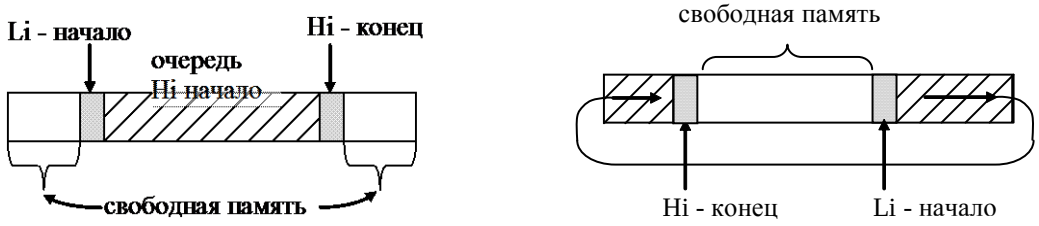


Рис. 3. Общая схема реализация кольцевого буфера на одноиндексном массиве

В связи с тем, что структура хранения очереди во многом аналогична структуре хранения стек, предлагается класс для реализации очереди построить наследованием от класса стек, описанного в лабораторной работе №3. При наследовании достаточно переопределить методы Get и GetNextIndex. В методе Get изменяется индекс для получения элемента (извлечение значений происходит из начала очереди), метод GetNextIndex реализует отношение следования на кольцевом буфере.

2.2. Алгоритмы

2.2.1. Очередь

Для работы с очередью предлагается реализовать следующие операции:

- Метод Put - добавить элемент;

Добавление элемента в очередь аналогично добавлению в стек, используется метод класса стек.

- Метод Get - удалить элемент;

При удалении элемента из очереди необходимо возвратить значение из динамического массива по индексу начала очереди, переместить указатель начала очереди и уменьшить количество элементов.

- Метод IsEmpty - проверить очередь на пустоту;
- Метод IsFull - проверить очередь на полноту.

Данные методы наследуются из класса стек.

2.2.2. Моделирование потока задач

Для моделирования момента появления нового задания можно использовать значение датчика случайных чисел. Если значение датчика меньше некоторого порогового значения $q1$, $0 \le q1 \le 1$, то считается, что на данном такте имитации в вычислительную систему поступает новое задание (тем самым параметр $q1$ можно интерпретировать как величину, регулирующую интенсивность потока заданий – новое задание генерируется в среднем один раз за $(1/q1)$ тактов).

2.2.3. Моделирование работы процессора

Моделирование момента завершения обработки очередного задания также может быть выполнено по описанной выше схеме. При помощи датчика случайных чисел формируется еще одно случайное значение, и если это значение меньше порогового значения $q2$, $0 \le q2 \le 1$,то принимается, что на данном такте имитации процессор завершил обслуживание очередного задания и готов приступить к обработке задания из очереди ожидания (тем самым параметр $q2$ можно интерпретировать как величину, характеризующую производительность процессора вычислительной системы – каждое задание обслуживается в среднем за $(1/q2)$ тактов).

2.2.4. Моделирование вычислительной системы

Возможная простая схема имитации процесса поступления и обслуживания заданий в вычислительной системе состоит в следующем.

1. Каждое задание в системе представляется некоторым однозначным идентификатором (например, порядковым номером задания).

2. Для проведения расчетов фиксируется (или указывается в диалоге) число тактов работы системы.

3. На каждом такте опрашивается состояние потока задач и процессор.

4. Регистрация нового задания в вычислительной системе может быть сведена к запоминанию идентификатора задания в очередь ожидания процессора. Ситуацию переполнения очереди заданий следует понимать как нехватку ресурсов вычислительной системы для ввода нового задания (отказ от обслуживания).

5. Для моделирования процесса обработки заданий следует учитывать, что процессор может быть занят обслуживанием очередного задания, либо же может находиться в состоянии ожидания (простоя).

6. В случае освобождения процессора предпринимается попытка его загрузки. Для этого извлекается задание из очереди ожидания.

7. Простой процессора возникает в случае, когда при завершении обработки очередного задания очередь ожидающих заданий оказывается пустой.

8. После проведения всех тактов имитации производится вывод характеристик вычислительной системы:

- количество поступивших в вычислительную систему заданий в течение всего процесса имитации;
- количество отказов в обслуживании заданий из-за переполнения очереди – целесообразно считать процент отказов как отношение количества не поставленных в очередь заданий к общему количеству сгенерированных заданий, умноженное на 100%;
- среднее количество тактов выполнения задания;
- количество тактов простоя процессора из-за отсутствия в очереди заданий для обслуживания – целесообразно считать процент простоя процессора как отношение количества тактов простоя процессора к общему количеству тактов имитации, умноженное на 100%.

3. Разработка программного комплекса

3.1. Структура

С учетом сформулированных выше предложений к реализации целесообразной представляется следующая модульная структура программы:

- TStack.h, TStack.cpp – модуль с классом, реализующим операции над стеком;
- TQueue.h, TQueue.cpp – модуль с классом, реализующим операции над очередью;
- TJobStream.h, TJobStream.cpp – модуль с классом, реализующим поток задач;
- TProc.h, TProc.cpp – модуль с классом, реализующим процессор;
- QueueTestkit.cpp – модуль программы тестирования.

3.2. Спецификации классов

С учетом предложенных к реализации алгоритмов можно сделать следующие объявления классов.

Класс $TStack$ (файл TStack.h) (см. описание лабораторной работы №3)

Класс $TQueue$ (файл TQueue.h):

```
class TQueue : public TStack {
 protected:
   int Li;                          // индекс начала очереди
   int GetNextIndex (int index); // получить следующий индекс
public:
   TQueue (int Size = MaxMemSize):TStack(Size) {Li = 0;};
   TElem Get ( void ) ;             // извлечь значение
};
```

Тип TElem описывает тип значений элементов структуры данных (определяется в TStack.h). Реализация класса TStack подробно описана в лабораторной работе №3. Для определения методов класса TQueue ниже приведен фрагмент файла TQueue.cpp (модуль реализации).

```
#include "Queue.h"

int Queue :: GetNextIndex(int index) { // Получение следующего индекса
   return  ++ index % MemSize ;
}

TElem Queue :: Get ( void ) {// извлечь значение
   if ( ~IsEmpty() { // очередь не пуста?
     TElem tmp = pMem[Li];
```

```
    Head = GetNextIndex( Li );
    DataCount--;
    return tmp;
  }
}
```

В конструкторе вызывается конструктор базового класса, размер области памяти определяется как параметр конструктора (по умолчанию устанавливается MaxMemSize) и явно учитывает тип элемента очереди (TElem), кроме этого устанавливается значение индекса начала очереди Li = 0.

Деструктор, методы IsEmpty, IsFull, Put наследуются из базового класса. Перед обращением к функции Put необходимо проверять очередь на полноту.

Метод Get переопределяется для очереди и обеспечивает получение значения из очереди по указателю начала очереди; выполнение операции выполняется с удалением возвращаемого значения из очереди. Перед обращением к этой функции необходимо проверять очередь на пустоту.

Метод GetNextIndex переопределяется в классе очереди и обеспечивает представление очереди в виде кольцевого буфера, позволяя устанавливать значения индексов начала и конца очереди при достижении ими граничного значения в 0.

Класс $TJobStream$ (файл TJobStream.h)

```
class TJobStream {
  private:
    double JobIntens;    // интенсивность потока задач
  public:
    TJobStream (int Intens);
    int GetNewJob(void); // генерация нового задания
};
```

Класс $TProc$ (файл TProc.h)

```
class TProc {
  private:
    double ProcRate;       // производительность процессора
    int JobId;             // Id выполняемого задания
  public:
    TProc (int Rate);
    int IsProcBusy(void) ; // процессор занят?
    int RunNewJob (int JobId); // приступить к выполнению нового задания
};
```

3.3. Этапы разработки

Как и ранее, важной частью лабораторной работы является организация поэтапной разработки программ. Предлагается следующая последовательность разработки (с учетом наличия работоспособного класса TStack):

Этап 1. Реализация программ поддержки очереди.

Этап 2. Реализация системы имитации обслуживания заданий.

Этап 3. Выполнение дополнительных заданий лабораторной работы.

При выполнении лабораторной работы следует продолжить практические освоение технологии итеративной разработки с самым тщательным тестированием правильности работы программ. Для проверки правильности должно разрабатываться достаточное количество тестов, которые гарантированно проверяют основные возможные ситуации при выполнении программ. Переход к следующему этапу разработки должен производиться только при успешном выполнении тестов; при проверке разработанных программ очередного этапа должны быть задействованы тесты всех предшествующих этапов.

Учитывая готовность класса TStack, при разработке программ очереди может быть предложена следующая последовательность реализации:

1. Реализация конструктора TQueue, перегрузка метода следования GetNextIndex. Тестирование разработанных программ.

2. Реализация метода извлечения из очереди. Тестирование.

Для этапа 2 может быть использована следующая последовательность разработки:

1. Разработка спецификации класса TJobStream, реализация конструктора и метода генерации нового задания.

2. Разработка спецификации класса TProc, реализация конструктора.

3. Реализация метода опроса состояния процессора и метода запуска нового задания на выполнение. Тестирование.

4. Разработка управляющей программы системы имитации, обеспечивающий циклическое выполнение тактов имитации: генерация нового задания, проверка состояния процессора, запуск нового задания. Тестирование.

5. Разработка программы вывода результатов имитации.

6. Выполнение вычислительных экспериментов. Подготовка отчета о выполнении лабораторной работы.

Входными параметрами вычислительного эксперимента являются: интенсивность потока задач, производительность процессора и количество тактов имитации. Максимальный размер очереди определен константным значением.

3.4. Рекомендации по разработке

Предусмотреть два режима работы программы – при небольшом числе тактов имитации следует выполнять потактовую распечатку состояния вычислительной системы и печать статистики, для длительных периодов имитации следует проводить только печать статистики.

Разработку программ лабораторной работы следует провести с обязательной проверкой возможных ошибочных ситуаций (например, выборка значения из пустой очереди). При обнаружении ошибочных ситуаций должно быть организовано завершение программы с выдачей аварийного кода завершения или, что более правильно, можно применить обработку исключений.

4. Возможные темы дополнительных заданий

Для развития навыков практического программирования рекомендуются следующие направления расширения постановки задачи:

1. Реализовать возможность задания приоритетов поступающих в систему задач – в этом случае задачи в очереди должны упорядочиваться в соответствии со своими приоритетами.

2. Дополнить постановку лабораторной работы возможностью моделирования для поступающих заданий различной длительностью выполнения.

3. Расширить разработанную систему для возможности имитации многопроцессорных вычислительных систем, в которых может быть несколько процессоров. Возможное расширение постановки лабораторной работы может состоять в использовании нескольких процессоров для выполнения отдельных заданий. Важным моментом при анализе поведения таких систем является выбор стратегии выделения имеющих процессоров для выполнения

заданий (так, например, может возникать ситуации длительного откладывания выполнения заданий, для которых необходимо большое количество процессоров).

4. Расширить разработанную систему имитации возможностью использования нескольких очередей входных заданий.

При расширении разработанных программ следует обратить внимание на трудоемкость необходимой модификации существующего программного кода: количество методов, в которые требуется внести изменения; количество новых внесенных ошибок разработки и т.п.

5. Критерии оценивания выполнения лабораторной работы

Лабораторная работа считается сданной при выполнении минимального набора требований – реализованы все классы, при малых значениях количества тактов имитации итоговая статистика соответствует результатам потактового выполнения.

Набор тестов для проверки:

1. Количество тактов имитации 10
   Размер очереди 3

   Интенсивность потока задач 0.5

   Производительность процессора 0.5

2. Количество тактов имитации 10
   Размер очереди 3

   Интенсивность потока задач 0.5

   Производительность процессора 0.2

3. Количество тактов имитации 10
   Размер очереди 3

   Интенсивность потока задач 0.2

   Производительность процессора 0.5

В первом тесте система является сбалансированной, во втором тесте малоинтенсивный поток задач при высокопроизводительном процессоре (в результате вычислительного эксперимента должен быть получен большой процент простоя процессора), в третьем тесте интенсивный поток задач при малопроизводительном процессоре (вычислительный эксперимент должен показать большое количество заданий, которым отказано в обслуживании из-за переполнения очереди заданий).

Построение подобных оценок результатов вычислительных экспериментов и рекомендации по изменению вычислительной системы могут быть включены в состав разрабатываемой системы имитации.

6. Вопросы и задания для самоконтроля

- Какие изменения могут потребоваться в структуре хранения при реализации очереди с приоритетами?

- Какие дополнительные понятия могут быть предложены при разработке системы имитации для более адекватного описания функционирования вычислительных систем?
- Насколько точно использованный способ генерации заданий описывает реально существующие потоки заданий в вычислительных системах? Какие дополнительные схемы генерации входного потока заданий могут быть предложены?
- Может ли разработанная система имитации быть использована в других областях приложений?
- Какие правила работы с очередью, реализация которой выполнена в библиотеке STL?
- Какие примеры задач можно привести, для решения которых может потребоваться использование очереди?

Лабораторная работа №5
Аналитические преобразования полиномов от нескольких переменных
(списки)

Введение

Наряду с привычным вычислительным применением компьютеры широко используются и для аналитической обработки данных. Среди примеров таких приложений – компьютерное доказательство теорем, логический вывод, анализ текстовой информации и многое другое. Среди таких примеров и задача обработки полиномов, задаваемых в общей аналитической форме. Полиномы являются хорошо изученной областью математики (алгебра полиномов), которая широко используется в приложениях (аппроксимация экспериментальных данных, построение функциональных зависимостей и т.п.).

Лабораторная работа направлена на изучение методов компьютерной обработки полиномов. С этой целью в лабораторной работе изучаются различные варианты структуры хранения и разрабатываются программы для обработки полиномов. Основной учебной целью работы является практическое освоение методов организации структур хранения данных с помощью списков. В ходе выполнения лабораторной работы разрабатывается общая форма представления линейных списков, разрабатываются программы работы со списками, которые могут быть использованы и в других областях приложений.

1. Постановка учебной задачи

1.1. Основные понятия и определения

Под *полиномом* от одной переменной понимается выражение вида:

$P_n(x)=a_n x^n+a_{n-1}x^{n-1}+\ldots+a_1x^1+a_0$

или в более общем виде

$$P_n(x) = \sum_{i=0}^{n} a_i x^i$$

где:

n – степень полинома;

a_i, $0 \le i \le n$ – коэффициенты полинома (действительные или комплексные числа).

Полином можно определить также как выражение из нескольких *термов*, соединенных знаками сложения или вычитания. *Терм* включает коэффициент и *моном*, содержащий одну или несколько переменных, каждая из которых может иметь *степень*

$$P(x,y,z) = \sum_{i,\,j,\,k} a_{ijk}\, x^i\, y^j\, z^k .$$

Как пример, полином от трех переменных может иметь вид

$P(X, Y, Z)=3x^3z - 2y^2z^2 + 3.$

Подобными называют два (или более) мономов, имеющих одинаковые степени при неизвестных.

В число основных операций над полиномами входят действия по вычислению значений полинома при заданных значениях переменных, а также большинство известных математических операций (сложение, вычитание, вычисление частных производных, интегрирование и т.п.).

Полиномы как формальный объект хорошо изучены в математике. Математическая модель данной предметной области – *алгебра полиномов*.

1.2. Требования к лабораторной работе

В рамках лабораторной работы ставится задача создания программных средств, поддерживающих эффективное представление полиномов и выполнение следующих операций над ними:

- ввод полинома
- организация хранения полинома
- удаление введенного ранее полинома;
- копирование полинома;
- сложение двух полиномов;
- вычисление значения полинома при заданных значениях переменных;
- вывод.

Состав реализуемых операций над полиномами может быть расширен при постановке задания лабораторной работы.

Предполагается, что в качестве структуры хранения будут использоваться списки. В качестве дополнительной цели в лабораторной работе ставится также задача разработки некоторого общего представления списков и операций по их обработке. В числе операций над списками должны быть реализованы следующие действия:

- поддержка понятия текущего звена;
- вставка звеньев в начало, после текущей позиции и в конец списков;
- удаление звеньев в начале и в текущей позиции списков;
- организация последовательного доступа к звеньям списка (итератор).

В ходе выполнения лабораторной работы должно быть выполнено сопоставление разработанных средств работы со списками с возможностями работы со списками в библиотеке STL.

Важной частью лабораторной работы должна являться разработка диалоговой управляющей программы с наглядным визуальным интерфейсом, который обеспечивает возможности создавать полиномы, выполнять реализованные операции обработки полиномов, демонстрировать вид имеющихся полиномов.

1.3. Условия и ограничения

При выполнении лабораторной работы можно использовать следующие основные предположения:

- Разработка структуры хранения должна быть ориентирована на представление полиномов от трех неизвестных.
- Степени переменных полиномов не могут превышать значения 9, т.е. $0 \le i, j, k \le 9$.

- Число мономов в полиномах существенно меньше максимально возможного количества (тем самым, в структуре хранения должны находиться только мономы с ненулевыми коэффициентами).

2. Методы выполнения лабораторной работы

2.1. Структуры данных и структуры хранения

2.1.1. Структуры хранения полиномов

Для представления полиномов могут быть выбраны различные структуры хранения. Критериями выбора структуры хранения являются размер требуемой памяти и сложность (трудоемкость) реализации операций над полиномами.

Возможный вариант структуры хранения – использование массивов (в случае полиномов от трех переменных – трехмерная матрица коэффициентов полинома). Такой способ обеспечивает простую реализацию операций над полиномами, но он не эффективен в части объема требуемой памяти. Так, при сделанных допущениях для хранения одного полинома в массиве потребуется порядка 8000 байт памяти – при этом в памяти будут храниться в основном параметры мономов с нулевыми коэффициентами.

Разработка более эффективной структуры хранения должна быть выполнена с учетом следующих рекомендаций:

- в структуре хранения должны храниться данные только для мономов с ненулевыми коэффициентами;
- порядок размещения данных в структуре хранения должен обеспечивать возможность быстрого поиска мономов с заданными свойствами (например, для приведения подобных).

Для организации быстрого доступа может быть использовано упорядоченное хранение мономов. Для задания порядка следования можно принять лексикографическое упорядочивание по степеням переменных, при котором мономы упорядочиваются по степеням первой переменной, потом по второй переменной, и только затем по третьей переменной. В общем виде это правило можно записать как соотношение: моном $X^{A1}Y^{B1}Z^{C1}$ предшествует моному $X^{A2}Y^{B2}Z^{C2}$ тогда и только тогда, если

$$(A_1 > A_2) \vee (A_1 = A_2) \& (B_1 > B_2) \vee (A_1 = A_2) \& (B_1 = B_2) \& (C_1 > C_2)$$.

Проверка лексикографического порядка занимает сравнительно много времени. Ее можно существенно упростить при помощи *свернутой степени* (*индекса*) монома, образуемой с использованием позиционной системы счисления: для монома со степенями (A, B, C) ставится в соответствие величина

ABC =A*100+B*10+C.

Данное соответствие является взаимно-однозначным. Обратное соответствие определяется при помощи выражений

A=E(ABC%100), B=E(ABC-A*100)%10, C=ABC-A*100-B*10.

Кроме того, введенное соответствие порождает порядок, полностью совпадающий с лексикографическим порядком

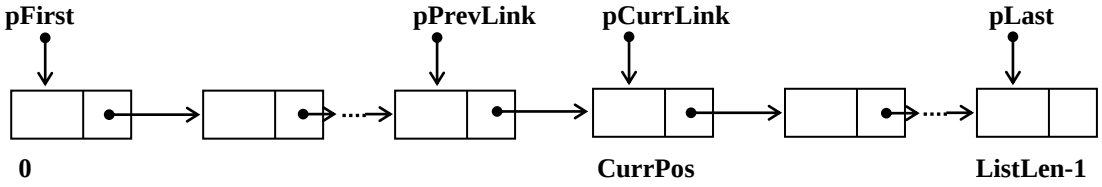
$X^{A1}Y^{B1}Z^{C1} > X^{A2}Y^{B2}Z^{C2} \Leftrightarrow ABC_1 > ABC_2$.

Выполненное обсуждение позволяет определить, что наиболее эффективным способом организации структуры хранения полиномов являются *линейный* (*односвязный*) *список*. Тем самым, в рамках лабораторной работы появляется подзадача – разработка структуры хранения

в виде линейных списков. Данная разработка должна быть выполнена в некоторой общей постановке с тем, чтобы разработанные программы работы со списками могли быть далее использованы и в других ситуациях, в которых необходимы списковые структуры хранения.

2.1.2. Структуры хранения списков

Возможный вариант общей схемы представления линейных списков может иметь следующий формат:



где

pFirst – указатель на первое звено списка;

pLast – указатель на последнее звено списка;

pCurrLink – указатель на текущее звено списка;

pPrevLink – указатель на звено, предшествующее текущему;

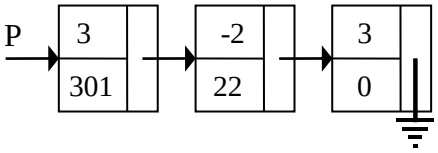
CurrPos – номер текущего звена;

ListLen – количество звеньев в списке.

Для повышения общности схемы реализации для фиксации ситуаций, в которых указатель не содержит адрес какого-либо звена списка (например, указатель следующего звена в конце списка) предлагается использовать переменную ***pStop*** вместо величины NULL (по умолчанию, эта переменная будут равняться NULL, другие значения константы будут использоваться по необходимости).

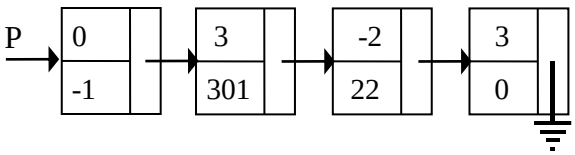
Необходимые операции над списком должны определяться при проектировании класса поддержки списков. В число необходимых операций должны входить операции создания и удаления списков, вставки и удаления звеньев, последовательного доступа к звеньям и др.

Применительно к полиномам в поле значений звеньев должны располагаться коэффициент и степень мономов – тем самым, структура хранения для приведенного ранее примера полинома имеет вид:



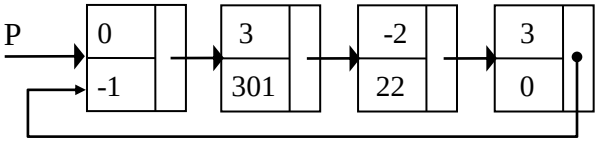
При внимательном рассмотрении задачи обработки полиномов может быть рекомендовано определенное расширение общей схемы представления линейных списков:

Для обеспечения однородности представления полиномов (в частности, для представления нулевого полинома) целесообразно ввести служебное начальное звено (*звено-заголовок*):



(звено-заголовок маркируется логически-недопустимыми значениями коэффициента и индекса монома).

Для повышения эффективности операций обработки полиномов (например, чтобы уйти от проверки нулевого указателя последнего звена списка) целесообразно использовать *циклические списки*, в которых указатель последнего звена указывает на первое звено:



2.2. Алгоритмы

2.2.1. Списки

Для работы со списками предлагается реализовать следующие операции:

- методы получения параметров состояния списка (проверка на пустоту, получение текущего количества звеньев);
- метод доступа к значению первого, текущего или последнего звена;
- методы навигации по списку (итератор);
- методы вставки перед первым, после текущего и последнего звеньев;
- методы удаления первого и текущего звена.

Состав операций может быть расширен при разработке спецификации класса для списков.

Расширение списковой структуры хранения для представления полиномов (звено-заголовок, цикличная структура списка) должно разрабатываться как производный класс от базового класса поддержки списков.

2.2.2. Полиномы

Для работы с полиномами предлагается реализовать следующие операции:

- конструкторы инициализации и копирования;
- метод присваивания;
- метод сложения полиномов.

Дополнительные операции могут быть определены при разработке спецификации класса для полиномов.

3. Разработка программного комплекса

3.1. Структура

С учетом всех перечисленных ранее требований может быть предложен следующий состав классов и отношения между этими классами (см. рис. 4):

- класс TDatValue для определения класса объектов-значений списка (абстрактный класс);
- класс TMonom для определения объектов-значений параметров монома;
- класс TRootLink для определения звеньев (элементов) списка (абстрактный класс);
- класс TDatLink для определения звеньев (элементов) списка с указателем на объект-значение;
- класс TDatList для определения линейных списков;

- класс THeadRing для определения циклических списков с заголовком;
- класс TPolinom для определения полиномов.

На рис. 4 показаны также отношения между классами: обычными стрелками показаны отношения наследования (базовый класс – производный класс), а ромбовидными стрелками – отношения ассоциации (класс-владелец – класс-компонент).

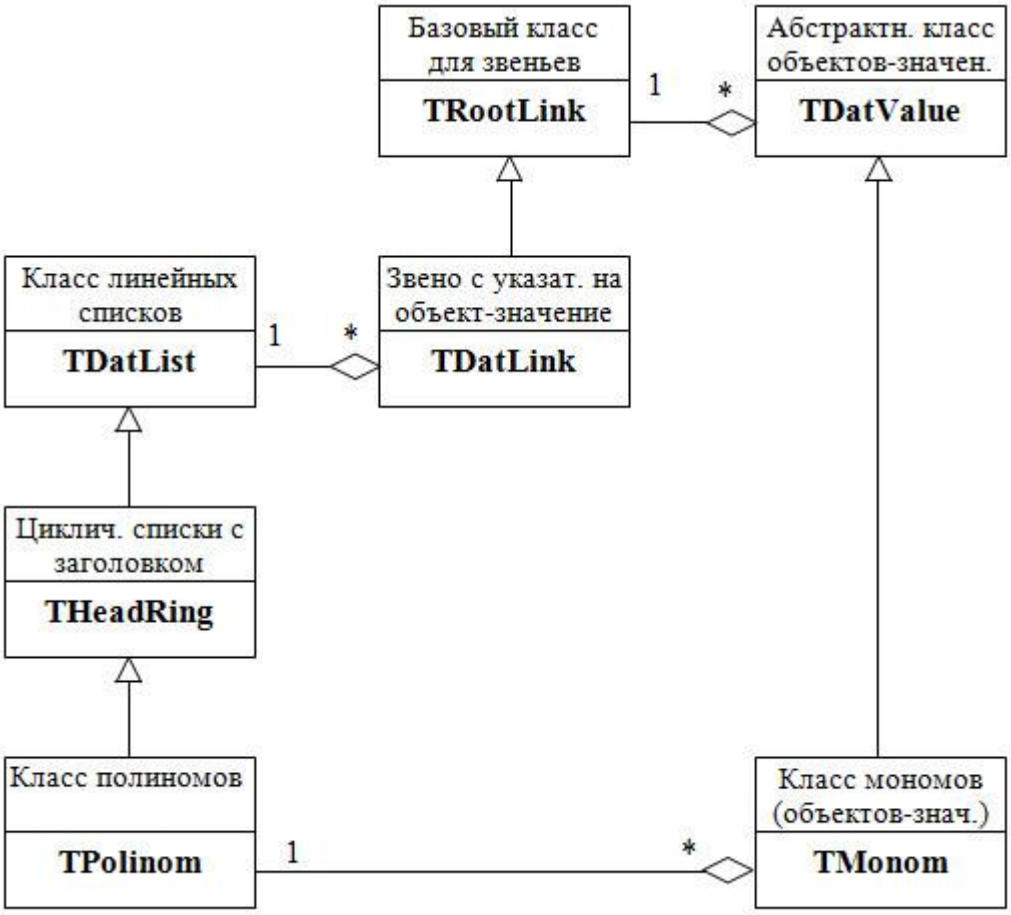


Рис. 5. Схема классов и связи между классами

В соответствии с предложенной структурой классов модульная (файловая) структура программной системы может иметь вид:

- DatValue.h – модуль, объявляющий абстрактный класс объектов-значений списка;
- RootLink.h, RootLink.cpp – модуль базового класса для звеньев (элементов) списка;
- DatLink.h, DatLink.cpp – модуль класса для звеньев (элементов) списка с указателем на объект-значение;
- DatList.h, DatList.cpp – модуль класса линейных списков;
- HeadRing.h, HeadRing.cpp – модуль класса циклических списков с заголовком;
- Monom.h, Monom.cpp – модуль класса моном;
- Polinom.h, Polinom.cpp – модуль класса полиномов;
- PolinomTestkit.cpp – модуль программы тестирования.
- UserComm.h, UserComm.cpp – модуль функций, реализующих визуальный диалоговый интерфейс для взаимодействия с пользователем.

3.2. Спецификации классов

С учетом предложенных к реализации алгоритмов можно сделать следующие объявления классов.

Абстрактный класс объектов-значений TDatValue (файл DatValue.h):

```
class TDatValue {
  public:
    virtual TDatValue * GetCopy() =0; // создание копии
   ~TDatValue() {}
};
```

Класса моном TMonom (файл Monom.h):

```
class TMonom : public TDatValue  {
    protected:
    int Coeff; // коэффициент монома
    int Index; // индекс (свертка степеней)
  public:
    TMonom ( int cval=1, int ival=0 ) {
      Coeff=cval; Index=ival;
    };
    virtual TDatValue * GetCopy(); // изготовить копию
    void SetCoeff(int cval) { Coeff=cval;   }
    int  GetCoeff(void)     { return Coeff; }
    void SetIndex(int ival) { Index=ival;   }
    int  GetIndex(void)     { return Index; }
    TMonom& operator=(const TMonom &tm) {
      Coeff=tm.Coeff; Index=tm.Index;
      return *this;
    }
    int operator==(const TMonom &tm) {
      return (Coeff==tm.Coeff) && (Index==tm.Index);
    }
    int operator<(const TMonom &tm) {
      return Index<tm.Index;
    }
    friend class TPolinom;
};
```

Базовый класс для звеньев (элементов) списка TRootLink (файл RootLink.h):

```
class TRootLink;
typedef TRootLink *PTRootLink;

class TRootLink {
  protected:
    PTRootLink  pNext;  // указатель на следующее звено
  public:
    TRootLink ( PTRootLink pN = NULL ) { pNext = pN; }
    PTRootLink  GetNextLink () { return  pNext; }
    void SetNextLink ( PTRootLink  pLink ) { pNext  = pLink; }
    void InsNextLink ( PTRootLink  pLink ) {
      PTRootLink p = pNext;  pNext  = pLink;
      if ( pLink != NULL ) pLink->pNext = p;
    }
    virtual void       SetDatValue ( PTDatValue pVal ) = 0;
    virtual PTDatValue GetDatValue ()                  = 0;

    friend class TDatList;
};
```

Класс для звеньев (элементов) списка с указателем на объект-значение TDatLink (файл DatLink.h):

```
class TDatLink;
```

```
typedef TDatLink *PTDatLink;

class TDatLink : public TRootLink {
  protected:
    PTDatValue pValue;  // указатель на объект значения
  public:
    TDatLink ( PTDatValue pVal = NULL, PTRootLink pN = NULL ) :
      TRootLink(pN) {
      pValue = pVal;
    }
    void       SetDatValue ( PTDatValue pVal ) { pValue = pVal; }
    PTDatValue GetDatValue    () { return  pValue;           }
    PTDatLink  GetNextDatLink () { return  (PTDatLink)pNext; }
    friend class TDatList;
};
```

Класс линейных списков (файл DatList.h):

```
class TDatList : {
  protected:
    PTDatLink pFirst;    // первое звено
    PTDatLink pLast;     // последнее звено
    PTDatLink pCurrLink; // текущее звено
    PTDatLink pPrevLink; // звено перед текущим
    PTDatLink pStop;     // значение указателя, означающего конец списка
    int CurrPos;         // номер текущего звена (нумерация от 0)
    int ListLen;         // количество звеньев в списке
  protected:  // методы
    PTDatLink GetLink ( PTDatValue pVal=NULL, PTDatLink pLink=NULL );
    void      DelLink ( PTDatLink pLink );   // удаление звена
  public:
    TDatList();
    ~TDatList() { DelList(); }
    // доступ
    PTDatValue GetDatValue ( TLinkPos mode = CURRENT ) const; // значение
    virtual int IsEmpty()  const { return pFirst==pStop; } // список пуст ?
    int GetListLength()    const { return ListLen; }       // к-во звеньев
    // навигация
    int SetCurrentPos ( int pos );          // установить текущее звено
    int GetCurrentPos ( void ) const;       // получить номер тек. звена
    virtual int Reset ( void );             // установить на начало списка
    virtual int IsListEnded ( void ) const; // список завершен ?
    int GoNext ( void );                    // сдвиг вправо текущего звена
                // (=1 после применения GoNext для последнего звена списка)
    // вставка звеньев
    virtual void InsFirst  ( PTDatValue pVal=NULL ); // перед первым
    virtual void InsLast   ( PTDatValue pVal=NULL ); // вставить последним
    virtual void InsCurrent( PTDatValue pVal=NULL ); // перед текущим
    // удаление звеньев
    virtual void DelFirst  ( void );    // удалить первое звено
    virtual void DelCurrent( void );    // удалить текущее звено
    virtual void DelList   ( void );    // удалить весь список
};
```

Класс циклических списков с заголовком THeadRing (файл HeadRing.h):

```
class THeadRing : public TDatList{
  protected:
    PTDatLink pHead;     // заголовок, pFirst - звено за pHead
  public:
    THeadRing ();
   ~THeadRing ();
   // вставка звеньев
    virtual void InsFirst( PTDatValue pVal=NULL ); // после заголовка
    // удаление звеньев
    virtual void DelFirst( void );                 // удалить первое звено
```

```
};
```

Класс полином TPolinom (файл Polinom.h):

```
class TPolinom : public THeadRing {
  public:
    TPolinom ( int monoms[][2]=NULL, int km=0 ); // конструктор
                    // полинома из массива «коэффициент-индекс»
    TPolinom ( const TPolinom &q);      // конструктор копирования
    PTMonom  GetMonom()  { return (PTMonom)GetDatValue(); }
    TPolinom & operator+( TPolinom &q); // сложение полиномов
    TPolinom & operator=( TPolinom &q); // присваивание
};
```

3.3. Проектирование пользовательского интерфейса

Как отмечалось ранее, важной частью лабораторной работы является разработка диалоговой управляющей программы. При минимальных требованиях разрабатывается консольное приложение с простым меню, содержащим пункты "ввод полинома", "сложение полиномов", "вывод полинома". Ввод полинома производится помономно, запрашивается коэффициент и три показателя степеней переменных. В расширенном варианте предлагается спроектировать диалоговые формы для управления работы с полиномами.

Возможный вариант диалоговой формы содержит две основные области: панель операций и панель визуализации имеющегося набора полиномов.

Панель операций содержит поля ввода полинома, удаления, присвоения и суммирования полиномов.

В качестве возможного развития диалоговой формы управления может быть, например, поддержка текстового способа задания полинома и визуализация спискового представления структуры хранения полиномов.

3.4. Этапы разработки

Одним из важных принципов определения этапности разработки программ является ориентированность на получение работоспособных версий программ, способных решать поставленную задачу (пусть даже в некотором более простом виде) на как можно более ранних этапах выполнения работ. В нашем случае для задачи обработки полиномов это означает как можно более раннюю реализацию тех или иных операций над полиномами, без выполнения полной реализации всех методов.

С учетом высказанных рекомендаций последовательность разработки программ может быть следующей:

Этап 1. Реализация списковой структуры хранения.

Этап 2. Реализация управления полиномами (в консольной или диалоговой форме).

Этап 3. Реализация операции сложения полиномов.

Этап 4. Выполнение дополнительных заданий лабораторной работы.

По-прежнему, вопросам проверки работоспособности (тестирования) программ должно уделяться большое внимание. Переход к следующим итерациям разработки следует производить только при успешном выполнении тестов; при проверке разработанных программ очередной итерации должны быть задействованы тесты всех предшествующих итераций.

Разработка диалоговой формы управления является зависимой от используемой среды программирования и требует не только знания конкретных функций, но и общего понимания, как обеспечить удобное и наглядное взаимодействие с пользователем. Выбор визуального представления диалоговой формы осуществляется обычно в ходе нескольких итераций – тем

самым, появление формы на ранних этапах разработки позволит более быстро согласовать ее визуальное представление. Кроме того, наличие средств управления в самом начале разработки позволит упростить проверку реализуемых операций обработки полиномов. И, наконец, разработанная форма может быть предъявлена как наглядный первый результат выполнения лабораторной работы.

В связи с этим, можно предложить разные варианты выполнения лабораторной работы:

1. Работа в малых группах, этапы 1 и 2 выполняют разные студенты в рамках единого проекта, третий этап выполняется совместно, дополнительные задания (этап 4) выдаются каждому индивидуально.
2. При среднем уровне подготовки студента и сжатых сроках выполнения работы, этап 2 выполняется в консольном варианте.
3. При высоком уровне подготовки студента сначала выполняется этап 2, в качестве представления полиномов используются строки и для демонстрации работоспособности создаваемой диалоговой формы используются операции ввода-вывода полиномов. После этого выполняется этап 1, разрабатывается списковая структура хранения. Этот этап требует разработки достаточно сложных программ и может потребовать достаточно большого времени. Следует тщательно определить итерации выполнения этапа и подготовить достаточный набор программ (тестов) для проверки работоспособности программ. После этого выполняются этапы 3 и 4.

Выполнение третьего этапа не должно вызывать больших затруднений, так как уже имеются диалоговые средства управления и разработаны надежно проверенные списковые структуры хранения.

3.5. Рекомендации по разработке

В результате выполнения лабораторной работы разрабатывается достаточно сложный программный комплекс. Большое внимание следует уделять проектированию диалоговой формы управления полиномами – обычные требования для интерфейса с пользователем состоят в обеспечении наглядности, понятности, дружественности и т.п.

И, конечно, принципиальное значение имеет работоспособность программного комплекса, что означает возможность решения поставленных задач и функционирование без сбоев при любых исходных данных. Здесь снова может быть рекомендовано привлечение независимых разработчиков для испытаний и проверки надежности разработанных программ.

4. Возможные темы дополнительных заданий

Для развития навыков практического программирования рекомендуются следующие направления расширения постановки задачи:

- Реализация дополнительных операций над полиномами (вычисление значения, сравнение, умножение, дифференцирование, интегрирование, …).
- Реализация операций для сохранения полиномов в текстовом файле и чтения ранее сохраненного массива полиномов.
- Реализация возможности предварительного ввода пользователем результатов выбранной операции обработки полиномов и их сравнения с результатами выполнения тех же самых операций с помощью разработанных программ (поддержка элементов учебного использования разработанного программного комплекса для обучения работы с полиномами).

При расширении разработанных программ следует обратить внимание на трудоемкость необходимой модификации существующего программного кода: количество методов, в которые требуется внести изменения; количество новых внесенных ошибок разработки и т.п.

5. Критерии оценивания выполнения лабораторной работы

Лабораторная работа считается сданной при выполнении минимального набора требований – реализованы все классы, успешно выполняются ниже приведенные тесты, диалоговая форма позволяет выполнить все реализованные операции обработки полиномов.

Набор тестов для проверки:

1. $P = 3x^5y^2z^5 - 5x^4y^3z^3 + 7x^3y^5z$
   $Q = 4x^3y^2z^6 - 6x^2yz^8$

   $P + Q = 3x^5y^2z^5 - 5x^4y^3z^3 + 7x^3y^5z + 4x^3y^2z^6 - 6x^2yz^8$

2. $P = 3x^5y^2z^5 - 5x^4y^3z^3 + 7x^3y^5z$
   $Q = 4x^7y^2z^6 - 6x^6yz^8$

   $P + Q = 4x^7y^2z^6 - 6x^6yz^8 + 3x5y2z5 - 5x^4y^3z^3 + 7x^3y^5z$

3. $P = 3x^5y^2z^5 - 5x^4y^3z^3 + 7x^3y^5z$
   $Q = 4x^5y^2z^5 + 5x^4y^3z^3$

   $P + Q = 7x^5y^2z^5 + 7x^3y^5z$

4. $P = 3x^5y^2z^5 - 5x^4y^3z^3 + 7x^7y^5z$
   $Q = 4x^6y^2z^6 - 6x^2yz^8$

   $P + Q = 7x^7y^5z + 4x^6y^2z^6 + 3x^5y^2z^5 - 5x^4y^3z^3 - 6x^2yz^8$

5. $P = 3x^5y^2z^5 - 5x^4y^3z^3 + 7x^7y^5z$
   $Q = -3x^5y^2z^5 + 5x^4y^3z^3 - 7x^7y^5z$

   $P + Q = 0$

В первом тесте для получения суммы полином *Q* пристраивается после полинома *P*. Во втором тесте для получения суммы полином *Q* располагается перед полиномом *P*. В третьем тесте при получении суммы встречаются подобные мономы: для одной пары подобных мономов коэффициенты складываются, для другой пары суммарный коэффициент обнуляется и этот моном удаляется из структуры хранения. В четвертом тесте мономы полинома P должны предварительно быть упорядочены. В пятом тесте в результате сложения двух полиномов получается нулевой полином.

6. Вопросы и задания для самоконтроля

- Зачем в списковой структуре для полиномов введено звено-заголовок?
- Почему для полиномов используются циклические списки?
- Какие дополнительные операции могут быть введены в рамках общей схемы представления линейных списков?

- Какие изменения могут потребоваться в разработанных программах при повышении максимально возможной степени при переменных полинома? При увеличении числа рассматриваемых переменных?
- Какая трудоемкость (количество необходимых операций) сложения полиномов?
- Какие правила работы со списками, реализация которых выполнена в библиотеке STL?
- Какие примеры задач можно привести, для решения которых могут быть использованы полиномы?

Лабораторная работа №6
Редактирование текстов
(иерархический связный список)

Введение

Обработка текстовой информации на компьютере широко применяется в различных областях человеческой деятельности: образование, наука, документооборот, кадровый и бухгалтерский учет и др. Вне зависимости от назначения текста типовыми операциями обработки являются создание, просмотр, редактирование и сохранение информации. В связи с тем, что объем текстовой информации может являться очень большим, для эффективного выполнения операций с ней необходимо выбрать представление текста, обеспечивающее структурирование и быстрый доступ к различным элементам текста. Так, текст можно представить в виде линейной последовательности страниц, каждая из которых есть линейная последовательность строк, которые в свою очередь являются линейными последовательностями слов. Такое представление можно осуществлять с любой степенью детализации в зависимости от особенностей прикладной задачи.

В рамках лабораторной работы рассматривается задача разработки учебного редактора текстов, в котором для представления данных используется иерархический связный список. Подобная иерархическая структура представления может применяться при компьютерной реализации математических моделей в виде деревьев и, тем самым, может иметь самое широкое применение в самых различных областях приложений.

1. Постановка учебной задачи

1.1. Основные понятия и определения

Текст – это несколько предложений, связанных друг с другом по смыслу и грамматически. В рамках лабораторной работы в качестве примеров текстов рассматриваются тексты программ.

Редактор текстов – программный комплекс, обеспечивающий выполнение операций обработки текста: создание, просмотр, редактирование и сохранение. Специализированные редакторы текстов могут поддерживать выполнение дополнительных операций (например, проверку синтаксиса или контекстный поиск).

Иерархический связный список – это многосвязный список, в котором на каждое звено имеется ровно один указатель, а каждое звено содержит два указателя (один на следующее звено в том же уровне, другой на звено в нижерасположенном уровне).

1.2. Требования к лабораторной работе

В рамках лабораторной работы ставится задача разработки учебного редактора текстов с поддержкой следующих операций:

- выбор текста для редактирования (или создание нового текста);
- демонстрация текста на экране дисплея;
- поддержка средств указания элементов (уровней) текста;
- вставка, удаление и замена строк текста;

- запись подготовленного текста в файл.

При выполнении операций чтения (при выборе для редактирования уже существующего текста) и записи редактор должен использовать стандартный формат, принятый в файловой системе для представления текстовых файлов, обеспечивая тем самым совместимость учебного редактора текстов и существующего программного обеспечения.

Выполнение данной лабораторной работы может потребовать проведения достаточно сложных программных работ. Для их успешной реализации создание каждого варианта редактора текста может поручаться группе студентов из 3-4 человек. Для лучшей координации выполняемых работ в каждой группе может быть выделен ответственный за разработку (*главный программист*), в задачу которого входило бы согласование заданий на разработку, распределение работ между участниками разработки, согласование спецификаций и т.п. Отдельный разработчик (*тестировщик*) может отвечать за тестирование разрабатываемого кода. Еще один разработчик (*технический писатель*) может быть ориентирован на подготовку документации и презентаций по учебному редактору.

Важной частью разработки учебного редактора является реализация диалоговой программы для взаимодействия с пользователем редактора. Данная часть разработки разработки также может поручена отдельному участнику группы разработчиков. С другой стороны, возможный вариант выполнения лабораторной работы состоять в использовании уже имеющихся (ранее разработанных) диалоговых программ. В этом случае, группе разработчиков следует освоить правила использования (API) переданных для использования программ.

Следует отметить важность выделенных ролей участников группы разработчиков. От тестировщика зависит безошибочность (надежность) разработанного редактора текста. От разработчика диалоговой программы зависит удобство использования редактора. И наконец, от технического редактора зависит легкость сопровождения и развития редактора и возможность более широкого привлечения потенциальных потребителей разработанных программ.

1.3. Условия и ограничения

В рамках выполнения данной лабораторной работы могут быть использованы следующие основные допущения:

- При планировании структуры текста в качестве самого нижнего уровня можно рассматривать уровень строк.
- В качестве тестовых текстов можно рассматривать текстовые файлы программы.

2. Метод решения

2.1. Структуры данных

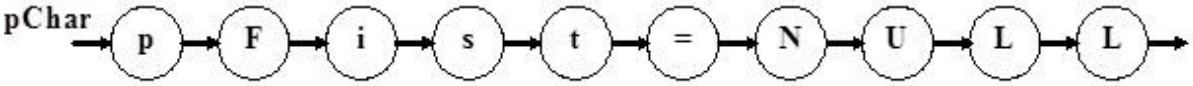
При выборе модели представления текста необходимо учитывать, какие операции будут выполняться. Можно рассмотреть несколько моделей представления текста: текст можно рассматривать как линейную последовательность символов, как линейную последовательность строк, как линейную последовательность страниц и т.д.

Так, для следующего примера текста:

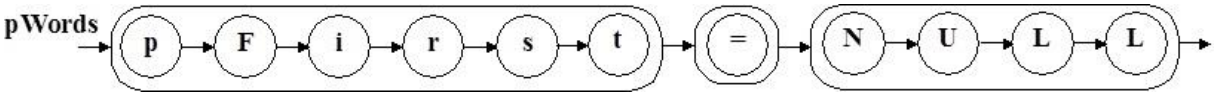
```
pFirst=NULL

ListLen=0
```

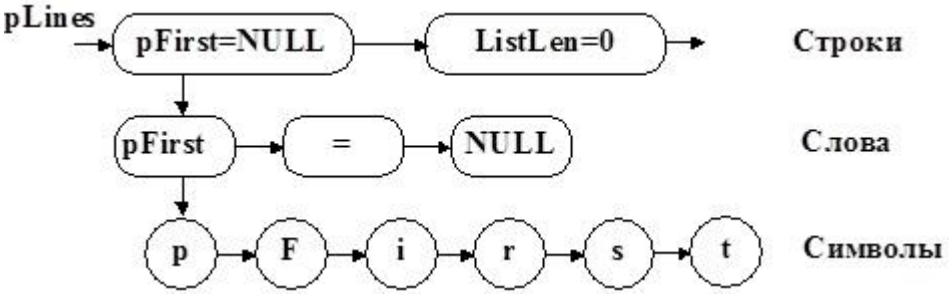
представление текста в виде последовательности символов имеет вид:



а представление в виде последовательности слов может иметь следующий формат:



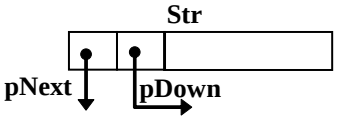
Единство возможных моделей текста можно обеспечить, если при введении каждого более высокого уровня использовать все ранее введенные (ниже расположенные) структуры представления. Такой подход может быть реализован с использованием *иерархических связных списков*. Звенья такого списка содержат по два указателя: один используется для организации отношения следования по элементам текста одного и того же уровня, а другой – для указания отношения следования по иерархии уровней представления текста (от более высокого уровня к ниже расположенному).



В силу необходимости выполнения операций редактирования текст является динамической структурой данных. В соотвествтии с определением, *динамическая структура* есть математическая структура, которой соответствует частично-упорядоченное (по включению) базовое множество *M*, элементы которого являются структурами данных. При этом отношения включения индуцируются операциями преобразования структуры данных.

Необходимым условием применения связного списка как структуры хранения текста является возможность нахождения такого унифицированного типа звеньев, который можно было бы использовать на любых уровнях представления текста, так как реализация своего типа звеньев для каждого уровня приведет к увеличению трудоемкости управления памятью и дублированию программ обработки. Возможное решение при выборе структуры звеньев может состоять в следующем:

- каждое звено структуры хранения содержит два поля указателей и поле для хранения данных;
- нижний уровень иерархической структуры звеньев ограничивается уровнем строк, а не символов. Это повышает эффективность использования памяти, так как в случае отдельного уровня для хранения символов, затраты на хранение служебной информации (указатели) в несколько раз превышают затраты на хранение самих данных.При использовании строк в качестве атомарных элементов поле для хранения данных является массивом символов заданного размера;



- количество уровней и количество звеньев на каждом уровне может быть произвольным;

- при использовании звена на атомарном уровне поле указателя на нижний уровень структуры устанавливается равным значению NULL;
- при использовании звена на структурных уровнях представления текста поле для хранения строки текста используется для размещения наименования соответствующего элемента текста;
- количество используемых уровней представления в разных фрагментах текста может быть различным.

При выбранном способе представления для следующего примера текста:

```
Раздел 2
     2.1. Полиномы
       1.Определение
       2.Структура
     2.2. Тексты
       1.Определение
       2.Структура
```

структура хранения текста будет иметь вид:

2.2. Алгоритмы

2.2.1. Навигация

Для индикации позиции в тексте, относительно которой выполняются операции перемещения по тексту, вводится понятие *текущей строки*.

Для перемещения по тексту предлагается реализовать следующие операции:

- переход к первой строке текста;
- переход к следующему элементу в том же уровне;
- переход к элементу в нижерасположенном уровне;
- переход к предыдущей позиции в тексте.

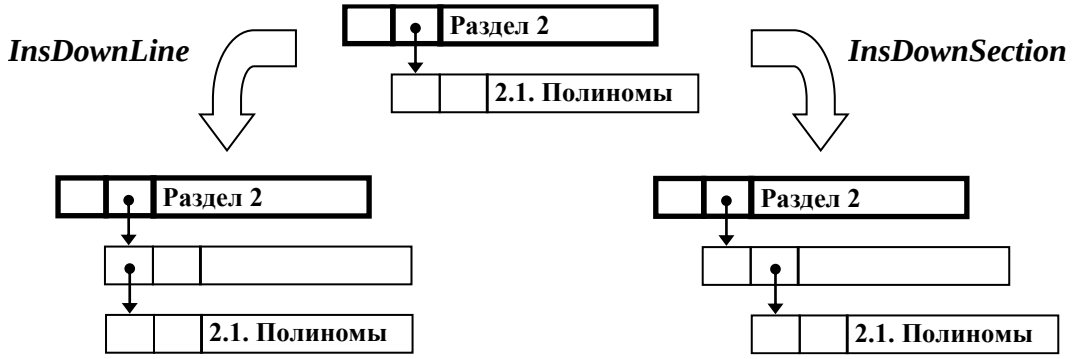
Разработанная структура хранения текста обеспечивает возможность передвижения по структуре только в направлении следующих или нижерасположенных элементов. Движение по элементам текста в обратном направлении возможно только при условии, если адреса звеньев, на которые необходимо переместиться, были каким-либо способом запомнены ранее. Возможный способ организации запоминания этих адресов состоит в использовании стека.

2.2.2. Доступ

Необходимо реализовать операции получения и замены текста текущей строки.

2.2.3. Модификация

Необходимо реализовать операции вставки и удаления элементов текста. Операции должны применяться для текущего и нижерасположенного уровней. Для операции вставки необходимо разработать вариант образования новых подуровней – так, например, выполнение операции вставки для подуровня может быть представлено следующим образом



Для операции удаления должны быть предусмотрены варианты удаления, как отдельных строк, так и всего содержимого подуровней текста.

2.2.4. Обход текста

Для последовательного доступа ко всем элемента текста (например, для печати текста) необходимо совершить обход текста.

Схема обхода может состоять в следующем. При начале обхода следует перейти от начала текста (корня дерева) до атомарного уровня по указателям нижерасположенных уровней, запоминая при этом все пройденные звенья в стеке. После обработки найденной строки, далее следует переходить по строкам того же уровня, до последней строки в этом уровне. После обхода уровня текущего уровня необходимо извлечь звено из стека и повторить всю выше приведенную последовательность действий. Обход текста будет завершен, когда стек пуст. Данная схема обхода представляет вариант TDN (top – down – next, вершина - нижний уровень – следующий элемент).

При запоминании иерархически представленного текста в текстовом файле необходимо обеспечить запоминание структуры текста. Возможный вариант состоит в использовании специальных символов для выделения моментов начала и завершения уровней текста (в качестве таких служебных символов можно использовать, например, знаки фигурных скобок «{» и «}»).

При таком подходе, общая схема алгоритма восстановления текста из текстового файла, будет иметь следующий вид:

```
ВВОД_ТЕКСТА
Повторить
  • ввод строки
  • ЕСЛИ '{' ТО завершить
  • ЕСЛИ '}' ТО выполнить рекурсивно ВВОД_ТЕКСТА
  • Добавить строку в том же уровне
```

2.2.5. Итератор

Как и при работе с линейными списками, использование итератора позволяет упростить реализацию операций с текстом и обеспечивает унифицированный способ обработки элементов структуры данных. Итератор включает следующие методы:

- инициализация (установка на корневое звено)

В этом методе происходит опустошение стека, после чего в стек помещаются указатель на корневое звено, указатели на следующее звено и следующее в подуровне.

- проверка завершения текста

  Если стек пуст, то текст завершен.

- переход к следующему звену

  В стек помещаются указатели на следующее звено и на звено, следующее в подуровне.

2.2.6. Копирование текста

Для копирования текста необходимо осуществить обход текста. Так как структура текста является нелинейной, то копирование производится за два прохода, при этом для навигации по исходному тексту и тексту копии используется один объединенный стек.

Первый проход производится при подъеме на строку из подуровня – для текущей строки выполняется:

- создание копии звена;
- заполнение в звене-копии поля указателя подуровня **pDown** (подуровень уже скопирован);
- запись в звене-копии в поле данных значения “Copy”, используемое как маркер для распознавания звена при попадании на него при втором проходе; предполагается, что в тексте данный маркер не встречается;
- запись в звене-копии в поле указателя следующего звена **pNext** указателя на звено-оригинал (для возможности последующего копирования текста исходной строки);
- запись указателя на звено-копию в стек.

Второй проход производится при извлечении звена-копии из стека (распознается по маркеру “Copy”)– в этом случае необходимо выполнить:

- заполнение в звене-копии полей данных и указателя следующего звена;
- указатель на звено-копию запоминается в служебной переменной.

2.2.7. Повторное использование памяти (сборка мусора)

При удалении разделов текста для освобождения звеньев следует учитывать следующие моменты:

обход всех звеньев удаляемого текста может потребовать длительного времени;

при допущении множественности ссылок на разделы текста (для устранения дублирования одинаковых частей) удаляемый текст нельзя исключить, так как он может быть задействован в других фрагментах текста.

Для решения этих проблем можно не освобождать память в момент удаления, а фиксировать удаление текста установкой соответствующего указателя в состояние NULL. При такой реализации операции удаления может возникнуть ситуация, когда в памяти, отведенной под хранение текста, будут присутствовать звенья, на которые нет указателей в тексте. Таким образом, эти звенья не могут быть возвращены системе для последующего использования, а память считается занятой (возникает так называемая *утечка памяти*). Такие звенья называются *«мусором»*.

Наличие «мусора» в системе может быть допустимым, если имеющейся свободной памяти достаточно для работы программ. В случае нехватки памяти необходимо выполнить *«сборку мусора» (garbage collection)*.

Для осуществления данного подхода необходимо программно реализовать систему управления памятью для представления текстов. Для данной системы управления память должна выделяться полностью при начале работы программы, размер выделяемой памяти может определяться как параметр системы. Для фиксации состояния памяти в классе **TTextLink** может быть определена статическая переменная **MemHeader** класс **TTextMem**:

```
class TTextMem {
  PTTextLink pFirst;      // первое звено
  PTTextLink pLast;       // последнее звено
  PTTextLink pFree;       // первое свободное звено
};
```

Для начального выделения и форматирования памяти используется статический метод **InitMemSystem** класса **TTextLink**. В этом методе выделяется память для хранения текстов. Указатель **pFirst** устанавливается на начало этого массива (после приведения типа к типу указателя на звено), указатель pLast на последний элемент массива. Далее этот массив размечается как список свободных звеньев, в самом начале работы список свободных звеньев может быть упорядочен по памяти.

Для выделения памяти под звено перегружается оператор **new**, который выделяет новое звено из списка свободных звеньев. При освобождении звена в перегруженном операторе **delete** происходит возвращение памяти в список свободных звеньев.

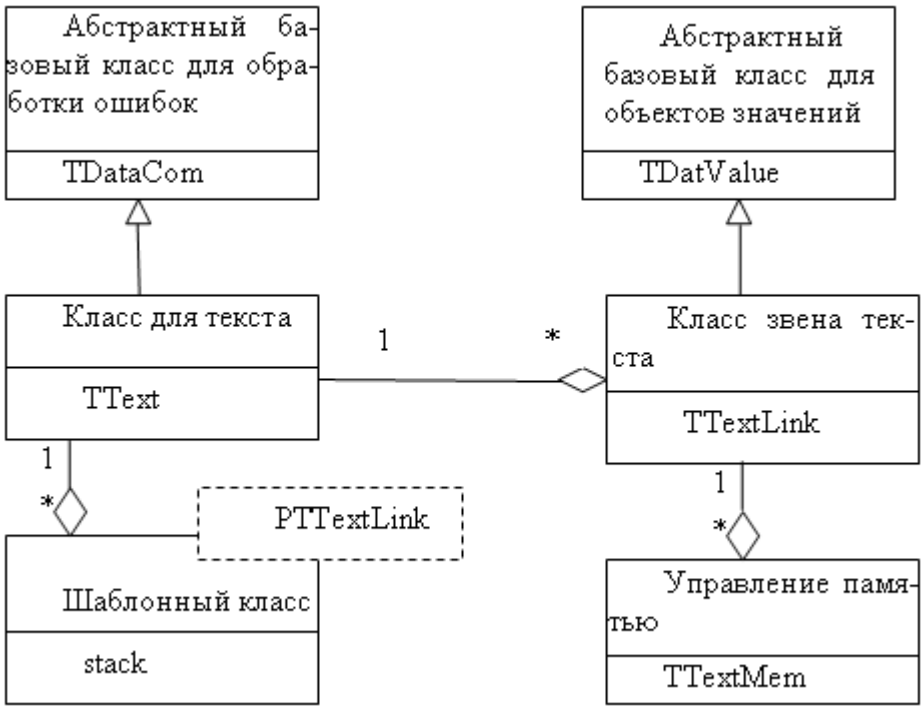
Алгоритм «сборки мусора» состоит из трех этапов. На первом этапе происходит обход текста и маркирование звеньев текста специальными символами (например, «&&&»). На втором этапе происходит обход и маркирование списка свободных звеньев. На третьем этапе происходит проход по всему непрерывному участку памяти как по непрерывному набору звеньев. Если звено промаркировано, то маркер снимается. В противном случае, найдено звено, на которое нет ссылок («мусор»), и это звено возвращается в список свободных звеньев.

3. Разработка программного комплекса

3.1. Структура

С учетом рассмотернных выше предложений к реализации целесообразной представляется следующая модульная структура программы:

- TTextLink.h, TTextLink.cpp – модуль с классом для звена текста;
- TText.h, TText.cpp – модуль с классом, реализующим операции над текстом;
- TTextViewer.h, TTextViewer.cpp – модуль с классом, реализующим визуализацию текста;
- TTextUsercom.h, TTextUsercom.cpp – модуль с классом, обеспечивающий диалоговое взаимодействие с пользователем;
- TTextTestkit.cpp – модуль программы тестирования.



3.2. Спецификации классов

С учетом предложенных к реализации алгоритмов можно сделать следующие объявления классов.

Класс **TTextMem** для системы управления памятью и класс **TTextLink** для строк текста (файл **TTextLink.h**):

```
class TTextLink;
typedef TTextLink *PTTextLink;
typedef char TStr[TextLineLength];
class TTextMem {
  PTTextLink pFirst;     // указатель на первое звено
  PTTextLink pLast;      // указатель на последнее звено
  PTTextLink pFree;      // указатель на первое свободное звено
  friend class TTextLink;
};
typedef TTextMem *PTTextMem;

class TTextLink : public TDatValue {
protected:
 TStr Str;  // поле для хранения строки текста
 PTTextLink pNext, pDown;  // указатели по тек. уровень и на подуровень
 static TTextMem MemHeader; // система управления памятью
public:
 static void InitMemSystem (int size=MemSize); // инициализация памяти
 static void PrintFreeLink (void);  // печать свободных звеньев
 void * operator new (size_t size); // выделение звена
 void operator delete (void *pM);   // освобождение звена
 static void MemCleaner (const TText &txt); // сборка мусора
 TTextLink (Tstr s = NULL, PTTextLink pn = NULL, PTTextLink pd = NULL){
  pNext = pn; pDown = pd;
  if (s != NULL) strcpy(Str,s); else Str[0]='\0';
 }
 ~TTextLink() {}
 int IsAtom () {return pDown == NULL;} // проверка атомарности звена
```

```
 PTTextLink GetNext() {return pNext;}
 PTTextLink GetDown() {return pDown;}
 PTDatValue GetCopy() {return new TTextLink(Str,pNext,pDown);}
protected:
 virtual void Print (ostream &os) {os << Str;}
 friend class TText;
};
```

Класс для представления иерархического связного списка (**TText.h**):

```
class TText : public TDataCom {
protected:
 PTTextLink pFirst;      // указатель корня дерева
 PTTextLink pCurrent;      // указатель текущей строки
 stack< PTTextLink > Path; // стек траектории движения по тексту
 stack< PTTextLink > St;   // стек для итератора
 PTTextLink GetFirstAtom (PTTextLink pl); // поиск первого атома
 void PrintText (PTTextLink ptl);         // печать текста со звена ptl
 PTTextLink ReadText (ifstream &TxtFile); //чтение текста из файла
public:
 TText (PTTextLink pl = NULL);
 ~TText () {pFirst =NULL;}
 PTText getCopy();
// навигация
 int GoFirstLink (void); // переход к первой строке
 int GoDownLink (void);  // переход к следующей строке по Down
 int GoNextLink (void);  // переход к следующей строке по Next
 int GoPrevLink (void);  // переход к предыдущей позиции в тексте
// доступ
 string GetLine(void);   // чтение текущей строки
 void SetLine ( string s); // замена текущей строки
// модификация
 void InsDownLine (string s);    // вставка строки в подуровень
 void InsDownSection (string s); // вставка раздела в подуровень
 void InsNextLine (string s);    // вставка строки в том же уровне
 void InsNextSection (string s); // вставка раздела в том же уровне
 void DelDownLine (void);        // удаление строки в подуровне
 void DelDownSection (void);     // удаление раздела в подуровне
 void DelNextLine (void);        // удаление строки в том же уровне
 void DelNextSection (void);     // удаление раздела в том же уровне
// итератор
 int Reset (void);              // установить на первую звапись
 int IsTextEnded (void) const;  // текст завершен?
 int GoNext (void);             // переход к следующей записи
//работа с файлами
 void Read (char * pFileName);  // ввод текста из файла
 void Write (char * pFileName); // вывод текста в файл
//печать
 void Print (void);             // печать текста
};
```

3.3. Этапы разработки

Выполнение лабораторной работы должно быть разделено на несколько этапов. Каждый из этапов должен иметь достаточно небольшую длительность, при этом должна обеспечиваться возможность решения поставленной прикладной задачи (в той или иной ограниченной постановке) на как можно более ранних этапах выполнения работ.

С учетом высказанных рекомендаций последовательность разработки программ может быть следующей:

Этап 1. Реализация программ для работы с текстами простой структуры (например, с текстами, состоящими из последовательности строк). Выполнение этапа может быть разделено на несколько итераций:

- Ввод линейного текста из текстового файла и вывода текста на экран (в наиболее простой форме в режиме консоли);
- Реализация навигации по линейному тексту и вывод содержимого текущей строки текста на экран;
- Реализация операции замены содержимого текущей строки;
- Вывод линейного текста в текстовый файл.

Результаты каждой итерации должны быть протестированы – переход к следующей итерации разработки должен осуществляться только после успешного выполнения всех запланированных тестов.

Этап 2. Реализация программ для работы с иерархическими представленными текстами. При выполнении этапа возможно выделение следующих итераций:

- Ввод с иерархически-представленного текста из текстового файла и вывода текста на экран (вывод может быть по-прежнему в режиме консоли, но с визуальным представлением иерархической структуры текста за счет использования, например, отступов для строк нижерасположенных уровней);
- Реализация навигации по иерархически-представленному тексту и вывод содержимого текущей строки текста на экран;
- Вывод иерархически-представленного текста в текстовый файл.

Одновременно с реализацией программ для иерархически-представленных текстов могут выполняться работы по разработке начального варианта диалоговых форм управления текстами (ввод текста из файла, визуализация текста в рабочем окне редактора текста, навигация по тексту с выделением текущей позиции текста, сохранение текста в текстовом файле).

Этап 3. Реализация операций изменения иерархической структуры текста – в числе возможных итераций:

- Вставка и удаление строк на текущем уровне текста; при вставке строки текста должны быть предусмотрена возможность создания подуровней текста; для операции удаления должен быть разработан вариант удаления разделов текста;
- Вставка и удаление строк для нижерасположенного уровня текущей строки текста; при вставке строки текста должны быть предусмотрена возможность создания подуровней текста; для операции удаления должен быть разработан вариант удаления разделов текста.

Этап должен включать развитие диалоговой формы управления.

Этап 4. Реализация итератора для иерархически-представленных текстов.

Этап 5. Реализация операции копирования текста. Для проверки правильности выполнения операции копирования рекомендуется подготовить программу автоматической проверки (исходный текст вводится из файла, затем создается копия текста, после чего текст запоминается еще в одном текстом файле, в завершении исходный и результирующий файлы проверяются на идентичность).

Этап 6. Реализация системы управления памятью для иерархически-представленных текстов. При выполнении этапа могут быть выделены итерации:

- Инициализация системы управления памятью;
- Реализация маркировки звеньев;
- Реализация сборки мусора.

Результаты выполнения данного этапа требуют проведения самого тщательного тестирования, поскольку возможные ошибки могут проявляться только после некоторого длительного времени работы с текстами; ошибочные ситуации могут возникать не при каждом выполнении программ.

3.4. Рекомендации по разработке

Набор программ для работы с иерархически-представленными текстами представляет собой достаточно сложную программную систему. При выполнении лабораторной работы следует уделять особое внимание технологическим аспектам разработки программного обеспечения: этапность разработки, тщательное тестирование, командную методику выполнения работ, обеспечение простого и понятного интерфейса, использование инструментов поддержки процесса разработки (система контроля версий, система поддержки тестирования и др.).

4. Возможные темы дополнительных заданий

Для развития навыков практического программирования рекомендуются следующие направления расширения постановки задачи:

- Реализовать операцию отмены выполненных операций изменения текста.
- Реализовать операцию визуализации текста с заданной глубиной просмотра нижерасположенных уровней текста.
- Реализовать операции изменения структуры текста.
- Реализовать операции поиска фрагментов текста по образцу.
- Реализовать операцию сопоставления текстов между собой.
- Реализовать операцию слияния текстов.
- Разработать программу вычисления частотного повторения слов текста.

5. Критерии оценивания выполнения лабораторной работы

Лабораторная работа считается сданной при выполнении следующего минимального набора требований:

- Реализованы все классы, в консольном приложении происходит демонстрация методов работы с текстом.
- Обеспечивается ввод текста из тестового файла и вывод текста в текстовый файл.
- Реализованы операции вставки и удаления строк для текущего уровня и нижерасположенного уровня текущей строки (без работы с разделами текста).
- Реализован итератор для иерархически-представленного текста.

В более расширенном варианте выполнения лабораторной работы должен быть реализован визуальный пользовательский интерфейс, разработана программа копирования текста, реализована сборка мусора.

6. Вопросы и задания для самоконтроля

- Какие возможные модели можно использовать для представления текста?

- В чем состоит модель иерархического представления текста?
- Какая структура хранения используется для иерархически представленного текста?
- Какие схемы обхода возможны для иерархически представленного текста?
- Почему при удалении строк текста допускается возможность образования «мусора»?
- В чем состоит процедура сборки мусора?

Лабораторная работа №7
Обработка геометрических объектов на ЭВМ (плексы)

Введение

Тема лабораторной работы относится к важной области задач обработки графических данных, потребность в анализе которых встречается при рассмотрении многих научно-технических и производственных задач. При построении различных программных систем общего и специального назначения (таких, например, как системы автоматизированного проектирования) могут оказаться необходимыми средства, обеспечивающие хранение графической информации на ЭВМ, возможность редактирования этой информации и использования ее в расчетах, а также получение изображений на экране дисплея или на бумаге.

Лабораторная работа является введением в проблематику *структурного* (*векторного*) представления графической информации, когда изображения представляются не в виде растровых данных, а формируются в виде набора графических элементов. Такой подход позволяет существенно повысить эффективность анализа и обработки графических данных.

Дополнительный учебный эффект при выполнении лабораторной работы состоит в изучении способов представления на ЭВМ математических моделей на основе сетевых структур данных.

1. Постановка учебной задачи

1.1. Основные понятия и определения

Под *чертежом* в рамках лабораторной работы называют графическое изображение моделируемого объекта, характеризующее его форму и размеры и выполненное по определенным правилам проектирования с применением общепринятых соглашений по изображению и обозначению графических элементам.

По чертежам изготовляют детали и производят сборку узлов, машин и конструкций. Чертеж на производстве – это основной технический документ, в котором указаны необходимые сведения о деталях, их материале, термической обработке, технические условия на изготовление. По нему проверяют качество обработки на всех стадиях технологического процесса. Различают чертеж детали, сборочный чертеж, чертеж общего вида и др.

1.2. Требования к лабораторной работе

В рамках лабораторной работы ставится задача разработки программного комплекса для представления, редактировании отображения на экране дисплея конкретного типа графических данных – *чертежей*, образуемых из ограниченного набора различных геометрических элементов (точек, линий, окружностей и т.п.). Разрабатываемый комплекс должен обеспечивать:

- представление чертежей в памяти ЭВМ;
- демонстрацию хранимых чертежей на экране дисплея;
- редактирование чертежей (вставку и удаление линий);

- запись (чтение) данных о чертежах в файлах.

Данная лабораторная работа является одной из завершающих работ лабораторного практикума учебного курса по методам программирования. Тем самым, постановка лабораторной работы должна иметь повышенную сложность с тем, чтобы для выполнения работы потребовалось бы участие нескольких разработчиков. Освоение принципов командной работы является одной из ключевых учебных задач лабораторной работы.

Для успешного выполнения лабораторной работы ее реализация должная поручаться группе студентов из 3-4 человек. Для лучшей координации выполняемых работ в каждой группе может быть выделен ответственный за разработку (*главный программист*), в задачу которого входило бы согласование заданий на разработку, распределение работ между участниками разработки, согласование спецификаций и т.п. Отдельный разработчик (*тестировщик*) может отвечать за тестирование разрабатываемого кода. Еще один разработчик (*технический писатель*) может быть ориентирован на подготовку документации и презентаций по программной системе.

Важной частью разработки программной системы является реализация диалоговой программы для взаимодействия с пользователем. Данная часть разработки также может быть поручена отдельному участнику группы разработчиков. С другой стороны, возможный вариант выполнения лабораторной работы состоять в использовании уже имеющихся (ранее разработанных) диалоговых программ. В этом случае, группе разработчиков следует освоить правила использования (API) переданных для использования программ.

Следует отметить важность выделенных ролей участников группы разработчиков. От тестировщика зависит безошибочность (надежность) разработанного кода программ. От разработчика диалоговой программы зависит удобство использования программной системы. И наконец, от технического редактора зависит легкость сопровождения и развития проекта и возможность более широкого привлечения потенциальных потребителей разработанных программ.

1.3. Условия и ограничения

В рамках выполнения данной лабораторной работы могут быть использованы следующие основные допущения:

- В качестве чертежа будем рассматривать плоский геометрический объект, состоящий из отрезков прямых (линий) и граничных точек этих линий (представление других геометрических элементов может рассматриваться как тема дополнительных заданий).
- Набор линий, образующих чертеж, должен быть связным, т.е. любая линия чертежа должна иметь общую точку хотя бы с одной другой линией чертежа.

Пример возможного чертежа представлен на рис. 6.

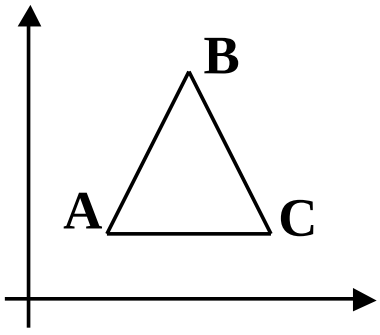


Рис. 6. Пример чертежа, состоящего из трех линий и трех точек.

2. Метод решения

2.1. Структуры данных

Каждый чертеж может быть представлен в виде множества базовых геометрических объектов – точек, линий, окружностей и т.д. Информационное описание объектов включает в себя параметры фигуры (координаты, размер, радиус и другие – с различной степенью детализации в зависимости от требований, предъявляемых к чертежу в данной предметной области). В общем случае для описания фигуры необходимо предусмотреть хранение координат некоторой опорной точки.

Операции обработки геометрических объектов состоят в задании и изменении параметров, а также визуализации (например, на экране дисплея) фигур.

Для обеспечения возможности динамической визуализации геометрических объектов в лабораторной работе вводится тип данных, значения которого вычисляются в соответствии с задаваемым формульным выражением (класс TFormValue).

Возможная **схема иерархии** классов для представления базовых геометрических элементов может состоять в следующем (см. рис. 7).

Наиболее общие свойства геометрических фигур и методы их разработки выделяются в абстрактный базовый класс (TChartRoot). От него наследуются классы для представления точки (TChartPoint), на основе которого реализуется класс для представления линии (TChartLine) и классы для других геометрических элементов.

Точка определяется координатами на плоскости, значение координат определяются при помощи формульных объектов класса TFormValue. При формировании чертежа следует обратить внимание, что одна и та же точка в чертеже может присутствовать несколько раз – в этом случае необходимо обеспечить однократное представление точки без дублирования (для обеспечения корректной работы операций вставки и удаления).

Для представления линии в наиболее простом варианте необходимо хранить начальную и конечную точки. В более общем случае при формировании чертежа в объекте линии могут храниться не координаты точек, а указатели на другие линии чертежа. Данная ситуация допускается при выполнении следующих условий:

Начальная точка текущей линии может определяться при помощи указателя на линию, конечная точка которой является начальной точкой текущей линии.

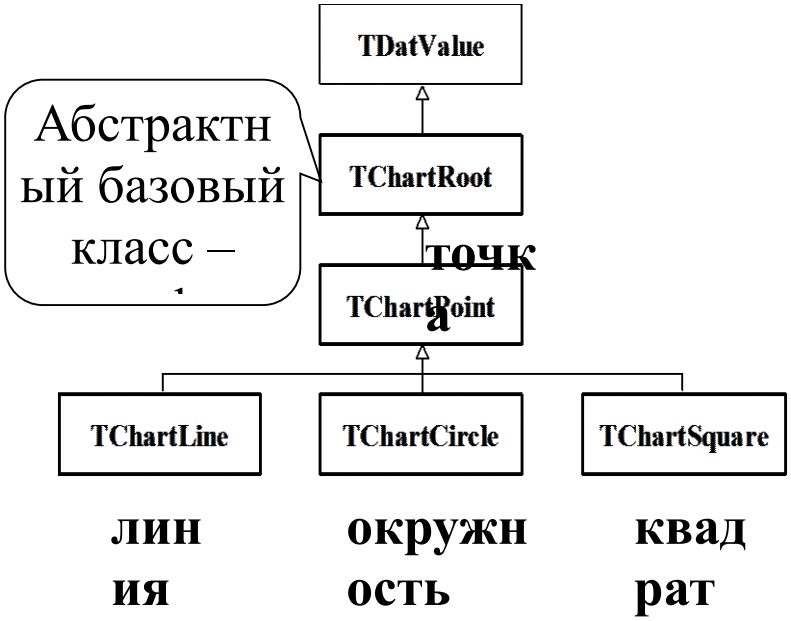


Рис. 7. Схема наследования классов для представления базовых геометрических элементов

Конечная точка текущей линии может определяться при помощи указателя на линию, конечная точка которой является конечной точкой текущей линии.

Наряду с базовыми геометрическими элементами, требующих для своего представления разработки отдельных классов, в лабораторной работе предлагается реализовать несколько различных способов формирования с*оставных геометрических объектов*, образуемых на основе уже существующих геометрических фигур (как базовых, так и составных) и рассматриваемых при выполнении операции обработки как единое целое. Возможные способы формирования составных объектов состоят в следующем:

- *Группирование,* когда составной объект представляет собой набор уже существующих (как базовых, так и составных) геометрических фигур – подобной операцией является, например, группирование в графическом редакторе системы Word,
- *Конструирование*, когда составной объект определяется на основе уже существующих геометрических фигур (например, ломаная может быть определена через набор конечных точек составляющих отрезков), конструирование приводит к созданию новой сложной геометрической фигуры, которая не может быть разделена на составные геометрические элементы,
- *Комбинирование,* когда составной объект формируется при помощи сборки существующих объектов (так, например, чертеж может быть образован из точек и линий), в рамках сконструированного объекта входящие в состав элементы по-прежнему могут обрабатываться как отдельные геометрические фигуры.

Для представления данных, описывающих структуру чертежа, предлагается использовать структуру хранения типа *плекс*. Плекс есть структура хранения, включающая звенья разных типов, отношения между которыми задаются при помощи сцепления. Таким образом, плекс является разновидностью многосвязного списка и используется для хранения *сетевых моделей данных*.

Основу плексов для представления чертежей, состоящих из линий и точек, составляют вершины (узлы), в каждой из которых располагается информация о той или иной линии чертежа (имя линия, координаты начальной и конечной точек[6]).

Как пример разработанной структуры хранения, на рис. 8 приведен плекс для чертежа из рис. 1.

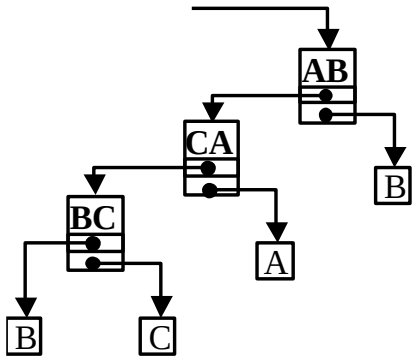


Рис. 8. Пример плекса для чертежа из рис. 6

[6] Как отмечалось ранее, вместо координат точек могут использоваться указатели на линии (при выполнении условия связности)

2.2. Алгоритмы

В качестве основных алгоритмов обработки плексов в рамках лабораторной работы должны быть рассмотрены операции обхода плексов, необходимых, например, при отрисовке чертежа.

Реализация алгоритмов обхода плексов может быть выполнена поэтапно: на первом этапе можно реализовать частный алгоритм обхода плексов без подплексов (как, например, на рис. 3), и только затем приступить к разработке общего алгоритма обхода.

2.2.1. Обход плекса без подплексов

Плексы без подплексов соответствуют представлению чертежей, которые можно нарисовать без отрыва карандаша от бумаги. Пример такого объекта - треугольник представлен на рис. 1. Соответствующий плекс без подплексов приведен на рис. 3.

Для того чтобы нарисовать линию, необходимо знать ее начальную и конечную точку. Так как в плексе допускается вместо указателя точки хранить указатель линии, то необходимо найти первую линию, у которой указатель начала – точка. В приведенном примере на рис. 3 это линия BC. Тогда конечная точка этой линии будет начальной точкой предыдущей линии (в примере – CA). То есть, на первом этапе необходимо отыскать линию, у которой указатель начала – точка, при этом указатели на все пройденные линии необходимо запомнить в стеке. На втором этапе из стека извлекаются линии, для которых в качестве начальной точки выступает конечная точка обработанной линии. Процесс заканчивается, когда в стеке больше не содержится указателей на линию.

2.2.2. Обход плекса с подплексами

Плекс с подплексами служит для представления чертежей, которые нельзя нарисовать без отрыва карандаша от бумаги. Пример такого чертежа представлен на рис. 9, возможная структура плекса для которого показана на рис. 10.

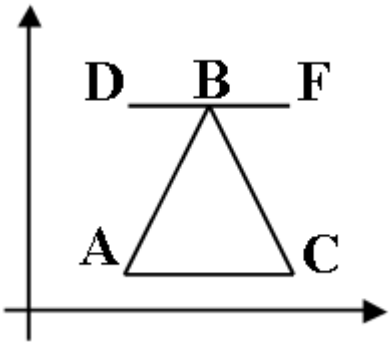


Рис. 9. Пример чертежа, для представления которого требуется плекс с подплексами (верхняя горизонтальная линия состоит из двух отзков DB и BF).

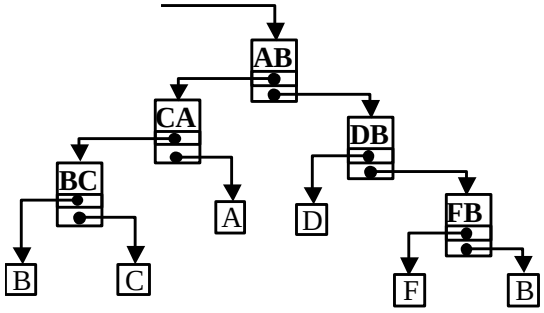


Рис. 10. Пример плекса с подплексами для чертежа из рис. 9

Для таких плексов указатель на конечную точку может также указывать на линию, то есть конечная точка линии верхнего узла плекса может являться конечной точкой линии правого подплекса. Для обхода плекса с подплексами можно предложить рекурсивный и нерекурсивный варианты. Общая схема рекурсивного алгоритма состоит в следующем:

```
TChartPoint *Show ( TChart *pN ) {
 if ( pN != NULL ) pL = NULL;
 else if ( pN ∈ TChartPoint ) pL = pN;
 else {
  pF = Show(pN->GetFirstPoint());
  pL = Show(pN->GetLastPoint());
  // рисование линии <pN,pF,pL>
 }
 return pN;
}
```

- Отрисовать подплекс, на который указывает указатель начальной точки; запомнить конечную точку подплекса;
- Отрисовать подплекс, на который указывает указатель конечной точки; запомнить конечную точку подплекса;
- Отрисовать корневую линию.

Для нерекурсивного алгоритма определим класс TChartLine для представления линии, которая подлежит отрисовке.

```
class TChartLine // класс для методов отрисовки рисунков
{
  TChart      *pLine; // линия
  TChartPoint *pFp;   // начальная точка
  TChartPoint *pLp;   // конечная точка
  friend class TChart;
};
```

Общая схема общего алгоритма обхода состоит в следующем:

Линии, определяемые при обходе плекса, помещаются в стек,

- При линии, извлекаемой из стека, последовательно определяются начальная и конечная точки,
- Для определения начальной точки используется метод GetFirstPoint линии; если получаемый указатель указывает на линию, обработка текущей линии откладывается (линия помещается в стек) и начинается анализ новой линии; данная процедура выполняется итеративно до получения линии с известной начальной точкой,
- Для определения конечной точки используется метод GetLastPoint линии; если получаемый указатель указывает на линию, обработка текущей линии снова откладывается – текущая линия помещается в стек; после этого формируется описание новой линии, которая также помещается в стек и цикл алгоритма обхода повторяется,
- При получении линии с определенными граничными точками следует выполнить обработку линии (отрисовать); из стека извлекается новая линия, для которой конечная точка обработанной линии используется в качестве первой неизвестной граничной точки,

Выполнение обхода завершается при опустошении стека линий.

3. Разработка программного комплекса

3.1. Структура

С учетом рассмотренных выше предложений к реализации целесообразной представляется следующая структура программы:

- FTRAN.h, FTRAN.cpp – модуль с классом для формульного транслятора;
- TFormValue.h – шаблонный класс для формульных значений;
- TChartRoot.h – шаблонный базовый абстрактный класс;
- TChartPoint.h – класс для точки;
- TChartGroup.h – класс для составного геометрического объекта;
- TChart.h, TChart.cpp – модуль с классами конструируемого геометрического объекта и линии;
- TPlexTestkit.cpp – модуль программы тестирования,
- UserComm.h, UserComm.cpp – модуль функций, реализующих визуальный диалоговый интерфейс для взаимодействия с пользователем.

На рис. 6 показаны отношения между классами: стрелками с незакрашенным треугольником показаны отношения наследования (базовый класс – производный класс), ромбовидными стрелками – отношения ассоциации (класс-владелец – класс-компонент), обычными стрелками показано наличие поля типа указатель на другой класс.

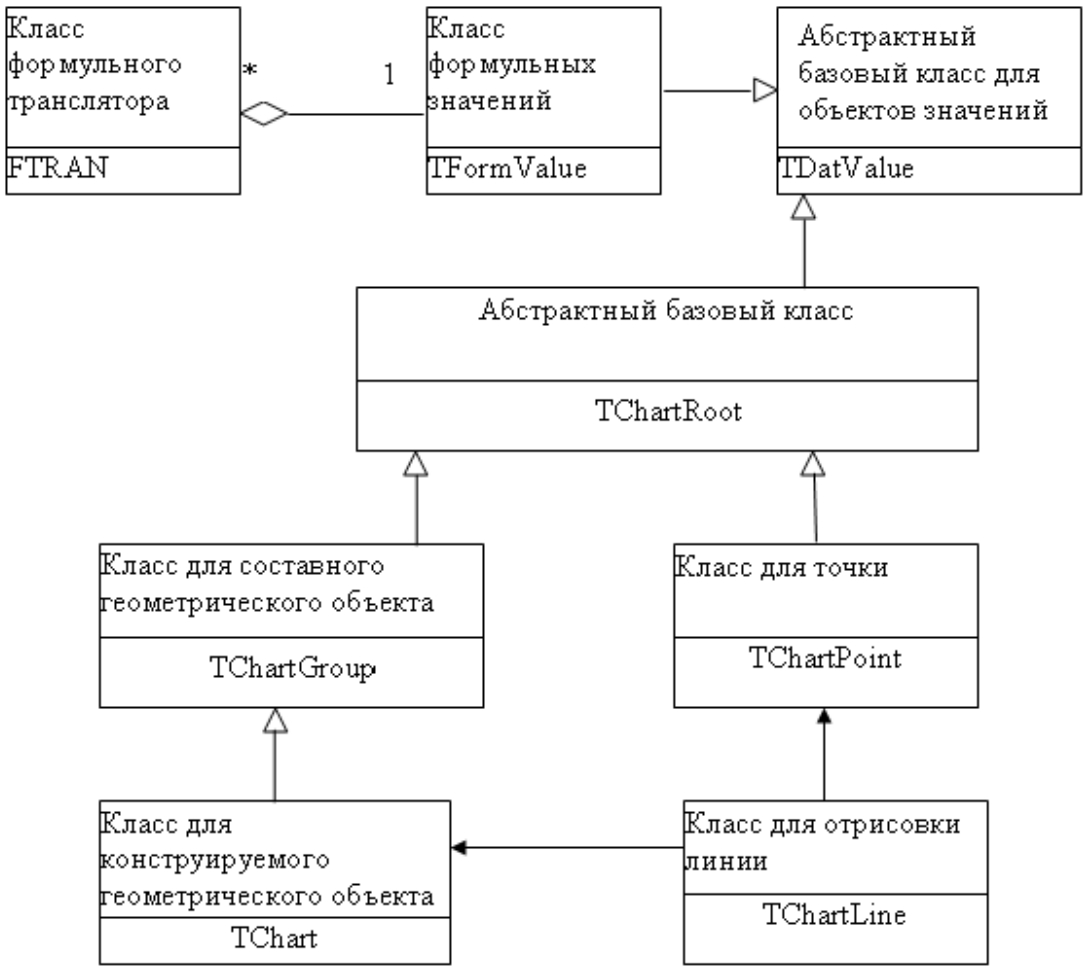


Рис. 11. Схема взаимодействия классов

3.2. Спецификации классов

С учетом предложенных к реализации алгоритмов спецификации классов могут состоять в следующем.

Класс **FTRAN** для формульного транслятора:

```
class FTRAN
{
protected:
  unsigned char Buffer[SIZE_POST];
public:
  int expression(char *text);
  int computing(double vars[], double *result);
};
```

Шаблонный класс для формульных значений (**TFormValue.h**):

```
#define FormLen 40

template <class TValue>
class TFormValue: public TDatValue
{
private:
  FTRAN ft;                // формульный транслятор
protected:
  TValue Value;            // значение
  char   Formula[FormLen]; // формула для вычисления значения
  double Param;            // значение параметра формулы

  virtual void Print(ostream &os);
public:
  TFormValue(TValue val = 0, const char *f = "");
  virtual ~TFormValue();
  void SetValue(TValue val = 0, const char *f = "");
  void SetFormula(const char *f = ""); // установить формулу
  TValue GetValue();                   // получить значение
  TFormValue& operator=(const TValue &val); // перегрузка присваивания
  TFormValue& operator=(const TFormValue &fval); // перегрузка =
  operator TValue() const;
  TValue GetValue(double par); // вычислить и вернуть значение
  TDatValue* GetCopy() { return new TFormValue(Value, Formula); }
};
```

Шаблонный базовый абстрактный класс (**TChartRoot.h**):

```
class TChartRoot: public TDatValue
{
protected:
  int Visible;            // видимость
  TFormValue<int> Active; // обрабатываются только активные объекты

  virtual void Print(ostream &os) { };
public:
  TChartRoot();
  virtual ~TChartRoot();
  int IsVisible(void) const; // проверка визуальности
  int IsActive (void) const; // проверка активности
  void SetActiveValue(int val = 1, char *f = NULL);
  virtual void Show() = 0;                    // визуализация объекта
  virtual void Hide() = 0;                    // скрытие объекта
  virtual void CalcParams(double t = -1)      // вычислить параметры
  { if (t >= 0) Active.GetValue(t); }
```

```
  virtual void ViewTimeShot(double t = -1);     // визуадизация объекта
};
```

Класс **TChartPoint** для точки:

```
class TChartPoint: public TChartRoot
{
protected:
  TFormValue<int> X, Y; // координаты точки
public:
  TChartPoint(int a = 0, int b = 0);
  virtual ~TChartPoint();
  int  GetValueX(void);
  int  GetValueY(void);
  void SetValueX(int val = 0, char *f = NULL);
  void SetValueY(int val = 0, char *f = NULL);
  virtual void Show(); // визуализация объекта
  virtual void Hide(); // скрытие объекта
  virtual void CalcParams(double t = -1); // вычислить параметры
  virtual TDatValue* GetCopy();           // создание копии
};
```

Класс **TChartGroup** для составного геометрического объекта:

```
class TChartGroup: public TChartRoot
{
protected:
  TDatList Group; // список графических элементов группы
public:
  TChartGroup() { }
  virtual ~TChartGroup();
  void InsUnit(TChartRoot *pUnit); // вставить объект в группу
  virtual void Show(); // визуализация объекта
  virtual void Hide();  // скрытие объекта
  virtual void CalcParams(double t = -1); // вычислить параметры
  virtual TDatValue* GetCopy(); // создание копии
};
```

Класс **TChart** для конструируемого геометрического объекта:

```
class TChart;
class TChartLine // класс для методов отрисовки рисунков
{
  TChart      *pLine; // линия
  TChartPoint *pFp;   // начальная точка
  TChartPoint *pLp;   // конечная точка
  friend class TChart;
};

class TChart: public TChartGroup
{
protected:
  stack<TChartLine> St;
public:
  TChart() {}
  virtual ~TChart() { }
  TChartRoot *GetFirstPoint(void); // получить начальную точку
  TChartRoot *GetLastPoint(void);  // получить конечную точку
  void SetFirstPoint(TChartRoot *pUnit); // вставить начальную точку
  void SetLastPoint (TChartRoot *pUnit); // вставить конечную точку
  virtual void Show(); // визуализация рисунка
  virtual void Hide(); // скрытие рисунка
};
```

3.3. Этапы разработки

Выполнение лабораторной работы должно быть разделено на несколько этапов. Каждый из этапов должен иметь достаточно небольшую длительность, при этом должна обеспечиваться возможность решения поставленной прикладной задачи (в той или иной ограниченной постановке) на как можно более ранних этапах выполнения работ.

С учетом высказанных рекомендаций последовательность разработки программ может быть следующей:

Этап 1. Реализация программ для работы с плексами простой структуры (например, для одной линии).

Этап 2. Реализация программ для работы с составными геометрическими объектами (поддержка группирования).

Этап 3. Реализация отрисовки чертежа на основе алгоритма обхода плексов без подплексов. Этап должен включать развитие диалоговой формы управления.

Этап 4. Реализация операций чтения и записи информации в файл.

Этап 5. Реализация общего алгоритма обхода плексов.

Этап 6. Реализация дополнительных операций обхода плексов.

Результаты выполнения каждого этапа требуют проведения самого тщательного тестирования, поскольку возможные ошибки могут проявляться только после некоторого длительного времени работы с плексами; ошибочные ситуации могут возникать не при каждом сценарии выполнения программ.

3.4. Рекомендации по разработке

Набор программ для работы с плексами представляет собой достаточно сложную программную систему. При выполнении лабораторной работы следует уделять особое внимание технологическим аспектам разработки программного обеспечения: этапность разработки, тщательное тестирование, командную методику выполнения работ, обеспечение простого и понятного интерфейса, использование инструментов поддержки процесса разработки (система контроля версий, система поддержки тестирования и др.).

4. Возможные темы дополнительных заданий

Для развития навыков практического программирования рекомендуются следующие направления расширения постановки задачи:

- Реализовать построение чертежа с использованием дополнительных геометрических элементов (окружностей и их дуг, прямоугольников и др.).
- Разработка операций динамического изменения (вставка и удаление линий) для чертежа (плекса).
- Реализовать проверку тождественности чертежей для разных схем структур данных.
- Разработка представления арифметических выражений с использованием плексов.
- Реализовать построение блок-схем.

5. Критерии оценивания выполнения лабораторной работы

Лабораторная работа считается выполненной при выполнении следующего минимального набора требований:

- Реализованы все классы, разработанные программы обеспечивают демонстрацию методов отрисовки плекса без подплексов, данные для построения считываются из файла (в качестве примера может быть использован чертеж, представленный на рис. 6).

В расширенном варианте выполнения лабораторной работы должен быть реализован обход плекса с подплексами, организован визуальный пользовательский интерфейс с функциями чтения-записи данных о плексе в файл, обеспечена поддержка анимации и др.

6. Вопросы и задания для самоконтроля

- Какие возможные модели можно использовать для представления чертежа?
- В чем состоит модель чертежа?
- Какая структура хранения используется для чертежа?
- Какие преимущества и недостатки есть у рекурсивного и нерекурсивного алгоритмов обхода плекса?
- Как можно модифицировать алгоритм обхода плекса для проверки чертежа на связность?

Лабораторная работа №8
Статистическая обработка результатов экзаменационной сессии (таблицы)

Введение

Представление данных во многих задачах из разных областей человеческой деятельности может быть организовано при помощи таблиц. Таблицы представляют собой последовательности строк (записей), структура строк может быть различной, но обязательным является поле, задающее имя (ключ) записи. Таблицы применяются в бухгалтерском учете (ведомости заработной платы), в торговле (прайс-листы), в образовательных учреждениях (экзаменационные ведомости) и являются одними из наиболее распространенных структур данных, используемых при создании системного и прикладного математического обеспечения. Таблицы широко применяются в трансляторах (таблицы идентификаторов) и операционных системах, могут рассматриваться как программная реализация ассоциативной памяти и т.п. Существование отношения «иметь имя» является обязательным в большинстве разрабатываемых программистами структур данных; доступ по имени в этих структурах служит для получения соответствия между адресным принципом указания элементов памяти ЭВМ и общепринятым (более удобным для человека) способом указания объектов по их именам.

Целью лабораторной работы помимо изучения способов организации таблиц является начальное знакомство с принципами проектирования структур хранения, используемых в методах решения прикладных задач. На примере таблиц изучаются возможность выбора разных вариантов структур хранения, анализ их эффективности и определения областей приложений, в которых выбираемые структуры хранения являются наиболее эффективными.

В качестве практической задачи, на примере которой будут продемонстрированы возможные способы организации таблиц, рассматривается проблема статистической обработки результатов экзаменационной успеваемости студентов (выполнение таких, например, вычислений как определение среднего балла по предмету и/или по группе при назначении студентов на стипендию или при распределении студентов по кафедрам и т.п.).

1. Постановка учебной задачи

1.1. Основные понятия и определения

Таблица (от лат. *tabula* – доска) – динамическая структура данных, базисным множеством которой является семейство линейных структур из записей (базисное отношение включения определяется операциями вставки и удаления записей).

Запись – кортеж, каждый элемент которого обычно именуется полем.

Имя записи (*ключ*) – одно из полей записи, по которому обычно осуществляется поиск записей в таблице; остальные поля образуют *тело записи*.

Двоичное дерево поиска – это представление данных в виде дерева, для которого выполняются условия:

для любого узла (вершины) дерева существует не более двух потомков (двоичное дерево);

для любого узла значения во всех узлах левого поддерева меньше значения в узле;

для любого узла значения во всех узлах правого поддерева больше значения в узле.

Хеш-функция – функция, ставящая в соответствие ключу номер записи в таблице (используется при организации таблиц с вычислимым входом).

1.2. Требования к лабораторной работе

В рамках данной лабораторной работы ставится задача создания программных средств, поддерживающих табличные динамические структуры данных (таблицы) и базовые операции над ними:

- поиск записи;
- вставка записи (без дублирования);
- удаление записи.

Выполнение операций над таблицами может осуществляться с различной степенью эффективности в зависимости от способа организации таблицы. В рамках лабораторной работы как показатель эффективности предлагается использовать количество операций, необходимых для выполнения операции поиска записи в таблице. Величина этого показателя должна определяться как аналитически (при использовании тех или иных упрощающих предположений), так и экспериментально на основе проведения вычислительных экспериментов.

В лабораторной работе предлагается реализовать следующие типы таблиц:

- просмотровые (неупорядоченные);
- упорядоченные (сортированные);
- таблицы со структурами хранения на основе деревьев поиска;
- хеш-таблицы или перемешанные (с вычисляемыми адресами).

Необходимо разработать интерфейс доступа к операциям поиска, вставки и удаления, не зависящий от способа организации таблицы.

Демонстрацию работоспособности разрабатываемых при выполнении лабораторной работы программ следует проводить на примере таблиц, содержащих данные о результатах экзаменационной успеваемости студентов. Для данного примера следует реализовать следующие операции:

для таблицы с результатами успеваемости отдельной студенческой группы:

- получение сведений об успеваемости отдельного студента (оценка по конкретному предмету, средняя оценка по всем предметам);
- определение средней оценки по группе по отдельному предмету или по всем предметам;
- подсчет количества студентов-отличников и т.п.

для нескольких таблиц с результатами успеваемости для всех студенческих групп курса:

- получение средних оценок по всем студенческим группам по отдельному предмету или по всем предметам;
- определение студенческой группы с лучшей успеваемостью по конкретному предмету (или по всем предметам экзаменационной сессии);
- подсчет количества студентов-отличников для всего курса и т.п.

Задача обработки результатов успеваемости может быть расширена возможностью хранения данных для нескольких экзаменационных сессий (например, за весь период обучения студентов).

Указанный выше набор операций обработки результатов успеваемости является минимально необходимым; определение полного перечня необходимых операций должно выполняться на этапе постановки лабораторной работе (в том числе, например, могут быть выбраны различающиеся наборы операций для получения разных постановок лабораторной работы).

Результаты обработки успеваемости рекомендуется дополнять построением графиков и диаграмм различного вида (например, диаграммы по средним баллам для разных студенческих групп, графики успеваемости студентов по разным предметам).

Полезным расширением постановки лабораторной работы является реализация возможности передачи данных в офисные программы семейства Microsoft Office (например, в систему обработку электронных таблиц Microsoft Excel).

1.3. Условия и ограничения

Сделаем следующие основные допущения:

- В качестве ключа будем рассматривать фамилию и инициалы студента, в качестве полей данных – экзаменационные оценки по учебным дисциплинам.
- Сведения об экзаменационной успеваемости должны быть разнесены по разным таблицам для каждой студенческой группы в отдельности.
- Исходные данные – количество и наименование дисциплин, данные о результатах сессии – должны извлекаться из текстовых файлов (для каждой студенческой группы в отдельности).

При изменении состояния таблиц должна обеспечиваться возможность сохранения данных в текстовых файлах.

Для контроля правильности работы программ должна обеспечиваться возможность пакетного выполнения операций (наименование операций и их параметры задаются при помощи текстового файла). После выполнения пакета операций и сохранения данных должна быть реализована возможность сравнения полученных файлов с заранее подготовленными проверочными файлами.

2. Метод решения

2.1. Структуры данных

Под *таблицей* следует понимать динамическую структуру данных, которая в каждый момент выполнения вычислений состоит из конечного набора элементов (записей); записи таблицы могут подразделяться на несколько полей; при этом количество и тип полей являются одинаковыми для всех записей таблицы. Первое поле всех записей таблицы является ключом, поля записи без ключевого поля образуют тело записи. Например, задавая соответствие между идентификаторами переменных (именами) и адресами памяти ЭВМ, мы можем построить простейшую таблицу вида,

имя	**адрес**
item	7542
…	…

sum	1726

в которой каждая строка-запись состоит из двух полей: поля-ключа (имени) и поля-тела записи (адреса).

Таблица, содержащая экзаменационные оценки имеет вид:

ФИО	Дисциплина 1	Дисциплина 2	Дисциплина 3	Дисциплина 4
Иванов И.И.	5	4	4,5	5
Петров П.П.	3	3	3	4
Сидоров С.С.	5	4	3	4

Для этой таблицы каждая строка состоит из ключа (ФИО) и тела записи (поля, содержащие оценки по четырем дисциплинам).

Основные операции, выполняемые над таблицами:

- поиск записи (по одному или нескольким ключам);
- вставка записи (с контролем возможных повторений);
- удаление записи.

Операции вставки и удаления служат для формирования требуемого набора записей; операция поиска записи по ключу обеспечивает доступ по имени (ключу) к записям таблицы.

2.2. Алгоритмы

2.2.1. Просматриваемые таблицы

Простейшим способом отыскания нужного элемента является метод полного просмотра (сканирования), когда искомый ключ сравнивается по очереди со всеми ключами таблицы, начиная с первого, вплоть до отыскания совпадающего элемента или до исчерпания записей. Если ключи в таблице расположены в произвольном порядке (неупорядоченная таблица), этот способ является единственно возможным.

Операция вставки в неупорядоченную таблицу может быть выполнена путём добавления новой записи в конец таблицы с корректировкой номера последней занятой строки. Операция удаления строки может быть реализована при помощи переписывания последней записи таблицы на место удаляемой и соответствующей корректировки номера последней строки.

Среднее количество просматриваемых записей таблицы при поиске записи по ключу при предположении равной вероятности использования ключей определяется следующим соотношением

$$T_{ср} = p\frac{N}{2} + (1-p)N,$$

где

p – вероятность того, что искомая запись имеется в таблице;

N – количество записей в таблице.

2.2.2. Упорядоченные таблицы

При большом количестве записей в таблице ($N>>1$) затраты на выполнение полного просмотра становятся значительными. Эффективность процедуры поиска можно повысить при размещении записей в таблице в порядке возрастания (или убывания) ключей (*упорядоченная*, или *сортированная таблица*). Упорядоченность таблиц может быть организована только при возможности сравнения ключей (на множестве ключей задано отношение линейного порядка). Для поиска нужной записи в таких таблицах может быть

использован быстрый метод *бинарного* (*двоичного*) поиска. Вместе с тем, в упорядоченных таблицах усложняется реализация операций вставки и удаления записей, при выполнении которых для сохранения упорядоченности становится необходимой перепаковка записей таблицы.

Среднее количество просматриваемых записей таблицы при использовании бинарного поиска определяется как

$$T_{ср} = \log_2 N .$$

2.2.3. Таблицы на основе деревьев поиска

Снижение трудоемкости операций вставки и удаления за счет исключения перепаковок может быть обеспечено при использовании списковых структур хранения. При этом сохранение эффективности двоичного поиска требует прямого доступа к звену со средней записью таблицы; из этого звена должна существовать возможность доступа к средним записям левой и правой частей таблицы и т.д. Данные требования могут быть обеспечены при использовании *деревьев поиска* для представления таблиц.

Использование деревьев поиска обеспечивает сложность в среднем $\log_2 N$ для всех операций обработки (поиска, вставки и удаления) таблиц

$$T_{ср} = \log_2 N$$

Максимальная сложность обработки деревьев поиска имеет порядок N (в случае вырождения дерева в список).

Для исключения вырожденных случаев деревья поиска обобщаются до *сбалансированных деревьев поиска*, в которых высота поддеревьев для любых узлов отличаются не более чем на 1. Подобные деревья обеспечивают сложность порядка $\log_2 N$ для любых вариантов исходных данных, но требуют расхода времени на проведение балансировки (методы балансировки деревьев поиска могут быть рассмотрены в качестве дополнительного задания).

2.2.4. Таблицы с вычисляемыми адресами

Иной возможный способ построения таблиц при большом количестве записей состоит в предварительном (перед непосредственным поиском по таблице) вычислении месторасположения искомой записи. Данный метод предполагает наличие некоторой простой функции $h(key)$, которая отображает множество имен на множество номеров строк таблицы. Эта функция называется функцией *хеширования* или расстановки; таблицы, получаемые при таком способе построения, называются *хеш-таблицами, таблицами с вычисляемыми адресами* или *перемешиваемыми таблицами*.

Функции расстановки могут быть построены разными способами. Например, в качестве номера строки, в которой хранится или будет храниться при вставке некоторый ключ, можно взять код первого символа имени, либо сумму всех кодов символов ключа по модулю числа M, где M – длина таблицы (размер массива, отведённого для её хранения).

При использовании таблиц с вычисляемыми адресами может возникнуть ряд дополнительных проблем. Так, например, при вставке новой записи функция расстановки может выдать номер занятой строки массива (функция расстановки может определять одни и те же значения для нескольких разных ключей). Такая ситуация при вставке записи называется относительным переполнением таблицы или *коллизией*. При возникновении коллизий возможны разные методы их разрешения. Рассмотрим два из них:

- *метод открытого перемешивания* состоит в добавлении к вычисленному занятому номеру некоторого фиксированного смещения (повторное

перемешивание) $k' = (k + p) \bmod N$; если новый адрес *k'* также является занятым, следует повторить процедуру повторного перемешивания до тех пор, пока не обнаружится свободная строка, либо таблица не будет исчерпана (если значения *p* и *N* являются взаимнопростыми, открытое перемешивание обеспечивает нахождение свободной строки массива);

- *метод цепочек* при возникновении коллизий формирует линейные списки (цепочки), в каждом из которых располагаются записи с одинаковым значением функции расстановки (в этом случае в строках массива для размещения записей следует добавить ещё одно поле для ссылки на следующее звено списка).

Среднее количество просматриваемых записей при поиске записи в перемешиваемых таблицах при предположении равной вероятности использования ключей и при использовании функции расстановки с равномерным рассеиванием ключей по строкам массива определяется следующим соотношением (разрешение коллизий по методу открытого перемешивания):

$$T_{ср} = \frac{(1 - \alpha/2)}{(1 - \alpha)}$$

где

α – коэффициент заполненности таблицы *(α=N/M);*

M – количество строк в массиве для хранения записей;

N – количество записей в таблице.

Следует отметить, что количество сравнений при поиске в перемешиваемых таблицах зависит не от количества записей в таблице, а от заполненности памяти, отведённой для размещения записей. Для примера, при заполненности массива на 75% (α=0.75) количество сравнений в среднем равно 2.5.

3. Разработка программного комплекса

3.1. Структура

При выполнении данной лабораторной работы следует разработать иерархию классов, учитывая, что все таблицы имеют как общие свойства (их описание следует поместить в определении базового класса), так и особенности выполнения отдельных операций (реализуются в отдельных классах для каждого вида таблиц). При разработке классов используется ранее разработанный класс TDatValue.

Рекомендуемый состав классов приведен ниже.

- TTabRecord.h, TTabRecord.cpp – модуль с классом объектов-значений для записей таблицы;
- TTable.h – абстрактный базовый класс, содержит спецификации методов таблицы;
- TArrayTable.h, TArrayTable.cpp – абстрактный базовый класс для таблиц с непрерывной памятью;
- TScanTable.h, TScanTable.cpp – модуль с классом, обеспечивающим реализацию просматриваемых таблиц;
- TSortTable.h, TSortTable.cpp – модуль с классом, обеспечивающим реализацию упорядоченных таблиц;
- TTreeNode.h, TTreeNode.cpp – модуль с абстрактным базовым классом объектов-значений для деревьев;

- TTreeTable.h, TTreeTable.cpp – модуль с классом, реализующим таблицы в виде деревьев поиска;
- TBalanceNode.h, TBalanceNode.cpp – модуль с базовым классом объектов-значений для сбалансированных деревьев;
- TBalanceTree.h, TBalanceTree.cpp – модуль с классом, реализующим таблицы в виде сбалансированных деревьев поиска;
- THashTable.h, THashTable.cpp – модуль с базовым классом, обеспечивающим реализацию таблиц с вычислимым входом;
- TArrayHash.h, TArrayHash.cpp – модуль с классом, обеспечивающим реализацию хеш-таблиц (разрешение коллизий на основе открытого перемешивания);
- TListHash.h, TListHash.cpp – модуль с классом, обеспечивающим реализацию хеш-таблиц (разрешение коллизий на основе метода цепочек);
- TTableTestkit.cpp – модуль программы тестирования.

3.2. Спецификации классов

С учетом предложенных к реализации алгоритмов можно сделать следующие объявления классов.

Класс объектов-значений для записей таблицы (файл **TTabRecord.h**):

```
typedef string TKey     // тип ключа записи
// Класс объектов-значений для записей таблицы
class TTabRecord : public TDatValue {
  protected:    // поля
    TKey Key;   // ключ записи
    PTDatValue pValue;   // указатель на значение
  public:  // методы
    TTabRecord (TKey k="", PTDatValue pVal = NULL)// конструктор
    void SetKey(TKey k);// установить значение ключа
    TKey GetKey(void);  // получить значение ключа
    void SetValuePtr(PTDatValue p);// установить указатель на данные
    PTDatValue GetValuePTR (void); // получить указатель на данные
    virtual TDatValue * GetCopy(); // изготовить копию
    TTabRecord & operator = (TTabRecord &tr);// присваивание
    virtual int operator == (const TTabRecord &tr); // сравнение =
    virtual int operator < (const TTabRecord &tr);  // сравнение «<»
    virtual int operator > (const TTabRecord &tr);  // сравнение «>»
//дружественные классы для различных типов таблиц, см. далее
  friend class TArrayTable;
  friend class TScanTable;
  friend class TSortTable;
  friend class TTreeNode;
  friend class TTreeTable;
  friend class TArrayHash;
  friend class TListHash;
};
```

Тип TKey описывает тип значений ключей записи. В данном примере для типа ключей использован класс string из стандартной библиотеки шаблонов (STL).

Абстрактный базовый класс содержит спецификации методов таблицы (**TTable.h**).

```
class  TTable : public TDataCom  {
  protected:
    int DataCount;  // количество записей в таблице
    int Efficiency; // показатель эффективности выполнения операции
  public:
    TTable(){ DataCount=0; Efficiency=0;} // конструктор
```

```
    virtual ~TTble( ) {}; // деструктор
    // информационные методы
    int GetDataCount ( ) const {return DatCount;}    // к-во записей
    int GetEfficiency ( ) const {return Efficiency;} // эффективность
    int IsEmpty ( ) const {return DataCount == 0;}   //пуста?
    virtual int IsFull ( ) const =0;                 // заполнена?
    // доступ
    virtual TKey GetKey (void) const=0;
    virtual PTDatValue GetValuePTR (void) const =0;
    // основные методы
    virtual PTDatValue FindRecord (TKey k) =0; // найти запись
    virtual void InsRecord (TKey k, PTDatValue pVal ) =0; // вставить
    virtual void DelRecord (TKey k) =0;        // удалить запись
    // навигация
    virtual int Reset (void) =0; // установить на первую запись
    virtual int IsTabEnded (void) const=0; // таблица завершена?
    virtual int GoNext (void) =0; // переход к следующей записи
    // (=1 после применения для последней записи таблицы)
};
```

Все методы данного класса разделены на четыре группы.

1. Информационные методы
   - *GetDataCount* – получить количество записей,
   - *GetEfficiency* – получить эффективность последней операции,
   - *IsEmpty* – проверить, не является ли таблица пустой,
   - *IsFull* – проверить, не является ли таблица полной.
2. Методы доступа к записям
   - *GetKey* – получить значение ключа текущей записи ,
   - *GetValuePTR* – получить указатель на значение текущей записи.
3. Методы навигации по таблице (итератор)
   - *Reset* – установить текущую позицию на первую запись,
   - *IsTabEnded* – проверка завершения таблицы (под ситуацией завершения таблицы понимается состояние после применения метода *GoNext* для текущей позиции, установленной на последней записи таблицы),
   - *GoNext* – переместить текущую позицию на следующую запись.
4. Методы обработки таблицы
   - *FindRecord* – поиск записи по значению ключа,
   - *InsRecord* – вставка записи в таблицу,
   - *DelRecord* – удаление записи.

Абстрактный класс для таблиц с непрерывной памятью служит для управления структурой хранения (**TArrayTable**)

```
#define TabMaxSize 25
enum TDataPos {FIRST_POS, CURRENT_POS, LAST_POS};

class  TArrayTable : public TTable {
  protected:
    PTTabRecord *pRecs; // память для записей таблицы
    int TabSize;        // макс. возм.количество записей в таблице
    int CurrPos;        // номер текущей записи (нумерация с 0)
  public:
    TArrayTable(int Size=TabMaxSize); // конструктор
    ~TArrayTble( ) {};                // деструктор
    // информационные методы
```

```
    virtual int IsFull ( ) const ; // заполнена?
    int GetTabSize( ) const ;      // к-во записей
    // доступ
    virtual TKey GetKey (void) const;
    virtual PTDatValue GetValuePTR (void) const;
    virtual TKey GetKey (TDataPos mode) const;
     virtual PTDatValue GetValuePTR (TDataPos mode) const;
    // основные методы
    virtual PTDatValue FindRecord (TKey k) =0; // найти запись
    virtual void InsRecord (TKey k, PTDatValue pVal ) =0; // вставить
    virtual void DelRecord (TKey k) =0;        // удалить запись
    //навигация
    virtual int Reset (void);   // установить на первую запись
    virtual int IsTabEnded (void) const; // таблица завершена?
    virtual int GoNext (void) ; // переход к следующей записи
    //(=1 после применения для последней записи таблицы)
    virtual int SetCurrentPos (int pos);// установить текущую запись
    int GetCurrentPos (void) const;     //получить номер текущей записи
  friend TSortTable;

};
```

Данный класс обеспечивает управление памятью (выделение и освобождение памяти). Методы доступа применимы к первой, текущей и последней записям таблицы, желаемый вариант доступа задается через параметр метода доступа. Методы *SetCurrentPos* (установить текущую позицию на запись с заданным номером) и GetCurrentPos (получить номер текущей записи) вводят операции прямого доступа к записям таблицы. В классе реализованы методы навигации.

Класс, обеспечивающий реализацию просматриваемых таблиц (**TScanTable**).

```
class  TScanTable: public TArrayTable {
  public:
    TScanTable(int Size=TabMaxSize): TArrayTable(Size){};//конструктор
    // основные методы
    virtual PTDatValue FindRecord (TKey k) ;//найти запись
    virtual void InsRecord (TKey k, PTDatValue pVal ) ;//вставить
    virtual void DelRecord (TKey k) ;//удалить запись

};
```

В данном классе реализованы методы обработки таблицы.

Упорядоченные таблицы (**TSortTable**).

```
enum TSortMethod {INSERT_SORT, MERGE_SORT, QUIQ_SORT};
class  TSortTable: public TScanTable {

 protected:
    TSortMethod SortMethod; // метод сортировки
    void SortData (void);   // сортировка данных
    void InsertSort (PTTabRecord *pMem, int DataCount); // метод вставок
    void MergeSort (PTTabRecord *pMem, int DataCount);  // метод слияния
    void MergeSorter (PTTabRecord * &pData,PTTabRecord * &pBuff,int Size);
    void MergeData (PTTabRecord *&pData,PTTabRecord *&pBuff,int n1,int n2);
    void QuiqSort (PTTabRecord *pMem, int DataCount); // быстрая сортировка
    void QuiqSplit (PTTabRecord *pData, int Size, int &Pivot);

 public:
    TSortTable(int Size=TabMaxSize): TScanTable(Size){};// конструктор
    TSortTable(const TScanTable &tab); // из просматриваемой таблицы
    TSortTable & operator=(const TScanTable &tab); // присваивание
    TSortMethod GetSortMethod(void);    // получить метод сортировки
    void SetSortMethod (TSortMethod sm);// установить метод сортировки
    // основные методы
    virtual PTDatValue FindRecord (TKey k) ; // найти запись
```

```
    virtual void InsRecord (TKey k, PTDatValue pVal ) ; // вставить
    virtual void DelRecord (TKey k) ;        // удалить запись

};
```

В данном классе приводятся три метода сортировки, другие методы сортировки могут рассматриваться как задания для самостоятельного выполнения. Метод поиска записи *FindRecord* основан на алгоритме бинарного поиска, в методах вставки *InsRecord* и удаления *DelRecord* происходит обращение к методу поиска и производится перепаковка таблицы.

Абстрактный базовый класс объектов-значений для деревьев (TTreeNode).

```
class  TTreeNode;
typedef  TTreeNode *PTTreeNode;
class  TTreeNode: public TTabRecord {
  protected:
    PTTreeNode pLeft, pRight; // указатели на поддеревья
  public:
    TTreeNode(TKey k="", PTDatValue pVal=NULL, PTTreeNode pL=NULL,
      PTTreeNode pR=NULL ): TTabRecord(k,pVal), pLeft(pL), pRight(pR) {};
    PTTreeNode GetLeft(void) const; // указатель на левое поддерево
   PTTreeNode GetRight(void) const; // указатель на правое поддерево
    virtual TDatValue * GetCopy();  // изготовить копию
  friend class TTreeTable;
  friend class TBalanceTree;
};
```

Таблицы со структурой хранения в виде деревьев поиска (**TTreeTable**).

```
class  TTreeTable: public TTable {
  protected:
    PTTreeNode pRoot; // указатель на корень дерева
    PTTreeNode *ppRef;// адрес указателя на вершину-результата в FindRecord
    PTTreeNode pCurrent;// указатель на текущую вершину
    int CurrPos;        // номер текущей вершины
    stack < PTTreeNode> St;// стек для итератора
    void DeleteTreeTab (PTTreeNode pNode); // удаление
  public:
    TTreeTable(): TTable(){CurrPos=0; PRoot=pCurrent=NULL; ppRef=NULL;}
    ~TTreeTable(){DeleteTreeTab (pRoot);} // деструктор
    // информационные методы
    virtual int IsFull ( ) const ; //заполнена?
    //основные методы
    virtual PTDatValue FindRecord (TKey k); // найти запись
    virtual void InsRecord (TKey k, PTDatValue pVal ); // вставить
    virtual void DelReco rd (TKey k);       // удалить запись
    // навигация
    virtual TKey GetKey (void) const;
    virtual PTDatValue GetValuePTR (void) const;
    virtual int Reset (void);  // установить на первую запись
    virtual int IsTabEnded (void) const; // таблица завершена?
    virtual int GoNext (void) ;// переход к следующей записи
    //(=1 после применения для последней записи таблицы)
};
```

Для обхода дерева используется стек из библиотеки STL (в нем запоминаются указатели на пройденные узлы дерева).

Базовый класс объектов-значений для сбалансированных деревьев (**TBalanceNode**).

```
#define BalOk 0
#define BalLeft -1
#define BalRight 1
class  TBalanceNode: public TTreeNode {
  protected:
    int Balance; // индекс балансировки вершины (-1,0,1)
  public:
    TBalanceNode (TKey k="", PTDatValue pVal=NULL, PTTreeNode pL=NULL,
      PTTreeNode pR=NULL, int bal=BalOk) : TTreeNode(k,pVal,pL,pR),
      Balance(bal) {}; // конструктор
    virtual TDatValue * GetCopy();  // изготовить копию
    int GetBalance(void) const;
    void SetBalance(int bal);
   friend class TBalanceTree;
};
```

Класс для сбалансированных деревьев (**TBalanceTree**).

```
class  TBalanceTree: public TTreeTable  {
    protected:
int InsBalanceTree(PTBalanceNode &pNode, TKey k, PTDatValue pVal);
int LeftTreeBalancing(PTBalanceNode &pNode); // баланс. левого поддерева
int RightTreeBalancing(PTBalanceNode &pNode);// баланс. правого поддерева
  public:
    TBalanceTree():TTreeTable(){} // конструктор
    //основные методы
    virtual void InsRecord (TKey k, PTDatValue pVal ); // вставить
    virtual void DelRecord (TKey k);                   // удалить
};
```

Дерево является сбалансированным, если для каждого его узла высота левого и правого поддеревьев различаются не более чем на 1. Так как операции вставки и удаления могут нарушить сбалансированность дерева, то необходимо в таких случаях производить процедуру балансировки.

Базовый класс для таблиц с вычислимым входом (**THashTable**).

```
class  THashTable : public TTable {
  protected:
    virtual unsigned long HashFunc(const Tkey key); // hash-функция
  public:
    THashTable() : TTable() {} //конструктор
};
```

В классе содержится хэш-функция, вычисляющая по ключу записи номер строки в структуре хранения.

Класс для таблиц с вычислимым входом, использующий для разрешения коллизий открытое перемешивание (**TArrayHash**).

```
#define TabMaxSize 25
#define TabHashStep 5
class  TArrayHash : public THashTable {
  protected:
    PTTabRecord *pRecs; // память для записей таблицы
    int TabSize;        // макс. возм. к-во записей
    int HashStep;       // шаг вторичного перемешивания
    int FreePos;        // первая свободная строка, обнаруженная при поиске
    int CurrPos;        // строка памяти при завершении поиска
    PTTabRecord pMark;  // маркер для индикации строк с удаленными записями
    int GetNextPos(int pos){return (pos+HashStep)%TabSize;};// откр. перем.
  public:
    TArrayHash(int Size= TabMaxSize, int Step= TabHashStep); // конструктор
    ~TArrayHash();
    // информационные методы
```

```
    virtual int IsFull ( ) const ; // заполнена?
    // доступ
    virtual TKey GetKey (void) const;
    virtual PTDatValue GetValuePTR (void) const;
    // основные методы
    virtual PTDatValue FindRecord (TKey k); // найти запись
    virtual void InsRecord (TKey k, PTDatValue pVal ); // вставить
    virtual void DelRecord (TKey k);        // удалить запись
    // навигация
    virtual int Reset (void);   // установить на первую запись
    virtual int IsTabEnded (void) const; // таблица завершена?
    virtual int GoNext (void) ; // переход к следующей записи
    //(=1 после применения для последней записи таблицы)
};
```

При открытом перемешивании добавление новых записей возможно при наличии свободных строк в таблице. Разрешение коллизий происходит путем вычисления нового адреса для записи, пока не будет найдена свободная строка. Поэтому при удалении строка таблицы помечается как удаленная без фактического удаления и освобождения памяти, но сохраняется возможность записи в неё нового значения при необходимости.

Класс для таблиц с вычислимым входом, использующий для разрешения коллизий метод цепочек (**TListHash**).

```
#define TabMaxSize 25
class  TListHash : public THashTable {
  protected:
    PTDatList *pList; // память для массива указателей на списки записей
    int TabSize;      // размер массива указателей
    int CurrList;     // список, в котором выполнялся поиск
  public:
    TListHash(int Size= TabMaxSize); // конструктор
    ~TListHash();
    // информационные методы
    virtual int IsFull ( ) const ; // заполнена?
    // доступ
    virtual TKey GetKey (void) const;
    virtual PTDatValue GetValuePTR (void) const;
    // основные методы
    virtual PTDatValue FindRecord (TKey k); // найти запись
    virtual void InsRecord (TKey k, PTDatValue pVal ); // вставить
    virtual void DelRecord (TKey k);        // удалить запись
    // навигация
    virtual int Reset (void);   // установить на первую запись
    virtual int IsTabEnded (void) const; // таблица завершена?
    virtual int GoNext (void) ; // переход к следующей записи
    //(=1 после применения для последней записи таблицы)
};
```

В методе цепочек добавление новых записей в случае возникновения коллизии происходит путем добавления значения в список, в связи с чем необходимо использовать ранее разработанный класс для списков (*TDatList*).

3.3. Этапы разработки

Как и ранее, выполнение лабораторной работы должно быть разделено на несколько этапов. Каждый из этапов должен иметь достаточно небольшую длительность, при этом должна обеспечиваться возможность решения поставленной прикладной задачи (в той или иной ограниченной постановке) на как можно более ранних этапах выполнения работ.

С учетом высказанных рекомендаций последовательность разработки программ может быть следующей:

Этап 1. Реализация программ для работы с просматриваемыми таблицами (классы TTabRecord, TTable, TArrayTable, TScanTable). Выполнение этапа может быть разделено на несколько итераций:

- Реализация некоторой упрощенной схемы начального заполнения таблицы;
- Реализация операции поиска;
- Реализация операции вставки;
- Реализация операции удаления.

Результаты каждой итерации должны быть протестированы – переход к следующей итерации разработки должен осуществляться только после успешного выполнения всех запланированных тестов.

На данном этапе должен быть разработан начальный вариант диалоговой формы управления табличными операциями.

Этап 2. Реализация первой очереди операций прикладной задачи из постановки лабораторной работы (статистическая обработка результатов экзаменационной сессии). В числе реализуемых операций может быть, например, ввод исходных данных из текстовых файлов, получение данных об успеваемости отдельных студентов, вычисление средних оценок по отдельным предметам и др.

Этап должен включать развитие диалоговой формы управления.

Этап 3. Реализация программ для работы с упорядоченными таблицами (класс TSortTable). Для проверки правильности работы программ результаты выполнения должны сравниваться с результатами, получаемыми при помощи просматриваемых (и надежно проверенных) таблиц. На этом же этапе может быть расширен список реализованных операций прикладной задачи и продолжено развитие диалоговой формы управления. На данном этапе должны быть начаты работы по получению графических форм результатов обработки экзаменационных данных (например, построение диаграмм по средним баллам для разных студенческих групп, графики успеваемости студентов по разным предметам и др.).

Этап 4. Реализация программ для работы с таблицами на основе деревьев поиска (классы TTreeNode, TTreeTable). Дальнейшее расширение списка реализованных операций прикладной задачи, диалоговой формы управления, графических форм вывода результатов. Тестирование.

Этап 5. Реализация программ для работы с таблицами с вычислимым входом (классы THashTable, TArrayHash). Дальнейшее расширение списка реализованных операций прикладной задачи, диалоговой формы управления, графических форм вывода результатов. Тестирование.

Этап 6. Выполнение вычислительных экспериментов для оценки эффективности различных способов организации таблиц. Разработка различных генераторов последовательности выполняемых операций (генерация только операций поиска, генерация потока операций всех типов, генерация потоков операций с равномерным использованием ключей и т.п.).

3.4. Рекомендации по разработке

Следует уделить большое внимание определению состава реализуемых операций прикладной задачи по статистической обработке результатов экзаменационной сессии. Набор операций может быть достаточно ограничен, но должен обеспечивать поддержку основных процедур анализа успеваемости, обычно выполняемых работниками деканата.

Ориентированность на практическое использование предъявляет особые требования к проектированию диалоговой формы управления таблицами – обычные требования для

интерфейса с пользователем состоят в обеспечении наглядности, понятности, дружественности и т.п.

При практическом использовании программ особое значение имеет наличие графических форм представления результатов обработки (графики, диаграммы и др.).

Как и прежде, должно быть обеспечено надежное функционирование разработанных программ. С этой целью может быть разработан тот или иной генератор тестовых данных для многократной проверки правильности программ (для фиксации процесса тестирования следует вести журнал тестирования). В числе тестов должны быть и таблицы достаточно большого объема (с объемом более тысячи записей).

При выполнении лабораторной работы может быть рекомендована *командная* форма разработки программ. В составе команды один разработчик может отвечать за реализацию программ для таблиц, другой разработчик может отвечать за разработку диалоговых форм управления таблицами, еще один может заниматься разработкой графических форм вывода результатов разработки, и, наконец, отдельный разработчик может проводить тестирование. Обязанности разработчиков могут изменяться в ходе выполнения лабораторной работы.

4. Возможные темы дополнительных заданий

Для развития навыков практического программирования рекомендуются следующие направления расширения постановки задачи:

- Реализовать дополнительные операции над таблицами (объединение таблиц, нахождение общей части нескольких таблиц, исключение общих записей и др.).
- Реализовать методы балансировки деревьев поиска и выполнить вычислительные эксперименты по оценке эффективности операций с таблицей при балансировке, сравнить работу с таблицей при наличии и в отсутствии методов балансировки.
- Реализовать дополнительный набор операций для обработки результатов успеваемости.
- Реализовать дополнительный набор графических форм вывода результатов обработки.
- Реализовать расширенное представление таблиц (наличие нескольких ключей, поиск записей с использованием нескольких ключей, поиск записей по значениям из тела записей и т.д.).
- Выполнить замену разработанных программ для реализации таблиц (без изменения программ обработки результатов экзаменационной сессии) на средства работы с таблицами в библиотеке STL. Оценить трудоемкость выполнения такой замены (общий размер изменяемого программного кода).
- Выполнить лабораторную работу на примере другой прикладной задачи (расписание движения поездов, сведения о стоимости товаров в разных магазинах и т.п.).

5. Критерии оценивания выполнения лабораторной работы

Лабораторная работа считается сданной при выполнении минимального набора требований:

- Реализованы просматриваемая, упорядоченная, хеш-таблица (с одним способом разрешения коллизий), таблицы на основе деревьев поиска. Эффективность поиска в лучшем, худшем и среднем случаях соответствует теоретическим оценкам.
- Для рассматриваемой прикладной задачи реализованы 4-5 операций для обработки результатов экзаменационной сессии и 2-3 графические формы вывода.

В более полном варианте выполнения лабораторной работы реализованы два варианта разрешения коллизий в хеш-таблице, реализован класс таблиц на основе сбалансированного дерева.

6. Вопросы и задания для самоконтроля

- Приведите примеры прикладных задач, при решении которых могут быть использованы таблицы.
- Какие дополнительные операции могут быть полезны при обработке результатов экзаменационной сессии?
- Для разных способов организации таблиц приведите примеры задач, в которых эти способы являются наиболее эффективными?
- Какова трудоемкость алгоритма сортировки вставками, алгоритма сортировки слиянием, метода быстрой сортировки?
- В чем различие сбалансированных и идеально сбалансированных деревьев поиска?
- Почему при реализации таблиц не используются идеально сбалансированные деревья поиска?
- Какие рекомендации следует учитывать при выборе хэш-функций?
- Какие существуют методы разрешения коллизий? Укажите их сильные и слабые стороны.

Литература

1. Топп У., Форд У. Структуры данных в C++. - М. Бином, 1999
2. Мейн М., Савитч У. Структуры данных и другие объекты в C++. - М.: Издательский дом “Вильямс”, 2003
3. Скиена С.С. Алгоритмы. Руководство по разработке. – СПб.: БХВ-Петербург, 2013
4. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ.- МЦМО, 1999
5. Вирт Н. Алгоритмы+структуры данных=программы. – М.: Мир, 1985.
6. Себеста Р. У. Основные концепции языков программирования. - М.: Издательский дом “Вильямс”, 2001.
7. Танненбаум Э. Современные операционные системы. - СПб.: Питер, 2002.

Учебно-методические разработки кафедры МО ЭВМ по учебному курсу «Методы программирования»

8. Балло Л.В., Барышева И.В., Гергель В.П., Гришагин В.А., Долгов Г.А., Кулакова А.П. Методы программирования : программа курса и описания лабораторных работ/ Под общей редакцией Гергеля В.П., Стронгина Р.Г. - Нижний Новгород: Изд-во Нижегородского университета, 1997.
9. Стронгин Р.Г., Кутасова Т.Ю. Методические материалы по общему курсу "ЭВМ и программирование". - Горький: изд. ГГУ, 1986.

10. Грудзинский А.О., Кутасова Т.Ю. Лабораторные работы по структурам данных. - Горький: изд. ГГУ, 1986.
11. Кутасова Т.Ю. Обработка геометрических объектов на ЭВМ (редактирование чертежей). - Горький: изд. ГГУ, 1987.
12. Грудзинский А.О. Реализация иерархических списков (редактирование текстов). - Горький: изд. ГГУ, 1986.
13. Баркалов А.В., Грудзинский А.О. Операционные системы (распределение ресурсов в системе мультипрограммирования). - Горький: изд. ГГУ, 1987.
14. Маркин Д.Л. Иерархическое меню. - Горький: изд. ГГУ, 1993.