Министерство образования и науки Российской Федерации
Национальный исследовательский Нижегородский государственный
университет им. Н.И. Лобачевского
Институт информационных технологий, математики и механики
Кафедра математического обеспечения и суперкомпьютерных технологий

Рабочие материалы студента по общему курсу «Методы программирования» (часть 2)

Учебный план подготовки
Бакалавров физико-математических наук по направлению
«прикладная математика и информатика»

Курс второй
Семестр третий
Лекции 36 часов
Практические занятия 36 часов
Лабораторные работы 36 часов
Зачет

Нижний Новгород, 2015

Рабочие материалы к учебному курсу «Методы программирования» подготовил профессор кафедры Программная инженерия, д.т.н., Гергель В.П.

Рабочие материалы заслушаны и утверждены на заседании кафедры математического обеспечения и суперкомпьютерных технологий.
Протокол №1 от 31.08.2015.

Директор Института ИТММ  Гергель В.П.

Содержание

Учебная программа	3
Тема 7: Редактирование текстов	9
Текст программ	16
Тема 8: Структуры хранения геометрических объектов	25
Текст программ	30
Тема 9.1: Организация доступа по имени. Просматриваемые таблицы	39
Текст программ	45
Тема 9.2: Упорядоченные таблицы	53
Текст программ	61
Тема 9.3: Представление таблиц с использованием деревьев поиска	65
Текст программ	74
Тема 9.4: Таблицы с вычислимым входом	81
Текст программ	86
Тема 10: Автоматизация управления ЭВМ и операционные системы	91
Тема 11: Представление графовых моделей на ЭВМ	97
Текст программ	102
Вопросы для контроля	108

Программа общего курса "ЭВМ и программирование"

1. Цели и задачи курса и его место в учебном процессе на факультете ВМК

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 проблемной области). Проблемный 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 проблемного специалиста, приспособленную для удобного описания объектов проблемной области и операций над этими объектами.

Цель данного курса состоит в изучении основных путей реализации математических методов моделирования и анализа в виде таких виртуальных машин.

1.2. Задачи изучения курса

Изучение курса включает освоение моделей и методов программного отображения на аппаратуру ЭВМ сложных математических моделей (отображающих объекты некоторой проблемной области и операции над ними), обеспечивающих создание виртуальных машин, в т.ч.

- методы представления математических структур, соответствующих сложным объектам (текстам, чертежам и т.п.), и операций над этими структурами;
- методы распределения ресурсов машины между модифицируемыми в процессе обработки структурами;
- методы фиксации шагов обработки как состояний в некотором фазовом пространстве (в результате чего преобразования могут рассматриваться как некоторые выкладки);
- методы указания структур, их частей и операций с помощью системы (виртуальных) обозначений.

1.3. Дисциплины, освоение которых необходимо при изучении данного курса

Курс опирается на материал одноименного вводного курса "ЭВМ и программирование", изучаемого в 1-2 семестрах и направленного на освоение ЭВМ как инструмента автоматизации исполнения алгоритмов обработки информации (общее представление об ЭВМ, понятие алгоритма, способы описания алгоритмов, программа на языке высокого уровня, пропуск задачи на ЭВМ, отладка).

При изучении курса предполагается знание учебного материала общего курса "Основы ЭВМ", дающего сведения о том, каким образом автоматизируется обработка информации на ЭВМ (универсальность вычислительной машины) и каковы пути расширения возможностей ЭВМ по реализации обработки информации. Студенты должны иметь представление о возможностях, которыми обладает аппаратура машин (архитектура ЭВМ) и ее программное расширение (системное обеспечение, работа в среде операционной системы).

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

Введение

Важность предмета. Проблема доказательства правильности программ. Способы снижения сложности программного обеспечения.

Цели и задачи курса. Структура учебного плана. Основная и дополнительная литература.

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.

1.2.3. Использование объектно-ориентированного программирования для реализации структур данных.

1.2.4. Практическая работа 1: Структура хранения для множеств. Характеристический вектор множества и его представление в виде битовой строки. Поэтапное проектирование структуры хранения битовой строки. Спецификация класса TBitField и реализация методов. Разработка класса TSet для представления множеств с использованием структуры данных типа битовая строка.

1.2.5. Практическая работа 2: Структуры хранения для матриц специального вида. Представление на основе двухиндексных массивов и оценка эффективности использования памяти. Исключение хранения ненулевых элементов. Представление матрицы как набора векторов разной длины. Матрица как вектор векторных элементов (шаблоны).

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.

1.4.2. Практическая работа 3: Разработка структуры хранения для динамической структуры типа стек. Поэтапная разработка с использованием наследования классов: Обработка кодов завершения программ (класс TDataCom), управления памятью и спецификации методов (класс TDataRoot), реализация операций стека (класс TStack).

1.4.3. Сравнение структур хранения линейных и динамических структур.

1.4.4. Хранение динамических структур при ограниченной памяти. Степень использования памяти. Управление размещением.

1.4.5. Практическая работа 4: Разработка структуры хранения для динамической структуры типа очереди. Введение циклических структур хранения. Инкапсуляция отношения следования. Реализация класса TQueue через наследования от класса TStack.

1.4.6. Хранение нескольких динамических структур и необходимость перераспределения памяти в процессе обработки информации. Пример: хранение двух стеков.

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.

1.5.3. Практическая работа 5: Структура хранения 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).

1.5.4. Роль гипотез о росте структур при разработке систем управления памятью. Пример использования гипотезы о сохранении тенденции роста с момента последней перепаковки. Смешанные гипотезы. Оценка параметров модели в ходе выполнения системы (адаптация). 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. Реализация списков с использованием языка высокого уровня. Стек свободной памяти. Исключение операций перепаковки.

1.6.3. Практическая работа 6: Реализация структуры хранения нескольких стеков с использованием списков на языке высокого уровня.

1.6.4. Сравнение непрерывной и списковой структур хранения (эффективность организации динамического распределения памяти, необходимость хранения дополнительной информации, возможность прямого способа доступа к данных).

1.6.5. Реализация списков с использованием динамически распределяемой памяти. Пример реализации структуры хранения. Примеры использования стеков: поразрядная сортировка, преобразование арифметических выражений в польскую форму записи.

1.6.6. Практическая работа 7: Разработка общего представления линейного списка для обеспечения списковой структуры хранения. Организация хранения в списках значений разного типа. Проектирование структуры списка. Операции для работы со списком (информационные методы, методы доступа к значениям в списке). Организация навигации по списку (реализация итератора). Вставка и удаление звеньев в списке. Обеспечение удобного интерфейса (безопасное преобразование типов, передача по значению, использование объектов вместо указателей). Разработка шаблона класса-переходника (proxy) для работы со списком конкретного типа значений.

1.6.7. Общая характеристика стандартной библиотеки шаблонов.2. Динамические структуры и представление на ЭВМ сложных математических моделей

2.1. **Учебно-практическая задача 2.1**: 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. **Учебно-практическая задача 2.2**: 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. Проектирование состава необходимых программ для обеспечения структуры хранения полиномов. Иерархия классов. Разработка программ циклического списка как производного класса от программ линейного списка. Поэтапная разработка программ.

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

↳2.3. **Учебно-практическая задача 2.3**: 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. **Учебно-практическая задача 2.4**: 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, алгоритм быстрой сортировки). 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. Проблемное языковое обеспечение.

↳4.1. Языковое обеспечение применений ЭВМ.

4.1.1. Сложные системы обозначений и правила их порождения. Элементарные символы (алфавит). Язык.

4.1.2. Пример языка для записи арифметических выражений.

↳4.2. Задание языка с помощью формальной грамматики.

4.2.1. Контекстно-свободные грамматики. Терминалы. Понятие языка (нетерминалы). Металингвистические переменные. Правила вывода.

4.2.2. Соглашения о компактных формах записи. Символ повторения. Указание необязательных элементов и альтернативных вариантов.

↳4.3. Наглядное представление грамматик с помощью синтаксических диаграмм.

4.3.1. Орограф. Вершины и дуги. Метки терминалов и нетерминалов. Повторения и циклы в графе. Необязательные варианты и непомеченные дуги.

4.3.2. Порождение допустимых цепочек с помощью синтаксических диаграмм.

4.3.3. Трансляция операторов языка как обработка входной последовательности автоматом, переходы которого описаны синтаксической диаграммой.

5. Автоматизация управления ЭВМ и операционные системы.

↳5.1. Управление прохождением задачи.

5.1.1. Основные стадии процесса решения задачи на ЭВМ с центральным управлением (подготовка программы и данных; построение исполняемого варианта программы; задание точки входа и пуск ЭВМ; автоматическое исполнение программ). Дополнительные операции управления процессом решения (контроль правильности

функционирования устройств ЭВМ; анализ особых ситуаций - деление на ноль, зацикливание и др.).

5.1.2. Автоматизация управления. Диспетчер. События и прерывания. Аппаратная поддержка (регистр прерываний, операции сохранения и восстановления состояния). Управляющие подпрограммы. Язык указаний пользователя; сообщения диспетчера. Пакет заданий. Состояния системы (исполнение заданий, обработка прерываний, ожидание). Обычный и привилегированный режимы. Защита памяти. Генерация системы (загрузка диспетчера).

5.1.3. Расширение функций диспетчера. Таймер (часы); возможность диагностики зацикливания. Стандартные функции. Организация обработки программ, написанных на языках высокого уровня (стадии преобразования исходного текста программы: исходные, объектные и абсолютные модули; управление программами системы программирования: транслятором, редактором связей, загрузчиком; использование системных библиотек).

5.2. Введение многопрограммного режима в целях равномерной загрузки устройств ЭВМ.

5.2.1. Фазы исполнения задач в многопрограммном режиме (счет, обмен, ожидание). Управление потоком задач; прерывание программы и переход к другой (запоминание слова состояния в информационном поле задачи); продолжение исполнения прерванной программы (восстановление слова состояния; релокация программы); динамическое распределение ресурсов (приоритеты; паспорт задачи; необходимость логических устройств ввода-вывода), защита задач от неправильных взаимодействий.

5.2.2. Операционная система как программное расширение устройства управления. Дополнительные функции (учет ресурсов, диагностика, архивная служба). Создание виртуальной машины для каждого пользователя. Режим распределения времени и обеспечения диалога с ЭВМ.

5.3. Математическая модель управления процессами и ресурсами в операционной системе.

5.3.1. Понятие процесса. Вектор состояния как непрерывная характеристика прерывистого процесса. Взаимодействие процессов.

5.3.2. Понятие ресурса. Разделение ресурса (формы разделения). Логический ресурс.

5.3.3. Понятие состояния операционной системы. Концепция процессов и ресурсов. Представление состояния в виде графа "процесс-ресурс". Автоматная модель процесса управления в операционной системе (состояния системы, входы и выходы, условия переходов из состояния в состояние). Обнаружение и исключение тупиков.

5.4. Представление графовых моделей на ЭВМ.

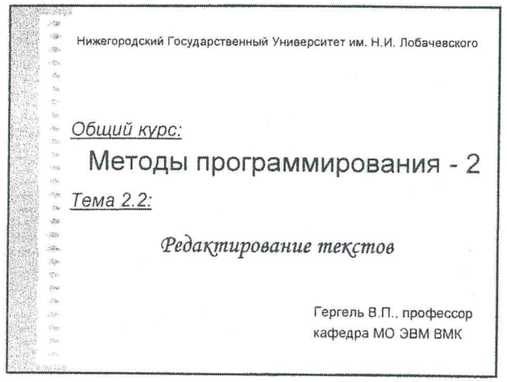
5.4.1. Элементы теории графов (понятие графа, ориентированные графы, путь в графе и его длина, циклы, помеченные и взвешенные графы, кратчайшие пути).

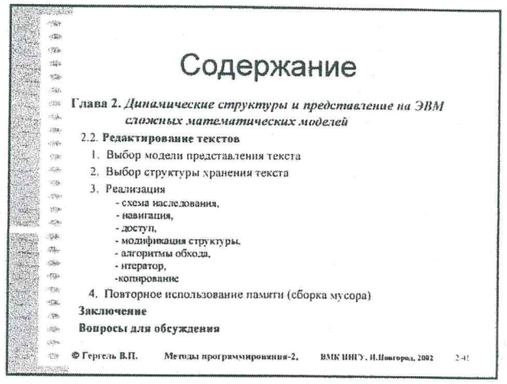
5.4.2. Выбор структуры представления графов. Использование матрицы смежности. Применение списков исходящих дуг для разреженных матриц смежности. Оценка способов представления графов.

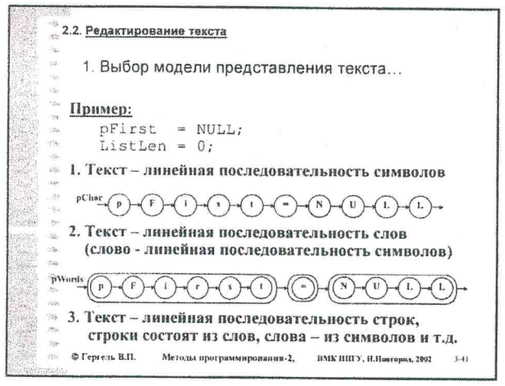
5.4.3. Реализация структуры представления графов. Классы для представления вершин и дуг. Класс TGraph для представления графов. Использование ранее разработанного программного обеспечения для реализации графов (множества, верхние треугольные матрицы, стеки, очереди, списки, таблицы).

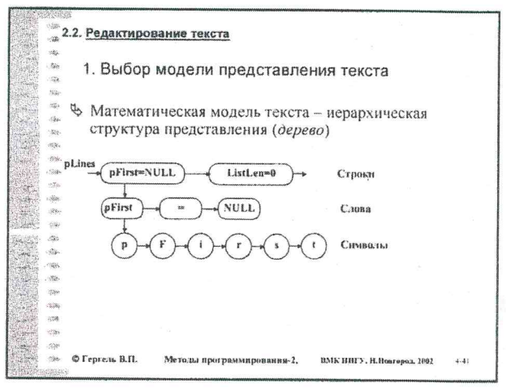
5.4.4. Алгоритмы обхода графов. Поиск в глубину и в ширину (обход по уровням). Реализация итератора графа.

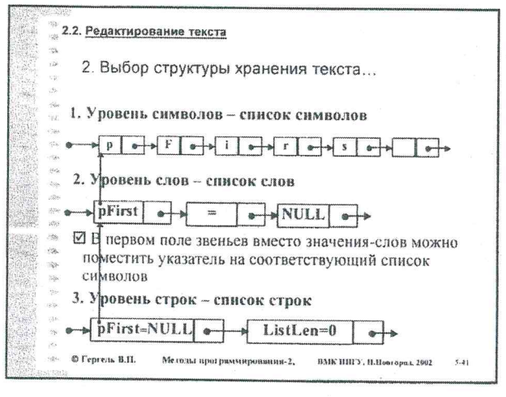
5.4.5. Пример задачи на обработку графов – поиск кратчайших путей. Алгоритм Дейкстры. Обоснование метода и оценка сложности.

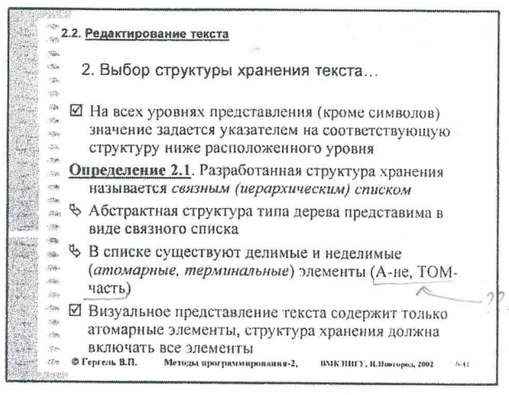












2. Выбор структуры хранения текста…

- Разные типы звеньев – трудности при управлении памятью, дублирование программ обработки

Единый тип звена

```
typedef TLink *PTLink;
class TLink{
  PTLink pNext;
  int Atom; // =1 - звено-атом
  union {
    PTLink pDown;
    char   Symb;
  };
};
```


2. Выбор структуры хранения текста…

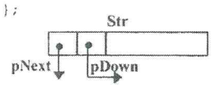
- ☑ Размер введенного унифицированного звена – 10 байт (=16 при учете округления при динамическом выделении памяти) ⇒ для представления символов такой вид звена является неэкономичным
- Возможное решение проблемы – повышение уровня атомарности
- Целесообразный уровень – **уровень строк**

typedef char[TextLineLength] TStr;

2. Выбор структуры хранения текста…

Структура звена

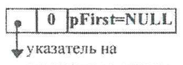
```
#define TextLineLength  20
typedef char TStr[TextLineLength];
class TTextLink : public TDatValue {
  protected:
    TStr Str;
    TTextLink *pNext, *pDown;
};
```



2. Выбор структуры хранения текста…

- ☑ Указатель pNext есть ссылка на следующий элемент того же уровня
- ☑ Указатель pDown есть ссылка на структуру хранения ниже расположенного уровня

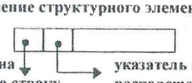
Представление строки (атомарного элемента)



- ☑ Признак атомарности элемента pDown==NULL

2. Выбор структуры хранения текста…

Представление структурного элемента



- ☑ Для ссылки на ниже расположенный уровень используется указатель pDown
- ☑ Поле Str можно использовать для именования структурных элементов текста (название всего текста, его разделов, подразделов и т.д.)

2. Выбор структуры хранения текста…

Пример структуры хранения

2. Выбор структуры хранения текста…

Базовые операции обработки звена

- Порождение звена (конструктор)

```
TTextLink ( TStr s=NULL,
            PTTextLink pn=NULL,
            PTTextLink pd=NULL );
```

- Переход к следующей звену

```
PTTextLink  GetNext() { return pNext; }
```

- Переход на подуровень

```
PTTextLink  GetDown() { return pDown; }
```

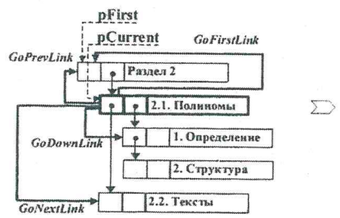

3. Реализация – схема наследования

3. Реализация – навигация по структуре…

```
PTTextLink pFirst;      // корень дерева
PTTextLink pCurrent;    // текущая строка
stack<PTTextLink> Path; // стек траектории
// навигация
int GoFirstLink (void); // к первой строке
int GoDownLink (void); // к след. строке по Down
int GoNextLink (void); // к след. строке по Next
int GoPrevLink (void); // к пред. позиции
```

- ☑ Указатель текущей строки pCurrent не включается в стек траектории движения по тексту

3. Реализация – навигация по структуре



- ☑ В стеке Path размечаются указатели на все звенья, лежащие на пути от начала текста до текущего звена

3. Реализация – доступ к текущей строке

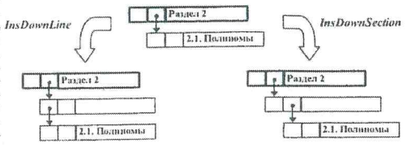
```
// доступ
string GetLine  (void); // чтение текста
                        текущего звена
void SetLine(string s); // замена текста
                        текущего звена
```


3. Реализация – модификация структуры…

```
// вставка и удаление строки в подуровне
void InsDownLine(string s);
void DelDownLine(void);
// вставка и удаление раздела в подуровне
void InsDownSection(string s);
void DelDownSection(void);
// вставка и удаление строки на том же уровне
void InsNextLine(string s);
void DelNextLine(void);
// вставка и удаление раздела на том же уровне
void InsNextSection(string s);
void DelNextSection(void);
```

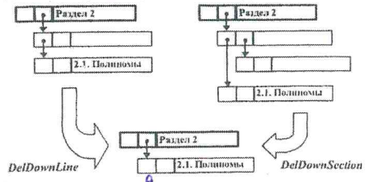
3. Реализация – модификация структуры…

Вставка строки и раздела в подуровне



3. Реализация – модификация структуры

Удаление строки и раздела в подуровне



☑ Операция удаления строки не выполняется, если исключаемый элемент не является атомарным

3. Реализация – алгоритмы обхода…

Печать текста: схема обхода – текст текущей строки, текст подуровня, текст следующего раздела текста того же уровня (top-down-next)

```
while (1) {
 if ( pLink != NULL ) {
   cout << pLink->Str;  // обработка звена
   St.push(pLink);      // запись в стек
   pLink = pLink->pDown;// переход на подуровень
 }
 else if ( St.empty() ) break;
 else {
   pLink = St.top(); St.pop();// выборка из стека
   pLink = pLink->pNext; // переход по тому же
 }                       // уровню
}
```


3. Реализация – алгоритмы обхода

Ввод текста из файла: уровень текста в файле можно выделить строками специального вида (например, скобками '{' и '}')

```
Глава 2
{
 2.1. Полиномы
 {
   1. Определение
   2. Структура
 }
 2.2. Тексты
 {
   1. Определение
   2. Структура
 }
}
```

Общая схема алгоритма

Повторить

- ввод строки
- ЕСЛИ '}' ТО Завершить
- ЕСЛИ '{' ТО Выполнить рекурсивно Ввод_текста
- Добавить строку на том же уровне

3. Реализация – итератор…

- ☑ Схема обхода TDN, нерекурсивный вариант
- ☑ Корневые звенья необработанных разделов текста запоминаются в стеке
- ☑ Текущая строка в стеке не хранится (кроме корневого звена всего текста)

3. Реализация – итератор…

Инициализация (Reset) – установка на корневое звено текста

```
pCurrent = pFirst;
if ( pCurrent != NULL ) {
  St.push(pCurrent);
  if ( pCurrent->pNext != NULL )
    St.push(pCurrent->pNext);
  if ( pCurrent->pDown != NULL )
    St.push(pCurrent->pDown);
}
```


3. Реализация – итератор…

Проверка завершения (IsTextEnded) – оценка номера текущего узла

```
return !St.empty(); // стек пуст ?
```


3. Реализация – итератор

Переход к следующему звену текста (GoNext)

- Получить звено из стека
- ЕСЛИ звено не является корнем всего текста ТО поместить следующие звенья (по указателям pNext и pDown) в стек

```
pCurrent = St.top(); St.pop();
if (pCurrent != pFirst ) {
  if ( pCurrent->pNext != NULL )
    St.push(pCurrent->pNext);
  if ( pCurrent->pDown != NULL )
    St.push(pCurrent->pDown);
}
```

Пример: программа, приложение

3. Реализация – копирование текста…

Общая замечания

- ☑ Для копирования текста необходимо предварительно скопировать разделы текста, на которые указывают указатели pDown и pNext ⇒ алгоритмы обхода NDT или DNT
- ☑ Для навигации по тексту-копии также необходим стек; использование стеков исходного текста и текста-копии должны быть согласованы

3. Реализация – копирование текста…

Общая схема алгоритма

- ☑ Для навигации по исходному тексту и тексту-копии используется один – объединенный - стек
- ☑ Каждое звено текста копируется за два прохода:
  - **1 проход** – при подъеме из подуровня (pDown)
    - создание копии звена
    - заполнение поля pDown (подуровень уже скопирован)
    - запись в поле Str значения Copy (для распознания звена при попадании на него при втором проходе)
    - запись в поле pNext указателя на звено-оригинал (для возможности последующего копирования текста исходной строки)
    - запись указателя на звено-копию в стек
  - **2 проход** – при извлечении звена из стека
    - заполнение полей Str и pNext
    - указатель на звено-копию запоминается в переменной cpl

3. Реализация – копирование текста

Пример: программа, приложение

4. Повторное использование памяти (сборка мусора)…

Общая замечания

- ☑ При удалении разделов текста для освобождения звеньев следует учитывать следующие моменты:
  - обход всех звеньев удаляемого текста может потребовать длительного времени,
  - при множественности ссылок на разделы текста (для устранения дублирования одинаковых частей) удаляемый текст нельзя исключить – этот текст может быть задействован в других фрагментах текста

↳ Память, занимаемая удаляемым текстом, не освобождается, а удаление текста фиксируется установкой указателей в состояние NULL (например, pFirst=NULL)

4. Повторное использование памяти (сборка мусора)…

- ☑ Подобный способ выполнения операций удаления текста может привести к ситуации, когда в памяти, используемой для хранения текста, могут присутствовать звенья, на которые нет ссылок в тексте и которые не возвращены в систему управления памятью для повторного использования. Элементы памяти такого вида носят наименование "*мусора*"
- ☑ Наличие "мусора" в системе может быть допустимым, если имеющейся свободой памяти достаточно для работы программ. В случае нехватки памяти необходимо выполнить "*сборку мусора*" (*garbage collection*)

4. Повторное использование памяти (сборка мусора)…

- ✉ Как получить доступ к системе управления памятью ?
- ☟ Возможный подход – разработка специализированной системы управления при помощи перегрузки операторов new и delete

```
void * operator new (size_t size);
void operator delete (void *p);
```


4. Повторное использование памяти (сборка мусора)…

Общая схема подхода…

- ☑ Для системы управления память выделяется полностью при начале работы программы; вся память форматируется и представляется в виде линейного списка свободных звеньев
- ☑ Для фиксации состояния памяти в классе TTextLink создается статическая переменная MemHeader типа TTextMem

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


4. Повторное использование памяти (сборка мусора)…

Общая схема подхода…

- ☑ Для выделения и форматирования памяти определяется статический метод **InitMemSystem** класса TTextLink

```
void InitMemSystem (int size) {
 char *p = new char[sizeof(TTextLink)*size];
 MemHeader.pFirst = (PTTextLink) p;
 MemHeader.pFree  = MemHeader.pFirst;
 MemHeader.pLast  = MemHeader.pFirst + (size-1);
 PTTextLink pLink = MemHeader.pFirst;
 for ( int i=0; i<size-1; i++, pLink++ ) {
   pLink->pNext = pLink+1;
 }
 pLink->pNext = NULL;
}
```


4. Повторное использование памяти (сборка мусора)…

Общая схема подхода…

- ☑ При запросе памяти в операторе **new** выделяется первое свободное звено

```
// выделение звена
void * operator new (size_t size) {
  PTTextLink pLink = MemHeader.pFree;
  if ( MemHeader.pFree != NULL )
    MemHeader.pFree = pLink->pNext;
  return pLink;
}
```


4. Повторное использование памяти (сборка мусора)…

Общая схема подхода…

- ☑ При освобождение звена в операторе **delete** звено включается в список свободных звеньев

```
// освобождение звена
void operator delete (void *pM) {
  PTTextLink pLink = (PTTextLink)pM;
  pLink->pNext = MemHeader.pFree;
  MemHeader.pFree = pLink;
}
```


4. Повторное использование памяти (сборка мусора)…

- ✉ Как при сканировании памяти различать звенья "мусора" и звенья текста ?
- ☟ Возможный подход – маркировка текстовых звеньев и звеньев списка свободных звеньев

Пример: программа, приложение

Заключение

- Характер использования текста определяет способ его представления
- Для хранения иерархически-представленного текста могут быть использованы связные списки
- Связные списки является общей структурой хранения структур типа дерева
- Использование итераторов позволяет обеспечить единый и простой способ обработки различных структур данных
- При разработке структур хранения сложных данных может потребоваться создание дополнительных средств управления памятью
- Возможный подход управления памятью – накопление и сборка мусора

Вопросы для обсуждения

- Дополнительные модели представления текста
- Набор операций при работе со связными списками
- Рекурсивные и итеративные алгоритмы обработки данных
- Статические данные и статические методы классов
- Перегрузка операторов new и delete
- Способы недопущения мусора при множественности ссылок на структурные элементы структуры хранения

Темы заданий для самостоятельной работы

- Расширение набора операций обработки текста (изменение структуры текста, копирование)
- Разработка визуальных средств работы с иерархическим текстом

Следующая тема

- Структуры хранения геометрических объектов на ЭВМ

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// texters.h Copyright (c) Гергель В.П. 03.09.2000, 19.01.2003
//
// обработка текстов - коды завершения

#ifndef __TEXTERS_H
#define __TEXTERS_H

#define  TextOK        0 /* ошибок нет      */
     /* коды ситуаций */
#define  TextNoDown  101 /* нет подуровня для текущей позиции */
#define  TextNoNext  102 /* нет след. раздела текущего уровня */
#define  TextNoPrev  103 /* текущая позиция в начале текста   */
     /* коды ошибок */
#define  TextError  -102 /* ошибка в тексте */
#define  TextNoMem  -101 /* нет памяти      */

#endif
// end of texters.h
```

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// textlink.h Copyright (c) Гергель В.П. 03.09.2000, 19.01.2003
//
// класс объектов-значений для строк текста

#ifndef __TEXTLINK_H
#define __TEXTLINK_H

#include <alloc.h>
#include <string.h>
#include "texters.h"
#include "datvalue.h"

#define TextLineLength  20
#define MemSize         20

class   TText;
class   TTextLink;
typedef TTextLink * PTTextLink;
typedef char TStr[TextLineLength]; → typedef char[TextLineLength] TStr;

class TTextMem {
  PTTextLink pFirst; // указатели на первое звено
  PTTextLink pLast;  // указатели на последнее звено
  PTTextLink pFree;  // указатели на первое свободное звено
  friend class TTextLink;
};
typedef TTextMem * PTTextMem;

class TTextLink : public TDatValue {
  protected:
    TStr Str;
    PTTextLink pNext, pDown;
    // система управления памятью
    static TTextMem MemHeader;
  public:
    static void InitMemSystem (int size=MemSize); // иниализация 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
```



```
  public:
    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; }
    TDatValue * GetCopy() { return new TTextLink(Str,pNext,pDown); }
  protected:
    virtual void Print(ostream &os) { os << Str; }
  friend class TText;
};
#endif
// end of textlink.h
```

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// textlink.cpp Copyright (c) Гергель В.П. 21.01.2003
//
// класс объектов-значений для строк текста

#include <conio.h>
#include "textlink.h"
#include "ttext.h"

void TTextLink :: InitMemSystem (int size) { // иниализация памяти
  char Line[100];
  MemHeader.pFirst = (PTTextLink) new char[sizeof(TTextLink)*size];
  MemHeader.pFree  = MemHeader.pFirst;
  MemHeader.pLast  = MemHeader.pFirst + (size-1);
  PTTextLink pLink = MemHeader.pFirst;
  for ( int i=0; i<size-1; i++, pLink++ ) { // размерка памяти
    pLink->pNext = pLink+1;
  }
  pLink->pNext = NULL;
}

void TTextLink :: PrintFreeLink (void) { // печать свободных звеньев
  PTTextLink pLink = MemHeader.pFree;
  cout << "List of free links" << endl;
  for ( ; pLink != NULL; pLink = pLink->pNext )
    cout << pLink->Str << endl;
}

void * TTextLink :: operator new (size_t size) { // выделение звена
  PTTextLink pLink = MemHeader.pFree;
  if ( MemHeader.pFree != NULL ) MemHeader.pFree = pLink->pNext;
  return pLink;
}

void TTextLink :: operator delete (void *pM) { // освобождение звена


}
```

```
void TTextLink :: MemCleaner(const TText &txt) { // сборка мусора
  // пробная версия - в памяти только один текст
  string st;
  // маркировка строк текста - маркер "&&&"
  for ( txt.Reset(); !txt.IsTextEnded(); txt.GoNext() ) {
    if ( st.find("&&&") != 0 ) txt.SetLine("&&&" + txt.GetLine());
  }
  // маркировка списка свободных звеньев
  PTTextLink pLink = MemHeader.pFree;
  for ( ; pLink != NULL; pLink = pLink->pNext ) strcpy(pLink->Str,"&&&");
  // сборка мусора
  pLink = MemHeader.pFirst;
  for ( ; pLink <= MemHeader.pLast; pLink++ ) {
    if ( strstr(pLink->Str,"&&&") != NULL ) { // строка текста или свободное звено
      strcpy(pLink->Str,pLink->Str+3);        // снятие маркировки
    }
    else delete pLink; // "неучтенное" звено в список свободных
  }
}

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// ttext.h - Copyright (c) Гергель В.П. 04.09.2000? 19.01.2003
//
// Тексты - иерархическая структура представления

#ifndef __TTEXT_H
#define __TTEXT_H

#include <stack>
#include <fstream.h>
#include "datacom.h"
#include "textlink.h"

class   TText;
typedef TText * PTText;

class TText : public TDataCom {
  protected:
    PTTextLink pFirst;      // указатель корня дерева
    PTTextLink pCurrent;    // указатель текущей строки
    stack<PTTextLink> Path; // стек траектории движения по тексту
                            // (звено pCurrent в стек не включается)
    // поля и методы реализации
    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 TxtFile
  public:
    TText(PTTextLink pl=NULL);
   ~TText() { pFirst = NULL; }
    PTText GetCopy();
    // навигация
    int GoFirstLink (void);  // переход к первой строке
    int GoDownLink  (void);  // переход к след. строке по Down
    int GoNextLink  (void);  // переход к след. строке по Next
    int GoPrevLink  (void);  // переход к пред. 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 подуровене
    void DelDownSection(void);      // удаление раздела в подуровене
    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 запись
    int IsTextEnded(void) const;    // таблица завершена ?
    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
};
#endif
// end of ttext.h

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
// ttext.cpp - Copyright (c) Гергель В.П. 04.09.2000? 19.01.2003
// Тексты - иерархическая структура представления

#define BufLength 80
#include <conio.h>
#include "ttext.h"

static char StrBuf[BufLength+1]; // буфер для ввода строк
static int  TextLevel;           // номер текущего уровня текста

TText :: TText(PTTextLink pl) {
  if ( pl == NULL ) pl = new TTextLink();
  pFirst = pl;
}
// навигация
int TText :: GoFirstLink ( void ) { // переход к первой строке
  while ( !Path.empty() ) Path.pop(); // очистка стека
  pCurrent = pFirst;
  if ( pCurrent == NULL ) SetRetCode(TextError); else SetRetCode(TextOK);
  return RetCode;
}
int TText :: GoDownLink  ( void ) { // переход к след. строке по Down
  SetRetCode(TextError);
  if ( pCurrent != NULL )
    if ( pCurrent->pDown != NULL ) {
      Path.push(pCurrent);
      pCurrent = pCurrent->pDown;
      SetRetCode(TextOK);
    }
  return RetCode;
}
int TText :: GoNextLink  ( void ) { // переход к след. строке по Next



}
```

```
int TText :: GoPrevLink  ( void ) { // переход к пред. позиции в тексте
  if ( Path.empty() ) SetRetCode(TextNoPrev);
  else {
    pCurrent = Path.top(); Path.pop();
    SetRetCode(TextOK);
  }
  return RetCode;
}

// доступ
string TText :: GetLine (void) { // чтение текущей строки
  if ( pCurrent == NULL ) return string("");
  else return string(pCurrent->Str);
}
void TText :: SetLine (string s) { // замена текущей строки
  if ( pCurrent == NULL ) SetRetCode(TextError);
  else strncpy(pCurrent->Str,s.c_str(),TextLineLength);
  pCurrent->Str[TextLineLength-1] = '\0';
}
// модификация
void TText :: InsDownLine (string s) { // вставка строки в подуровень
  if ( pCurrent == NULL ) SetRetCode(TextError);
  else {
    PTTextLink pd = pCurrent->pDown;
    PTTextLink pl = new TTextLink("",pd,NULL);
    strncpy(pl->Str,s.c_str(),TextLineLength);
    pl->Str[TextLineLength-1] = '\0'; // установка, если s длиннее Str
    pCurrent->pDown = pl;             // установка указателя на новую строку
    SetRetCode(TextOK);
  }
}
void TText :: InsDownSection(string s) { // вставка раздела в подуровень
  if ( pCurrent == NULL ) SetRetCode(TextError);
  else {
    PTTextLink pd = pCurrent->pDown;
    PTTextLink pl = new TTextLink("",NULL,pd);
    strncpy(pl->Str,s.c_str(),TextLineLength);
    pl->Str[TextLineLength-1] = '\0'; // установка, если s длиннее Str
    pCurrent->pDown = pl;             // установка указателя на новую строку
    SetRetCode(TextOK);
  }
}
void TText :: InsNextLine(string s) { // вставка строки в том же уровне









}
void TText :: InsNextSection(string s) { // вставка раздела в том же уровне









}
```



```
void TText :: DelDownLine(void) { // удаление строки в подуровне
  SetRetCode(TextOK);
  if ( pCurrent == NULL ) SetRetCode(TextError);
  else if ( pCurrent->pDown != NULL ) {
    PTTextLink pl1 = pCurrent->pDown;
    PTTextLink pl2 = pl1->pNext;
    if ( pl1->pDown == NULL ) pCurrent->pDown = pl2; // только для атома
  }
}
void TText :: DelDownSection(void) { // удаление раздела в подуровене
  SetRetCode(TextOK);
  if ( pCurrent == NULL ) SetRetCode(TextError);
  else if ( pCurrent->pDown != NULL ) {
    PTTextLink pl1  = pCurrent->pDown;
    PTTextLink pl2  = pl1->pNext;
    pCurrent->pDown = pl2;
  }
}
void TText :: DelNextLine(void) { // удаление строки в том же уровне









}
void TText :: DelNextSection(void) { // удаление раздела в том же уровне

}

// итератор
int TText :: Reset ( void ) { // установить на первую запись
  while ( !St.empty() ) St.pop(); // очистка стека
  // текущая строка в стеке не хранится
  // исключение - первая строка текста, которая на дне стека
  pCurrent = pFirst;
  if ( pCurrent != NULL ) {
    St.push(pCurrent);
    if ( pCurrent->pNext != NULL ) St.push(pCurrent->pNext);
    if ( pCurrent->pDown != NULL ) St.push(pCurrent->pDown);
  }
  return IsTextEnded();
}
int TText :: IsTextEnded ( void ) const { // таблица завершена ?
  return !St.size();
}

int TText :: GoNext ( void ) { // переход к следующей записи
  if ( !IsTextEnded() ) {
    pCurrent = St.top(); St.pop(); // если после выборки стек пуст, значит
    if ( pCurrent != pFirst ) {    // первая строка текста уже была обработана
      if ( pCurrent->pNext != NULL ) St.push(pCurrent->pNext);
      if ( pCurrent->pDown != NULL ) St.push(pCurrent->pDown);
    }
  }
  return IsTextEnded();
}
```

```
// копирование текста
PTTextLink TText :: GetFirstAtom(PTTextLink pl) { // поиск первого атома
  PTTextLink tmp = pl;
  while ( !tmp->IsAtom() ) {
    St.push(tmp); tmp = tmp->GetDown();
  }
  return tmp;
}

PTText TText :: GetCopy() { // копирование текста
  PTTextLink pl1, pl2, pl=pFirst, cpl=NULL;

  if ( pFirst != NULL ) {
    while ( !St.empty() ) St.pop(); // очистка стека
    while (1) {
      if ( pl != NULL ) {  // переход к первому атому
      pl = GetFirstAtom(pl); St.push(pl);
      pl = pl->GetDown();
      }
      else if ( St.empty() ) break;
      else {
        pl1 = St.top(); St.pop();
      if ( strstr(pl1->Str,"Copy") == NULL ) { // первый этап создания копии
          // создание копии - pDown на уже скопированный подуровень
        pl2 = new TTextLink("Copy",pl1,cpl);   // pNext на оригинал



      }
      else { // второй этап создания копии



        }
      }
    }
  }
  return new TText(cpl);
}

// печать текста
void TText :: Print () {
  TextLevel = 0;
  PrintText(pFirst);
}

void TText :: PrintText ( PTTextLink ptl ) {
   if ( ptl != NULL ) {
     for ( int i=0; i<TextLevel; i++ ) cout << "   ";
     cout << " " << *ptl << endl;
     TextLevel++; PrintText(ptl->GetDown());
     TextLevel--; PrintText(ptl->GetNext());
  }
}

// чтение текста из файла
void TText :: Read ( char *pFileName ) {
  ifstream TxtFile(pFileName);
  TextLevel = 0;
  if ( TxtFile != NULL ) pFirst = ReadText(TxtFile);
}
```



```
PTTextLink TText :: ReadText (ifstream &TxtFile) {
  PTTextLink pHead, ptl;
  pHead = ptl = new TTextLink();
  while ( TxtFile.eof() == 0 ) {
    TxtFile.getline(StrBuf,BufLength,'\n');
    if ( StrBuf[0] == '}' ) { TextLevel--; break; }
    else if ( StrBuf[0] == '{' ) { // рекурсия
      TextLevel++; ptl->pDown = ReadText(TxtFile);
    }
    else {  // присоединение следующей строки
      ptl->pNext = new TTextLink(StrBuf,NULL,NULL);
      ptl = ptl->pNext;
    }
  }
  ptl = pHead;
  if ( pHead->pDown == NULL ) { // удаление первой строки, если нет подуровня
    pHead = pHead->pNext; delete ptl;
  }
  return pHead;
}

//---------------------------------------------------------------------------
// тест для программ обработки текста
// 02.05.2002

#include <vcl.h>
#include <conio.h>
#pragma hdrstop
#include "ttext.h"

#pragma argsused

//TText txt, *pText;
string strs[100];  //  строки текста
int ns, nl;        // к-во строк и номер строки на экране

void TextTyper(TText &txt) { // печать текста
  clrscr(); gotoxy(1,2);
  txt.Print(); ns=0;
  for ( txt.Reset(); !txt.IsTextEnded(); txt.GoNext() )
    strs[ns++] = txt.GetLine();
  txt.GoFirstLink();
  nl=2; gotoxy(1,nl); cout << '\333'; // текущая строка
}

void TextLineMark(TText &txt) { // печать текущей строки
  string st = txt.GetLine();
  gotoxy(1,nl); cout << " ";
  for ( nl=0; nl < ns; nl++ )
    if ( st == strs[nl] ) break;
  nl+=2; gotoxy(1,nl); cout << "\333"; // текущая строка
}

#define HOME  71
#define DOWN  80
#define NEXT  77
#define UP    72
#define ESC   27
#define INS   82
#define DEL   83
#define ENTER 13
```

```
void TextProcessor(TText &txt) {
  int dir, unit;
  string st;
  char ch;
  do {
    gotoxy(1,1);
    cout << ">,v,^, Home, Ins, Del, Enter, Esc";
    ch = getch();
    if ( ch == ESC ) break;
    if ( ch != ENTER ) ch = getch();
    switch (ch) {
      case ENTER:
        gotoxy(1,1); clreol();
        cout << "Line (without blanks): "; cin >> st;
        txt.SetLine(st);
        TextTyper(txt);                  break;
      case HOME: txt.GoFirstLink(); break;
      case DOWN: txt.GoDownLink();  break;
      case NEXT: txt.GoNextLink();  break;
      case UP:   txt.GoPrevLink();  break;
      case INS:
      case DEL:
        gotoxy(1,1); clreol();
        cout << "Direction: 0 - Down, 1 - Next "; cin >> dir;
        gotoxy(1,1); clreol();
        cout << "Unit: 0 - Line, 1 - Section ";   cin >> unit;
        if ( ch == INS ) { // вставка
          gotoxy(1,1); clreol();
          cout << "Line(without blanks): "; cin >> st;
          if ( dir == 0 )
            if ( unit == 0 ) txt.InsDownLine(st); else txt.InsDownSection(st);
          else
            if ( unit == 0 ) txt.InsNextLine(st); else txt.InsNextSection(st);
        }
        else { // удаление
          if ( dir == 0 )
            if ( unit == 0 ) txt.DelDownLine(); else txt.DelDownSection();
          else
            if ( unit == 0 ) txt.DelNextLine(); else txt.DelNextSection();
        }
        TextTyper(txt);
      break;
    }
    TextLineMark(txt);
  } while ( ch != ESC );
}
TTextMem TTextLink::MemHeader;
int main(int argc, char* argv[]) {
  TTextLink::InitMemSystem();
  TText txt, *pText;
  txt.Read("tt.dat");
  TextTyper(txt);
  TextProcessor(txt);
  cout << "Final Printing" << endl;
  TTextLink::MemCleaner(txt);
  TTextLink::PrintFreeLink();
  txt.Print();
  cout << "printing by iterator" << endl;
  for ( txt.Reset(); !txt.IsTextEnded(); txt.GoNext() )
    cout << txt.GetLine() << endl;
//  pText = txt.GetCopy();
//  cout << "text copy" << endl;
//  pText->Print();
  return 0;
}
```


Нижегородский Государственный Университет им. Н.И. Лобачевского

Общий курс:

Методы программирования - 2

Тема 2.3:

Структуры хранения геометрических объектов

Гергель В.П., профессор
кафедра МО ЭВМ ВМК

Содержание

Глава 2. ***Динамические структуры и представление на ЭВМ сложных математических моделей***

2.3. **Структуры хранения геометрических объектов на ЭВМ**

1. Представление базовых геометрических объектов (понятие, схема реализации, формульное задание параметров, динамическая визуализация)
2. Группирование объектов
3. Конструирование объектов
4. Комбинирование объектов (понятие плекса, алгоритмы обхода)
5. Использование плексов для представления выражений общего вида

Заключение

Вопросы для обсуждения

2.3. Структуры хранения геометрических объектов

1. Представление базовых геометрических объектов…

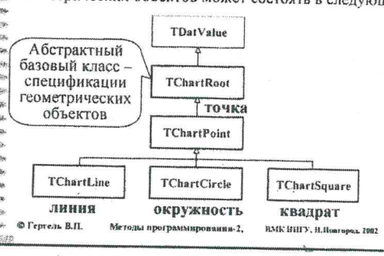
Рассмотрим в качестве базовых объектов **основные геометрические фигуры** – точку, окружность, прямоугольник и т.д.

- **Информационное описание объектов** – параметры фигуры (координаты, размер, радиус и др.). В общем случае, описание фигуры включает значение координат некоторой опорной точки
- **Операции обработки** геометрических объектов включают методы для задания и изменения параметров; расширим набор операций процедурами визуализации (например, на экране дисплея) и скрытия фигур.

2.3. Структуры хранения геометрических объектов

1. Представление базовых геометрических объектов…

Возможная **схема нерархин** классов для реализации геометрических объектов может состоять в следующем



2.3. Структуры хранения геометрических объектов

1. Представление базовых геометрических объектов…

(!!!) Для обеспечения возможности динамической визуализации геометрических объектов введем тип данных, значения которого вычисляются в соответствии с задаваемым формульным выражением

```
template <class TValue>
class TFormValue : public TDatValue {
  protected:
    char   Formula[FormLen]; // формула
    TValue Value;  // значение
    double Param;  // параметр формулы
  private:
    FTRAN ft;      // формульный транслятор
  public:
    TFormValue(TValue val=0, const char *f="");
    TValue GetValue(double par); // вычислить
    TValue GetValue();   // получить значение
};
```


2.3. Структуры хранения геометрических объектов

1. Представление базовых геометрических объектов…

Абстрактный базовый класс TChartRoot

```
class TChartRoot : public TDatValue {
  protected:  // поля
    int Visible;              // видимость
    TFormValue<int> Active; // активность
  public:
    TChartRoot();
    int IsVisible( void ) const; // высвечен ?
    int IsActive ( void ) const; // активен ?
    void SetActiveValue(int val=1,char *f=NULL);
    virtual void Show()=0; // визуализация
    virtual void Hide()=0; // скрытие объекта
    // перевычисление параметров
    virtual void CalcParams(double t=-1);
    // перевысветка объекта
    virtual void ViewTimeShot(double t=-1);
};
```

1. Представление базовых геометрических объектов…

Класс TChartPoint для описания точки на плоскости

```
class TChartPoint : public TChartRoot {
  protected:  // поля
    TFormValue<int> X, Y; // координаты точки
  public:
    TChartPoint (int a=0, int b=0);
    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 точки
    virtual void Hide(); // скрытие точки
    // перевычисление параметров
    virtual void CalcParams(double t=-1);
};
```


1. Представление базовых геометрических объектов…

Класс TChartCircle для описания окружности

```
class TChartCircle : public TChartPoint {
  protected:  // поля
    TFormValue<int> Radius; // радиус окружности
  public:
    TChartCircle (int a=0, int b=0, int rad=1);
    void SetRadiusValue(int val=1,char *f=NULL);
    virtual void Show(); // визуализация
    virtual void Hide(); // скрытие окружности
    // перевычисление параметров
    virtual void CalcParams(double t=-1);
};
```


1. Представление базовых геометрических объектов

Пример: Гелиоцентрическая система мира

	Размер (км)	Расст. от Солнца (млн.км)	Период обращения (лет)
Меркурий	2439	57,9	0,24
Венера	6050	108,2	0,62
Земля	6378	149,6	1,00

```
pEarth->SetValueX(150,"150+100*sin(100*(6.28*x/365))");
pEarth->SetValueY(150,"150-100*cos(100*(6.28*x/365))");
```

программа, приложение

2. Группирование объектов

Составной объект – набор геометрических объектов (как базовых, так и составных), рассматриваемых при выполнении операций обработки как единый объект

```
class TChartGroup : public TChartRoot {
 protected:
  TDatList Group; // список объектов
 public:
  TChartGroup () {}
  void InsUnit ( TChartRoot *pUnit ); // добавить
  virtual void Show(); // визуализация
  virtual void Hide(); // скрытие
  // перевычислить параметры
  virtual void CalcParams(double t=-1);
};
```

программа, приложение

3. Конструирование объектов

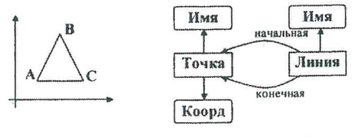
Геометрический объект может быть сконструирован с использованием уже существующих объектов (например, ломаная может быть определена через набор конечных точек составляющих отрезков)

```
class TChartPolyline : public TChartGroup {
 public:
  TChartPolyline () {}
  void InsPoint (TChartRoot *pUnit ): // добавить
  virtual void Show(); // визуализация
  virtual void Hide(); // скрытие
  // перевычислить параметры
  virtual void CalcParams(double t=-1);
};
```

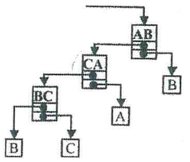
программа, приложение

4. Комбинирование объектов - понятие

Геометрический объект может быть образован при помощи сборки существующих объектов – рассмотрим данный способ построения новых объектов на примере рисунков (чертежей), образованных только из объектов двух базовых типов: точек и линий



4. Комбинирование объектов – структура хранения…



☑ Узел структуры хранения представляет линию чертежа

☑ Указатель на начальную точку линии может также указывать на линию, т.е. конечная точка предыдущей линии является начальной точкой следующей линии

Определение 2.2. Структура хранения данного вида называется ***плексом*** (содержит элементы разного типа)

↳ Плекс является общей структурой хранения сетевых моделей данных

4. Комбинирование объектов – структура хранения…

☑ Повторяющая точка на чертеже должна быть представлена одним и тем же объектом (не следует допускать множественности представления одного и того же значения)

☑ Разработанная структура хранения позволяет обеспечить представление чертежей, которые можно нарисовать без отрыва карандаша от бумаги

4. Комбинирование объектов – структура хранения

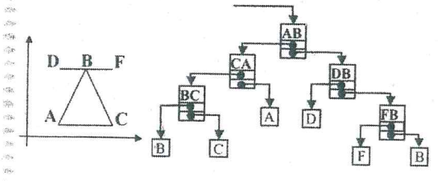
```
class TChart : public TChartGroup {
 protected:
   stack<TChartLine> St; // стек для обхода
 public:
   TChart () {}
   TChartRoot *GetFirstPoint(void); // нач.т.
   TChartRoot *GetLastPoint(void):  // кон.т.
   void SetFirstPoint  ( TChartRoot *pUnit );
   void SetLastPoint   ( TChartRoot *pUnit );
   virtual void Show(); // визуализация рисунка
   virtual void Hide(); // скрытие рисунка
};
```


4. Комбинирование объектов – алгоритм обхода 1

```
// общая схема алгоритма обхода для плекса
// переход на крайнюю левую точку
while ( pN ∉ TChartPoint ) {
  St.push(pN); pN = pN->GetFirstPoint();
}
// подъем по плексу и рисование
pF = pN;
while ( !St.Empty() ) {
  pN = St.top(); St.Pop();
  pL = pN->GetLastPoint();
  // рисование линии <pN,pF,pL>
  pF = pL;
}
```

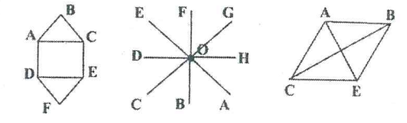

4. Комбинирование объектов – плекс с подплексами…

Рассмотрим чертеж, который нельзя нарисовать без отрыва карандаша от бумаги



4. Комбинирование объектов – плекс с подплексами

☑ Указатель на конечную точку линии может также указывать на линию, т.е. конечная точка линии верхнего узла плекса может являться конечной точкой линии правого нижнего подплекса



4. Комбинирование объектов – алгоритм обхода 2…

Рекурсивный вариант (общая схема)

```
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;
}
```


4. Комбинирование объектов – алгоритм обхода 2…

Для итеративного выполнения рекурсии определим класс для описания линии

```
class TChartLine {
  TChart      *pLine; // линия
  TChartPoint *pFp;   // начальная точка
  TChartPoint *pLp;   // конечная точка
  friend class TChart;
};
```


4. Комбинирование объектов – алгоритм обхода 2…

Алгоритм обхода для плекса с подплексами

- ☑ 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

4. Комбинирование объектов – алгоритм обхода 2…

Алгоритм обхода для плекса с подплексами

- ☑ 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 (нарисовать); из стека извлекается новая линия, для которой конечная точка обработанной линии используется в качестве первой неизвестной граничной точки

4. Комбинирование объектов – алгоритм обхода 2

Алгоритм обхода для плекса с подплексами

- ☑ Выполнение обхода завершается при опустошении стека линий

программа, приложение

5. Использование плексов для представления выражений общего вида

- ☑ Плекс может рассматриваться как структура представления для выражений самого общего вида (линия – операция, точки – операнды)

Пример: Арифметическое выражение (a*b+c)*(d-e/f)

Заключение

- Схемы наследования для построения базовых геометрических объектов
- Способы построения сложных объектов – группирование, конструирование, комбинирование
- Плекс – структура хранения сетевых информационных моделей
- Плекс – структура представления выражений самого общего вида
- Алгоритмы обхода плекса – плекс без подплексов, плекс общего вида (рекурсивный и итеративный вариант)

Вопросы для обсуждения

- Примеры использования плексов для представления сложных структур
- Сравнение рекурсивных и итеративных вариантов алгоритмов
- Методика итеративной реализации рекурсивных алгоритмов

Темы заданий для самостоятельной работы

- Разработка операций динамического изменения (вставка и удаление линий) для чертежа (плекса)
- Разработка представления арифметических выражения с использованием плексов

Следующая тема

- Организация доступа по имени

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// formval.h Copyright (c) Гергель В.П. 25.01.2003
//
// класс объектов-значений для формульных значений

#ifndef __FORMVAL_H
#define __FORMVAL_H

#include <string.h>
#include "datvalue.h"
#include "ftran.h"

#define FormLen  40

template <class TValue>
class TFormValue : public TDatValue {
  protected:
    TValue Value;             // значение
    char   Formula[FormLen]; // формула для вычисления значения
    double Param;             // значение параметра формулы
  private:
    FTRAN ft;                 // формульный транслятор
  public:
    TFormValue(TValue val=0, const char *f="") { SetValue(val,f); }
    void SetValue(TValue val=0, const char *f="") {
      Value=val; SetFormula(f);
    }
    void SetFormula(const char *f="") {         // установить формулу
      strcpy(Formula,f); ft.expression(Formula);
    }
    TValue GetValue() { return Value; }         // получить значение
    TFormValue & operator=(const TValue &val) { // перегрузка присваивания
      SetValue(val); return *this;
    }
    TFormValue & operator=(const TFormValue &fval) { // перегрузка присваивания
      SetValue(fval.Value,fval.Formula); return *this;
    }
    operator TValue () const { return Value; }
    TValue GetValue(double par) {               // вычислить и вернуть значение
      double args[1], val;
      args[0] = par;
      if ( strlen(Formula) > 0 ) {
        ft.computing(args,&val); Param=par; Value=val;
      }
      return Value;
    }
    TDatValue * GetCopy() { return new TFormValue(Value,Formula); }
  protected:
    virtual void Print(ostream &os) {
      os << "Value = " << Value << ", Formula = " << Formula;
    }
};
#endif
// end of formval.h
```



```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// tcroot.h - Copyright (c) Гергель В.П. 26.01.2003
//
// Графические геометрические объекты - базовый (абстрактный) класс

#ifndef __TCROOT_H
#define __TCROOT_H

#include "formval.h"

class TChartRoot : public TDatValue {
  protected:  // поля
    int Visible;             // видимость
    TFormValue<int> Active; // активность (обрабатываются только активные
объекты)
  public:
    TChartRoot () { Visible=0; Active=1; }
    int IsVisible( void ) const { return Visible; } // проверка визуальности
    int IsActive ( void ) const { return Active;  } // проверка активности
    void SetActiveValue(int val=1, char *f=NULL) { Active.SetValue(val,f); }
    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
      Hide(); CalcParams(t); Show();                // в момент времени t
    }
    protected:
      virtual void Print(ostream &os) {};
};
#endif

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// tcpoint.h - Copyright (c) Гергель В.П. 26.01.2003
//
// Графические геометрические объекты - точка

#ifndef __TCPOINT_H
#define __TCPOINT_H

#include "tb_chart.h"
#include "tcroot.h"

class TChartPoint : public TChartRoot {
  protected:  // поля
    TFormValue<int> X, Y; // координаты точки
  public:
    TChartPoint (int a=0, int b=0)  { X=a; Y=b; }
    int  GetValueX(void) { return X.GetValue(); }
    int  GetValueY(void) { return Y.GetValue(); }
    void SetValueX(int val=0, char *f=NULL) { X.SetValue(val,f); }
    void SetValueY(int val=0, char *f=NULL) { Y.SetValue(val,f); }
    virtual void Show() { // визуализация объекта
      if ( IsActive() && !IsVisible() ) {
        Form1->Image1->Canvas->Pixels[X][Y] = clBlack;
        Visible = 1;
      }
    }
```

```
    virtual void Hide() { // скрытие объекта
      if ( IsActive() && IsVisible() ) {
       Form1->Image1->Canvas->Pixels[X][Y] = clWhite;
       Visible = 0;
      }
    }
    virtual void CalcParams(double t=-1) { // вычислить параметры
      if ( t >= 0 ) Active.GetValue(t);
      if ( (t >= 0) && IsActive() ) { X.GetValue(t); Y.GetValue(t); }
    }
    virtual TDatValue * GetCopy() {    // создание копии
      TChartPoint *p = new TChartPoint(X,Y);
      p->Active = Active;
      p->X = X; p->Y = Y;
      return p;
    }
};
#endif

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// tccircle.h - Copyright (c) Гергель В.П. 26.01.2003
//
// Графические геометрические объекты - окружность

#ifndef __TCCIRCLE_H
#define __TCCIRCLE_H

#include "tcpoint.h"

class TChartCircle : public TChartPoint {
  protected:  // поля
    TFormValue<int> Radius; // радиус окружности
  public:
    TChartCircle (int a=0, int b=0, int rad=1) : TChartPoint(a,b) { Radius=rad; }
    void SetRadiusValue(int val=1, char *f=NULL) { Radius.SetValue(val,f); }
    virtual void Show() { // визуализация объекта
      if ( IsActive() && !IsVisible() ) {
        Form1->Image1->Canvas->Ellipse(X,Y,X+Radius,Y+Radius);
        Visible = 1;
      }
    }
    virtual void Hide() { // скрытие объекта
     if ( IsActive() && IsVisible() ) {
        Form1->Image1->Canvas->Pen->Color = clWhite;
        Form1->Image1->Canvas->Ellipse(X,Y,X+Radius,Y+Radius);
        Form1->Image1->Canvas->Pen->Color = clBlack;
        Visible = 0;
      }
    }
    virtual void CalcParams(double t=-1) { // вычислить параметры
      if (  t >= 0 ) TChartPoint::CalcParams(t);
      if ( (t >= 0) && IsActive() ) { Radius.GetValue(t); }
    }
    virtual TDatValue * GetCopy() {    // создание копии
      TChartCircle *p = new TChartCircle(X,Y,Radius);
      p->Active = Active;
      p->X = X; p->Y = Y;
      p->Radius = Radius;
      return p;
    }
};
#endif
```



```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// tcgroup.h - Copyright (c) Гергель В.П. 26.01.2003
//
// Графические геометрические объекты - группа

#ifndef __TCGROUP_H
#define __TCGROUP_H

#include "datlist.h"
#include "tcroot.h"

class TChartGroup : public TChartRoot {
  protected:  // поля
    TDatList Group; // список графических элементов группы
  public:
    TChartGroup () {}
    void InsUnit ( TChartRoot *pUnit ) { // вставить графический объект в группу
      Group.InsFirst(pUnit);
    }
    virtual void Show() { // визуализация объекта
      if ( IsActive() && !IsVisible() ) {
        for ( Group.Reset(); !Group.IsListEnded(); Group.GoNext() ) {
          TChartRoot *pChart = (TChartRoot *)Group.GetDatValue();
          pChart->Show();
        }
        Visible = 1;
      }
    }
    virtual void Hide() { // скрытие объекта
    }
    virtual void CalcParams(double t=-1) { // вычислить параметры
    }
    virtual TDatValue * GetCopy() {    // создание копии
      TChartGroup *p = new TChartGroup;
      for ( Group.Reset(); !Group.IsListEnded(); Group.GoNext() ) {
        TChartRoot *pChart = (TChartRoot *)Group.GetDatValue();
        p->InsUnit(pChart);
      }
      p->Active = Active;
      return p;
    }
};
#endif
```

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// tcpolyln.h - Copyright (c) Гергель В.П. 01.02.2003
//
// Графические геометрические групповые объекты - ломаная

#ifndef __TCPOLYLN_H
#define __TCPOLYLN_H

#include "tcgroup.h"

class TChartPolyline : public TChartGroup {
  public:
    TChartPolyline () {}
    void InsPoint ( TChartRoot *pUnit ) { // вставить точку
      TChartPoint *pPoint = dynamic_cast<TChartPoint *>(pUnit);
      if ( pPoint != NULL ) InsUnit(pUnit);
    }
    virtual void Show() { // визуализация ломаной
      if ( IsActive() && !IsVisible() ) {
        TChartPoint *pPoint = (TChartPoint *)Group.GetDatValue(FIRST);
        Form1->Image1->Canvas->MoveTo(pPoint->GetValueX(),pPoint->GetValueY());
        for ( Group.Reset(); Group.GoNext(), !Group.IsListEnded(); ) {
          pPoint = (TChartPoint *)Group.GetDatValue();
          Form1->Image1->Canvas->LineTo(pPoint->GetValueX(),pPoint->GetValueY());
        }
        Visible = 1;
      }
    }
    virtual void Hide() { // скрытие ломаной




};
#endif

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// tchart.h - Copyright (c) Гергель В.П. 01.02.2003
//
// Графические конструируемые геометрические объекты - линия

#ifndef __TCHART_H
#define __TCHART_H

#include <stack>
#include "tcpoint.h"
#include "tcgroup.h"

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 () {}
    TChartRoot *GetFirstPoint(void) { // получить начальную точку
      return (TChartRoot *)Group.GetDatValue(FIRST);
    }
    TChartRoot *GetLastPoint(void) { // получить конечную точку
      return (TChartRoot *)Group.GetDatValue(LAST);
    }
    void SetFirstPoint ( TChartRoot *pUnit ); // вставить начальную точку
    void SetLastPoint  ( TChartRoot *pUnit ); // вставить конечную точку
    virtual void Show(); // визуализация рисунка
    virtual void Hide(); // скрытие рисунка
};
#endif

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// tchart.cpp - Copyright (c) Гергель В.П. 01.02.2003
//
// Графические конструируемые геометрические объекты - линия

#include "tchart.h"

void TChart :: SetFirstPoint ( TChartRoot *pUnit ) { // вставить начальную точку
  TChartPoint *pPoint = dynamic_cast<TChartPoint *>(pUnit);
  TChart  *pNode  = dynamic_cast<TChart *>(pUnit);
  if ( (pPoint != NULL) || (pNode != NULL) ) {
    if ( Group.GetListLength() == 2 ) Group.DelFirst();
    Group.InsFirst(pUnit);
  }
}

void TChart :: SetLastPoint ( TChartRoot *pUnit ) { // вставить конечную точку




}

void TChart :: Show() { // визуализация рисунка
  if ( IsActive() && !IsVisible() ) {
    TChartLine CurrLine;
    TChartPoint *pPt;
    TChartRoot  *pRt;
    // инициализация
    CurrLine.pLine = this;
    CurrLine.pFp = CurrLine.pLp = NULL;
    while ( !St.empty() ) St.pop(); // очистка стека
    St.push(CurrLine);
    // цикл обхода
    while ( !St.empty() ) {
      CurrLine = St.top(); St.pop();
      while ( CurrLine.pFp == NULL ) { // нахождение начальной точки
        pRt = CurrLine.pLine->GetFirstPoint();
        if ( (pPt = dynamic_cast<TChartPoint *>(pRt)) != NULL ) // точка
          CurrLine.pFp = pPt;
```

```
    else { // слева - линия - переход к ее обработке
      St.push(CurrLine);
      CurrLine.pLine = dynamic_cast<TChart *>(pRt);
    }
  }
  if ( CurrLine.pLp == NULL ) { // нахождение конечной точки
```

```
  }
  if ( (CurrLine.pFp != NULL) && (CurrLine.pLp != NULL) ) { // отрисовка
 Form1->Image1->Canvas->MoveTo(CurrLine.pFp->GetValueX(),CurrLine.pFp->GetValueY());
 Form1->Image1->Canvas->LineTo(CurrLine.pLp->GetValueX(),CurrLine.pLp->GetValueY());
    // перенос конечной точки текущей линии на предшествующий уровень рекурсии
    pPt = CurrLine.pLp;
    if ( !St.empty() ) {
      CurrLine = St.top(); St.pop();
      if ( CurrLine.pFp == NULL ) CurrLine.pFp = pPt;
      else CurrLine.pLp = pPt;
      St.push(CurrLine);
    }
  }
} // завершение отрисовки чертежа
    Visible = 1;
  }
}

void TChart :: Hide() { // скрытие рисунка
  if ( IsActive() && IsVisible() ) {
    Form1->Image1->Canvas->Pen->Color = clWhite;
    Visible = 0; Show(); Visible = 0;
    Form1->Image1->Canvas->Pen->Color = clBlack;
  }
}
```



```
#include <vcl.h>
#pragma hdrstop

#include "tb_chart.h"
#include "tccircle.h"
#include "tcgroup.h"
#include "tcpolyln.h"
#include "tchart.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;
TChartCircle *pSun, *pMercury, *pVenus, *pEarth;
TChartGroup    Group;
TChartPolyline Polyln;
TChartPoint *pPoint1, *pPoint2, *pPoint3;
TChart Line1, Line2, Line3, Line4, Line5;
TChartPoint Pt1(200,50),  Pt2(100,100);
TChartPoint Pt3(50,150),  Pt4(150,150);
TChartPoint Pt5(100,200), Pt6(200,200);
double TimeValue, TimeStep, TimeStop;


//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
```

```
//---------------------------------------------------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender) {
  pSun = new TChartCircle(150,150,10);
  pMercury = new TChartCircle(150,150-38,4);
  pMercury->SetValueX(150,"150+38*sin(4.17*100*(6.28*x/365))");
  pMercury->SetValueY(112,"150-38*cos(4.17*100*(6.28*x/365))");
  pVenus = new TChartCircle(150,150-72,10);
  pVenus->SetValueX(150,"150+72*sin(1.61*100*(6.28*x/365))");
  pVenus->SetValueY(78,"150-72*cos(1.61*100*(6.28*x/365))");
  pEarth = new TChartCircle(150,150-100,11);
//  pEarth->SetActiveValue(1,"if sin(100*(6.28*x/365))>0 then 1 else 0");
  pEarth->SetValueX(150,"150+100*sin(100*(6.28*x/365))");
  pEarth->SetValueY(50,"150-100*cos(100*(6.28*x/365))");
  Group.InsUnit(pSun);
  Group.InsUnit(pMercury);
  Group.InsUnit(pVenus);
  Group.InsUnit(pEarth);
//  Group.SetActiveValue(1,"if sin(100*(6.28*x/365))>-0.75 then 1 else 0");
  pPoint1 = new TChartPoint(10,10);
  pPoint2 = new TChartPoint(50,50);
  pPoint3 = new TChartPoint(150,10);
  Polyln.InsPoint(pPoint1);
  Polyln.InsPoint(pPoint2);
  Polyln.InsPoint(pPoint3);

  Line5.SetFirstPoint(&Pt6);
  Line5.SetLastPoint(&Pt4);

  Line4.SetFirstPoint(&Pt5);
  Line4.SetLastPoint(&Line5);

  Line3.SetFirstPoint(&Line4);
  Line3.SetLastPoint(&Pt2);

  Line2.SetFirstPoint(&Pt3);
```

```
  Line2.SetLastPoint(&Line3);

  Line1.SetFirstPoint(&Line2);
  Line1.SetLastPoint(&Pt1);
}
//---------------------------------------------------------------------------

void __fastcall TForm1::BitBtn1Click(TObject *Sender) {
  pSun->Show();
}
//---------------------------------------------------------------------------

void __fastcall TForm1::BitBtn2Click(TObject *Sender) {
  if ( Group.IsVisible() ) Group.Hide();
  else Group.Show();

}
//---------------------------------------------------------------------------

void __fastcall TForm1::BitBtn3Click(TObject *Sender) {
  TimeValue = 0;
  TimeStep  = 0.1;
  TimeStop  = 20;
  Timer1->Enabled = true;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::Timer1Timer(TObject *Sender) {
  Group.ViewTimeShot(TimeValue);
  TimeValue +=TimeStep;
  if ( TimeValue > TimeStop )   Timer1->Enabled = false;
}
//---------------------------------------------------------------------------

void __fastcall TForm1::BitBtn4Click(TObject *Sender) {
  if ( Polyln.IsVisible() ) Polyln.Hide();
  else Polyln.Show();
}
//---------------------------------------------------------------------------

void __fastcall TForm1::BitBtn5Click(TObject *Sender) {
  if ( Line1.IsVisible() ) Line1.Hide();
  else Line1.Show();

}
//---------------------------------------------------------------------------
```


Нижегородский Государственный Университет им. Н.И. Лобачевского

Общий курс:

Методы программирования - 2

Тема 3.1:

Организация доступа по имени (таблицы)

Гергель В.П., профессор
кафедра МО ЭВМ ВМК

Содержание

Глава 3. ***Организация доступа по имени***
- 3.0. **Введение**
  1. Структура памяти
  2. Структура данных
  3. Понятие таблицы
  4. Важность понятия таблиц
  5. Формализация
  6. Заключение
- 3.1. **Просматриваемые таблицы**
  1. Общая характеристика подхода
  2. Реализация (структура записи, структура хранения, схема наследования)
  3. Оценка сложности (понятие временной сложности, О-нотация, виды зависимостей, минимальная и максимальная сложность, сложность в среднем)

Заключение и Вопросы для обсуждения

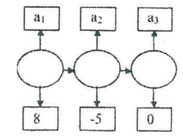
3. Организация доступа по имени

1. Структура памяти

- Для чтения или записи значения необходимо указать адрес элемента памяти
- Для человека более привычный способ указания данных - **имя**

3. Организация доступа по имени

2. Структура данных



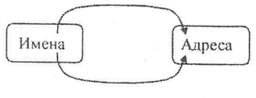
- Рассмотренный ранее материал касался в основном способов реализации основного (базисного) отношения
- Как реализовать отношение "иметь имя" ?

3. Организация доступа по имени

3. Понятие таблицы...

- Пусть
  - ***K*** - есть множество имен,
  - ***A*** - есть множество адресов,
- Тогда отношение "иметь имя" есть функция

$f: K \to A$



3. Организация доступа по имени

3. Понятие таблицы...

- Как задать функцию f?
- Возможный способ – *табличное задание функции*

имя	адрес
AB	3012
…	…
FE	4408
…	…

- ***Таблица*** - последовательность строк ***(записей)***
- **Запись** может состоять из нескольких ***полей***
- Одно из полей должно задавать ***имя записи*** (***ключ***), остальные поля образуют ***тело записи***

- Таблица – линейная структура ?

3. Понятие таблицы

Операции под таблицей.

- Поиск записи по ключу
- Вставка новой записи
- Удаление записи

Определение 3.1. Таблица – динамическая структура данных. Базисное множество – семейство линейных структур из записей, базисное отношение включения определяется операциями вставки и удаления записей.

4. Важность понятия таблиц

- ☑ Организация доступа по имени для управления информацией в привычной для человека форме
- ☑ Представление данных во многих задачах из разных областей приложений (таблицы идентификаторов, номенклатура изделий, словари и т.п.)
- ☑ Абстрагирование от проблем распределения памяти при размещении данных
- ☑ Реализация концепции *ассоциативной памяти* (память с доступом к данным по содержимому в этих данных)
- ☑ Отображение на ЭВМ такого важного математического понятия как *множества*

5. Формализация…

☑ Пусть

- $\mathbf{K}$ - множество имен,
- $\mathbf{A}$ - множество значений,
- $\mathbf{A^*=A+æ}$ - расширенное множество ($æ \notin \mathbf{A}$),
- $\mathbf{Z = K \times A^*}$ - множество записей ($z = <k,a> \in Z$),
- Z - множество всех подмножеств Z.

↳ Тогда таблица есть структура $\mathbf{T} = < Z , p >$

- p - базисное отношение включения,
- $t_n^k \in \mathbf{T}$ - текущий элемент (состояние) из k записей (n – размер памяти),

5. Формализация…

☑ Основные операции

- $T[k] \to A^*$ - поиск по ключу (при $<k,*> \notin \mathbf{T} \Rightarrow a = æ$ и $\delta=3$),
- $T+z \to T'$ - вставка записи (при $k=n \Rightarrow T' = T$ и $\delta=2$),
- $T - k \to T'$ - исключение записи (при $k=0 \Rightarrow T' = T$ и $\delta=1$).

☑ Дополнительные операции

- $T(n) \to t_n^0$ - создание таблицы,
- $\alpha_0(T)$ - предикат проверки пустоты ($\alpha_0(T)=1$ при $k=0$),
- $\alpha(T)$ - проверка переполнения памяти ($\alpha(T)=1$ при $k=n$).

↳ При завершении операции формируется код выполнения δ

- =1 - таблица пуста,
- =2 – таблица заполнена,
- =3 – записи в таблице нет,
- =4 – запись в таблице уже есть

5. Формализация…

☑ Аксиомы

- $T=T(n) \Rightarrow t = t_n^0$ и $\alpha_0(t)=1$ – созданная таблица является пустой
- $T'=T+z \Rightarrow \alpha_0(t')=0$ – таблица после операции вставки не пуста
- $T'=T-k \Rightarrow \alpha(t)=0$ – таблица после исключения записи не полна
- $(t+<k,a>)[k] \neq æ$ – после вставки записи в таблице поиск этой же записи должен быть успешным
- $(t-k)[k]=æ$ – после удалении записи в таблице данная запись должна отсутствовать
- $k1 \neq k2 \Rightarrow (t \pm k2)[k1]=t[k1] \pm k2$ - результат поиска не зависит от операций вставки и удаления записей с другими ключами

5. Формализация…

☑ Теоретико-множественные операции

- $T \cup T \to T$ - объединение таблиц,
- $T \cap T \to T$ - пересечение таблиц,
- $T - T \to T$ - вычитание таблиц.

Аксиомы ?

↳ Развитие табличного способа представления данных приводит к *реляционным базам данных.*

6. Заключение…

Варианты расширения понятия таблицы

- Наличие нескольких ключей
- Доступ по телу записей
- Сложные запросы для поиска
- Дополнительные операции

6. Заключение

Принципы реализации таблиц

- ☑ Операции над таблицей не определяют (не предполагают) тот или иной порядок размещения записей в памяти ЭВМ.
- ☑ Реализация таблицы должна способствовать быстрому выполнению операций (в основном, доступа)

↳ Свобода в размещении записей позволяет разработать несколько способов организации таблиц

↳ Введение различных способов предполагает явное или неявное существование разных типовых ситуаций при использовании таблиц

1. Общая характеристика подхода

Идея подхода (пример) – начальное накопление списка слов при изучении иностранного языка

…
one
two
three
four
…

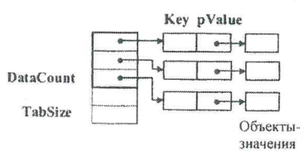
- ☑ **Поиск** – линейный (последовательный) просмотр
- ☑ **Вставка** – добавление в конец списка
- ☑ **Удаление** – исключение (стирание) с уплотнением (ex., путем переписыванием последней записи в удаляемую строку)

2. Реализация – структура записи

```
class TTabRecord : public TDatValue  {
  protected:
    TKey        Key;   // ключ
    PTDatValue pValue; // указатель на значение
  public:
    ...
};
```

Изложение материала проводится на примере значений типа TWord

2. Реализация – структура хранения



2. Реализация – схема наследования

2. Реализация – класс TTable...

☑ Информационные методы
- IsEmpty – проверить, не является ли таблица пустой
- IsFull – проверить, не является ли таблица полной
- GetDataCount – получить количество записей в таблице

☑ Методы доступа к значениям в списке
- GetKey – получить значение ключа текущей записи
- GetDatValue – получить указатель на значение текущей записи

2. Реализация – класс TTable...

☑ Методы навигации по таблице (*итератор*)
- Reset – установить текущую позицию на первую запись
- GoNext – переместить тек. позицию на следующую запись
- IsTabEnded – проверка завершения таблицы (под ситуацией завершения таблицы понимается состояние после применения GoNext для текущей позиции, установленной на последней записи таблицы)

2. Реализация – класс TTable

☑ Методы обработки таблицы
- FindRecord – поиск записи по значению ключа,
- InsRecord – вставка записи в таблицу,
- DelRecord – удаление записи

2. Реализация – класс TArrayTable

☑ Управление памятью (выделение/освобождение)

☑ Расширение к-ва записей, к которым возможен доступ

Методы GetKey и GetDatValue применимы к первой (FIRST), текущей (CURRENT) или последней (LAST) записи таблицы; желаемый вариант доступа задается через параметр метода)

☑ Введение операции прямого доступа к записям
- SetCurrentPos – установить текущую позицию на запись с заданным номером
- GetCurrentPos – получить номер текущей записи

☑ Реализация методов навигации (!)

2. Реализация – класс TScanTable

☑ Методы обработки таблицы
- FindRecord – поиск записи по значению ключа,
- InsRecord – вставка записи в таблицу,
- DelRecord – удаление записи

Пример: программа, приложение

3. Оценка эффективности...

☑ Эффективность алгоритма – время решения задачи (*временная сложность алгоритма T*)

☑ Время выполнения алгоритма на ЭВМ зависит от многих факторов – возможный способ состоит в измерении временной сложности в инструкциях (шагах), которые нужно выполнить для достижения результата

☑ Выражение для оценки временной сложности обычно зависит от размера (количества) входных данных N

Пример. Сложность алгоритма перемножения матриц

$$T = 2N^3 \cdot N^2,$$

где N есть порядок перемножаемых матриц

3. Оценка эффективности...

☑ Детальное построение выражения для временной сложности алгоритма часто является затруднительным, во многих случаях достаточно знания *верхних граничных оценок*

Определение 3.2. Функция f является верхней границей для функции g

$$g(N)=O(f(N)),$$

если

$$\exists n_0, c>0, \forall n>n_0 \Rightarrow g(N) \le f(N)$$

☑ Если $g(N)=O(f(N))$, то говорят, что g(N) имеет *порядок* (*степень* или *скорость*) *роста* f(N).

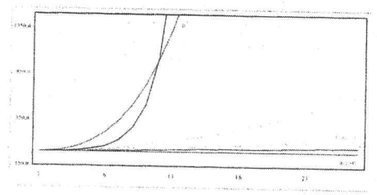
3. Оценка эффективности...

При построении верхних оценок определяющим является элемент наиболее высокого порядка формульного выражения сложности алгоритма

Типовые зависимости граничных оценок
- O(1) – константная оценка,
- O(logN) – логарифмическая оценка,
- O(N) – линейная оценка,
- O(N logN) – линейно- логарифмическая оценка,
- $O(N^a)$ – полиномиальная оценка (a - константа),
- $O(a^N)$ – экспоненциальная оценка (a - константа).

3. Оценка эффективности...

Типовые зависимости граничных оценок



3. Оценка эффективности...

Типовые зависимости граничных оценок

n	$\log_2 n$	n	n $\log_2 n$	n^2	n^3	2^n
1	0,0	1	0,0	1	1	2
2	1,0	2	2,0	4	8	4
5	2,3	5	11,6	25	125	32
10	3,3	10	33,2	100	1000	1024
15	3,9	15	58,6	225	3375	32768
20	4,3	20	86,4	400	8000	1048576
30	4,9	30	147,2	900	27000	1073741824
40	5,3	40	212,9	1600	64000	1099511627776
50	5,6	50	282,2	2500	125000	1125899906842620

3. Оценка эффективности...

T(n)	Размер задачи n P	10 P*	Увеличение размера задачи
n	1000	10000	10,0
n^2	32	100	3,2
n^3	10	22	2,2
2^n	10	13	1,3

*) ЭВМ с 10-кратной более высокой производительностью

Для решения вычислительно-трудоемких задач более действенным является уменьшение сложности используемых алгоритмов, чем повышение быстродействия компьютеров

3. Оценка эффективности...

Зависимость оценок сложности от входных данных
- Минимальная сложность T_{min}
- Максимальная сложность T_{max}
- Сложность в среднем $T_{ср}$

3. Оценка эффективности…

Сложность операций просматриваемых таблиц

Поиск

- $T_{min} = 1$
- $T_{max} = N$
- $T_{cp} = N/2$ (поиск имеющихся в таблице записей)
  $= p(N/2)+(1-p)N$ (p-вероятность наличия записи)

При построении оценок использовалось предположение, что вероятность использования всех ключей является одинаковой

Экспериментальная оценка сложности:
программа, приложение

3. Оценка эффективности

Сложность операций просматриваемых таблиц

Вставка

- $T_{cp} = p(N/2)+(1-p)(N+1)$
  $= N+1$ (вставка без дублирования)

Удаление

- $T_{cp} = p(N/2+1)+(1-p)N$

4. Заключение

Просматриваемые таблицы эффективны при незначительном количестве записей

Эффективное использование просматриваемых таблиц предполагает:

- поиск и удаление только имеющихся в таблице записей
- вставка только новых записей (для исключения проверки дублирования)

Заключение

- Таблицы являются важным и широко распространенным в практике типом структур данных
- Просматриваемые таблицы являются одним из самых простых способов организации таблиц; данный подход является эффективным при небольшом наборе имеющихся записей
- Временная сложность как показатель эффективности алгоритма; методика анализа сложности (О-нотация, виды зависимостей, минимальная и максимальная сложность, сложность в среднем)

Вопросы для обсуждения

- Возможные способы повышения эффективности просматриваемых таблиц
- Другие возможные способы организации таблиц

Темы заданий для самостоятельной работы

- Использование списков для структуры хранения просматриваемых таблиц
- Учет частоты использования записей (введение кэша)


```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// trecord.h Copyright (c) Гергель В.П. 03.09.2000
//
//  класс объектов-значений для записей таблицы

#ifndef __TRECORD_H
#define __TRECORD_H

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

typedef string  TKey;

class TTabRecord : public TDatValue  {
  protected:
    TKey        Key;   // ключ записи
    PTDatValue pValue; // указатель на значение
  public:
    TTabRecord ( TKey k="", PTDatValue pVal=NULL )
      { Key=k; pValue=pVal; }
    void SetKey ( TKey k ) { Key = k;     }
    TKey GetKey ( void   ) { return Key; }
    void       SetValuePtr ( PTDatValue p ) { pValue = p;     }
    PTDatValue GetValuePtr ( void )         { return pValue; }
    virtual  TDatValue * GetCopy(); // изготовить копию
    TTabRecord & operator=(TTabRecord &tr) {
      Key = tr.Key; pValue = tr.pValue;
      return *this;
    }
    virtual int  operator==(const TTabRecord &tr) { return Key == tr.Key; }
    virtual int  operator< (const TTabRecord &tr) { return Key  > tr.Key; }
    virtual int  operator> (const TTabRecord &tr) { return Key  < tr.Key; }
  protected:
    virtual void Print(ostream &os) { os << Key << " " << *pValue; }

  friend class TArrayTable;
  friend class TScanTable;
  friend class TSortTable;
  friend class TTreeNode;
  friend class TTreeTable;
  friend class TArrayHash;
  friend class TListHash;
};
typedef TTabRecord *PTTabRecord;
#endif
// end of trecord.h

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// trecord.cpp Copyright (c) Гергель В.П. 03.09.2000
//
//  класс записей таблицы

#include "trecord.h"

TDatValue * TTabRecord :: GetCopy() { // изготовить копию
  TDatValue *temp = new TTabRecord(Key,pValue);
  return temp;
}
// end of trecord.cpp
```

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// tword.h Copyright (c) Гергель В.П. 01.12.2002
//
// Таблицы - класс объектов-значений - слова

#ifndef __TWORD_H
#define __TWORD_H

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

class TWord : public TDatValue {
  protected:
    string Word;
  public:
    TWord ( string w = "" ) { Word = w; }
    virtual TDatValue * GetCopy(); // изготовить копию
    string  GetWord   ( void )         { return Word; }
    void    SetWord   ( string &w )    { Word = w;    }
    TWord & operator= (const TWord &w) { Word = w.Word; return *this; }
    int     operator==(const TWord &w) { return Word==w.Word; }
  protected:
    virtual void Print(ostream &os) { os << Word; }
};
typedef TWord *PTWord;
#endif
// end of tword.h

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// tword.cpp - Copyright (c) Гергель В.П. 09.08.2000
//
//   класс объектов-значений для слов

#include "tword.h"

TDatValue * TWord :: GetCopy() { // изготовить копию
  TDatValue *temp = new TWord(Word);
  return temp;
}
// end of tword.cpp

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// ttable.h - Copyright (c) Гергель В.П. 30.08.2000
//
// Таблицы - базовый класс

#ifndef __TTABLE_H
#define __TTABLE_H

#include <iostream.h>
#include "datacom.h"
#include "trecord.h"

#define TabOK         0  // ошибок нет
#define TabEmpty   -101  // таблица полна
#define TabFull    -102  // таблица пуста
#define TabNoRec   -103  // нет записи
#define TabRecDbl  -104  // дублирование записи
#define TabNoMem   -105  // нет памяти
```



```
class TTable : public TDataCom {
  protected:
    int DataCount;  // количество записей в таблице
    int Efficiency; // показатель эффективности выполнения операции
  public:
    TTable() { DataCount=0; Efficiency=0; }
    virtual ~TTable() {};
    // информационные методы
    int GetDataCount()   const { return DataCount;      } // к-во записей
    int GetEfficiency()  const { return   Efficiency;   } // эффективность
    int IsEmpty()        const { return DataCount == 0; } // пуста ?
    virtual int IsFull() const = 0;                       // заполнена ?
    // 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 GoNext для последней записи таблицы)
    // доступ
    virtual TKey GetKey( void ) const=0;
    virtual PTDatValue GetValuePtr( void ) const=0;
    // Печать таблицы
    friend ostream& operator<<(ostream &os,TTable &tab ) {
      //  cout << "Печать таблицы" << endl;
      cout << "Table printing" << endl;
      for ( tab.Reset(); !tab.IsTabEnded(); tab.GoNext() ) {
        os << " Key: " <<  tab.GetKey()
           << " Val: " << *tab.GetValuePtr();  os << endl;
      }
      return os;
    }
};
#endif

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// arraytab.h - Copyright (c) Гергель В.П. 03.09.2000
//
// Таблицы - базовый класс для таблиц с непрерывной памятью

#ifndef __ARRAYTAB_H
#define __ARRAYTAB_H

#include "ttable.h"

#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. к-во записей
    int CurrPos;        // номер текущего записи (нумерация от 0)
  public:
    TArrayTable(int Size=TabMaxSize) {
      pRecs = new PTTabRecord[Size];
      for ( int i=0; i<Size; i++ ) pRecs[i] = NULL;
      TabSize = Size; DataCount = CurrPos = 0;
    }
```

```
  ~TArrayTable() {
     for ( int i=0; i<DataCount; i++ )
       delete pRecs[i];
     delete [] pRecs;
   }
   // информационные методы
   virtual int IsFull() const {                      // таблица заполнена ?
     return DataCount >= TabSize;
   }
   int GetTabSize() const { return TabSize; }        // к-во записей
   // доступ
   virtual TKey GetKey( void ) const { return GetKey(CURRENT_POS); } // ключ
   virtual PTDatValue GetValuePtr( void ) const {    // указатель на значение
     return GetValuePtr(CURRENT_POS);
   }
   virtual TKey GetKey( TDataPos mode ) const;             // ключ
   virtual PTDatValue GetValuePtr( TDataPos mode ) const; // указ-ль на значен.
   // 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 GoNext для последней записи таблицы)
   virtual int SetCurrentPos ( int pos );     // установить текущую запись
   int GetCurrentPos ( void ) const {         // получить номер текущей записи
     return CurrPos;
   }
   friend TSortTable;
};
#endif

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// arraytab.cpp - Copyright (c) Гергель В.П. 03.09.2000
//
// Таблицы - базовый класс для таблиц с непрерывной памятью

#include "arraytab.h"

TKey TArrayTable :: GetKey ( TDataPos mode ) const { // значение
  int pos=-1;
  if ( ! IsEmpty() ) {
    switch ( mode ) {
      case FIRST_POS: pos = 0;              break;
      case LAST_POS:  pos = DataCount-1;    break;
      default:        pos = CurrPos;        break;
    }
  }
  return (pos==-1) ? string("") : pRecs[pos]->Key;
}             /*-------------------------------------------*/

PTDatValue TArrayTable :: GetValuePtr ( TDataPos mode ) const { // значение
  int pos=-1;
  if ( ! IsEmpty() ) {
    switch ( mode ) {
      case FIRST_POS: pos = 0;              break;
      case LAST_POS:  pos = DataCount-1;    break;
      default:        pos = CurrPos;        break;
    }
  }
  return (pos==-1) ? NULL : pRecs[pos]->pValue;
}             /*-------------------------------------------*/
```



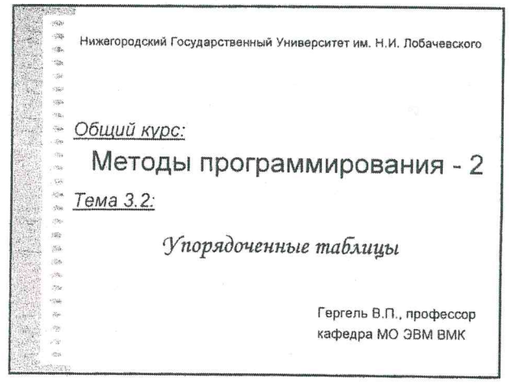
```
int TArrayTable :: Reset ( void ) { // установить на первую запись
  CurrPos = 0;
  return IsTabEnded();
}             /*-------------------------------------------*/

int TArrayTable :: IsTabEnded ( void ) const { // таблица завершена ?
  return CurrPos >= DataCount;
}             /*-------------------------------------------*/

int TArrayTable :: GoNext ( void ) { // переход к следующей записи
  if ( !IsTabEnded() ) CurrPos++;
  return IsTabEnded();
}             /*-------------------------------------------*/

int TArrayTable :: SetCurrentPos ( int pos ) { // установить текущую запись
  CurrPos = ( (pos>-1) && (pos<DataCount) ) ? pos : 0;
  return IsTabEnded();
}             /*-------------------------------------------*/
```

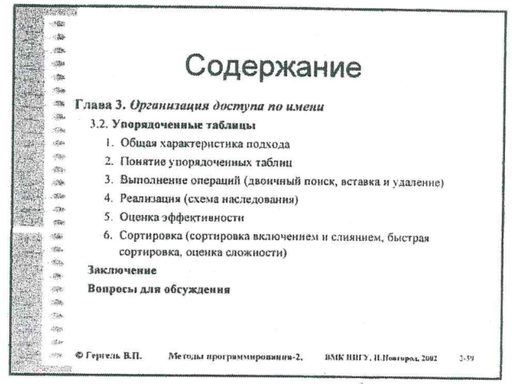
```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// scantab.h - Copyright (c) Гергель В.П. 03.09.2000
//
// Просматриваемые Таблицы

#ifndef __SCANTAB_H
#define __SCANTAB_H

#include "arraytab.h"

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

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// scantab.cpp - Copyright (c) Гергель В.П. 03.09.2000
//
// Просматриваемые Таблицы

#include "scantab.h"

PTDatValue TScanTable :: FindRecord ( TKey k ) { // найти запись
  int i;
  SetRetCode(TabOK);
  for ( i=0; i<DataCount; i++ )
    if ( pRecs[i]->Key == k ) break;
  Efficiency = i+1;
  if ( i<DataCount ) { CurrPos = i; return pRecs[i]->pValue; }
  SetRetCode(TabNoRec); return NULL;
}              /*-------------------------------------------*/

void TScanTable :: InsRecord ( TKey k, PTDatValue pVal ) { // вставить запись




}              /*-------------------------------------------*/

void TScanTable :: DelRecord ( TKey k ) { // удалить запись




}              /*-------------------------------------------*/
```

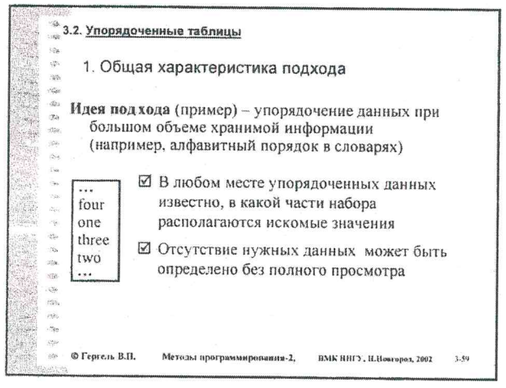


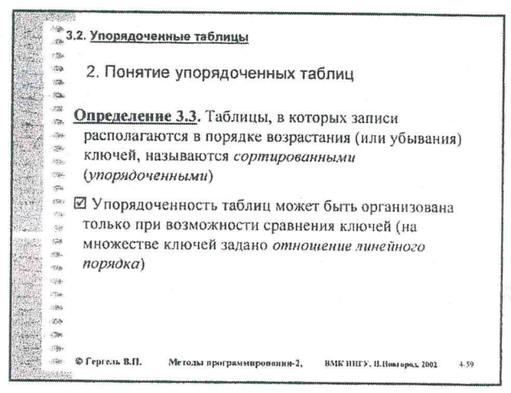
```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// Copyright (c) Гергель В.П. 03.09.2000
//
// Тестирование таблицы

#include <conio.h>
#include "tword.h"
#include "scantab.h"

  TScanTable *pTab=NULL;
  string *pKeys =NULL;
  TWord  *pWords=NULL;
  int     DataCount=0, MemSize;

// заполнение таблицы
void TableGenerator(void) {
  int  MaxKeyValue, RetCode;
  char Line[100];
  cout << "Input the record's number - ";
  cin  >> DataCount;
  cout << "Input the Maximum Key Value - ";
  cin  >> MaxKeyValue;
  MemSize = DataCount + 10;
  pTab   = new TScanTable(MemSize);
  pKeys  = new string[MemSize];
  pWords = new TWord[MemSize];
  for ( int i=0; i<DataCount; i++ ) {
    sprintf(Line,"%d",random(MaxKeyValue));
    pKeys[i]  = string(Line);
    pWords[i] = TWord("*"+pKeys[i]+"*");
    pTab->InsRecord(pKeys[i],&pWords[i]);
    if ( (RetCode = pTab->GetRetCode()) != TabOK ) {
      cout << "Retcode: " << RetCode << endl;
    }
  }
}

// выполнение операций обработки таблицы
void TableProcessor(void) {
 int com; string key;
  while ( 1 ) {
    cout << "Input Command (0-Exit, 1-Find, 2-Ins, 3-Del, 4-Print) - ";
    cin >> com;
    if ( com==0 ) break;
    if ( com!=4 ) {
      cout << "Input the key of record - ";
      cin  >> key;
    }
    if ( com==1) { // поиск
      cout << " Find " << *pTab->FindRecord(key);
      cout << " Effect  = " << pTab->GetEfficiency();
      cout << " RetCode = " << pTab->GetRetCode() << endl;
    }
    if ( com==2) { // вставка
      if ( DataCount >= MemSize )
        cout << "MemBuffer is full" << endl;
      else {
        pKeys [DataCount] = key;
        pWords[DataCount] = TWord("*"+key+"*");
        pTab->InsRecord(key,&pWords[DataCount]);
        DataCount++;
        cout << "Insert: RetCode = " << pTab->GetRetCode() << endl;
      }
```

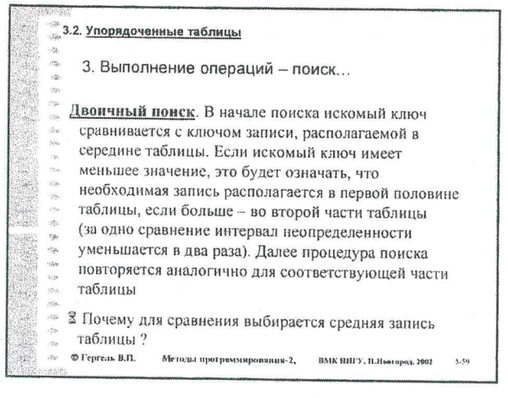
```
    }
    if ( com==3) { // удаление
      pTab->DelRecord(key);
      cout << "Delete: RetCode = " << pTab->GetRetCode() << endl;
    }
    if ( com == 4 ) cout << *pTab; // Table printing
  }
}

// оценка сложности операции вставки
void TableEvaluator(void) {
  int  IterCount, k;
  long Total=0;
  cout << "Input the iteration's number - ";
  cin  >> IterCount;
  for ( int i=0; i<IterCount; i++ ) {
    k = random(DataCount);
    pTab->FindRecord(pKeys[k]);
    Total += pTab->GetEfficiency();
  }
  cout << "Insert in the table - efficiency evaluation" << endl;
  cout << "Table Size: " << pTab->GetDataCount() << endl;
  cout << "Iterations: " << IterCount << endl;
  cout << "Average Efficiency: " << Total/double(IterCount) << endl;
}

void main() {
  cout << "Тестирование программ поддержки таблиц" << endl;
    TableGenerator();
    TableProcessor();
    TableEvaluator();
  getch();
  delete pTab;
  delete[] pKeys;
  delete[] pWords;
}
```



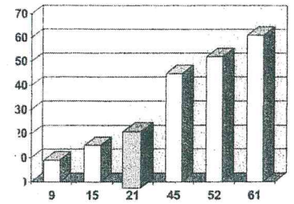






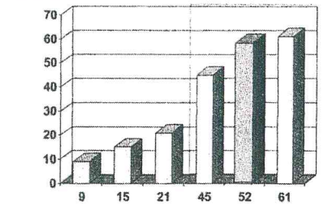
3. Выполнение операций – поиск…

Поиск записи с ключом 45



3. Выполнение операций – поиск…

Поиск записи с ключом 45



5. Оценка эффективности…

Сложность операций упорядоченных таблиц

Поиск При сравнении ключа искомой записи с ключом записи с позиции **i** таблицы размер интервала неопределенности

$$L = [(i-1)/N](i-1) + [(N-I)/N](N-i)$$

(при равной вероятности использования ключей)

- Минимальное значение величины L достигается при i = N/2

Как результат,

- $T_{min} = 1$
- $T_{max} = T_{cp} = \log_2 N$ (поиск имеющихся записей)

5. Оценка эффективности

Сложность операций упорядоченных таблиц

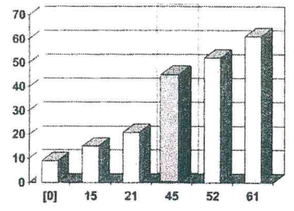
Вставка и удаление

- $T_{max} = T_{cp} = \log_2 N + N/2 + 1$

Экспериментальная оценка сложности:
программа, приложение

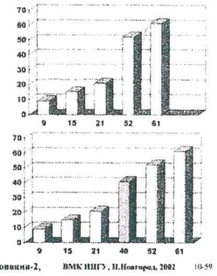
3. Выполнение операций – поиск…

Поиск записи с ключом 45



3. Выполнение операций – вставка

- ☑ поиск места вставки
- ☑ сдвиг правой части таблицы вправо (перепаковка)
- ☑ вставка новой записи



6. Сортировка…

Определение 3.4. Действия, связанные с размещением записей в порядке возрастания (или убывания) ключей называют *сортировкой*

Определение 3.5. Алгоритм сортировки называют *устойчивым* , если он никогда не меняет относительный порядок в таблице двух записей с равными ключами

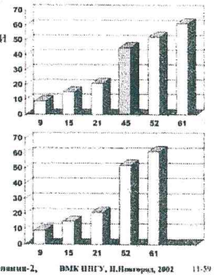
Определение 3.6. Упорядочивание данных, при которой все значения располагаются в ОП, называются *внутренней сортировкой* (проблема внешней сортировки остаются за рамками рассмотрения)

6. Сортировка…

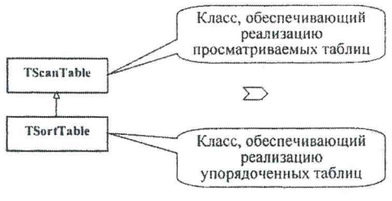
- ☑ Всего для сортировки известно более 200 алгоритмов (см., например, Кнут Д. Искусство программирования т.3 Сортировка и поиск .М: Мир, 1978г. 338 стр.)
- ☑ Далее будут рассмотрены ряд алгоритмов сортировки – при изучении методов надо постараться не просто вспомнить какой-либо ранее изученный алгоритм сортировки, а попытаться вывести его, т.е. найти путь (идею), в рамках которого был придуман (изобретён) конкретный алгоритм

3. Выполнение операций – удаление

- ☑ поиск удаляемой записи
- ☑ сдвиг правой части таблицы влево (перепаковка)



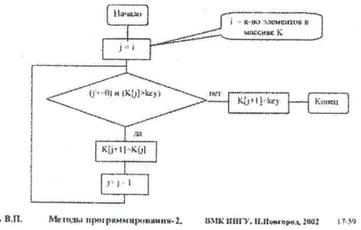
4. Реализация – схема наследования



Пример: программа, приложение

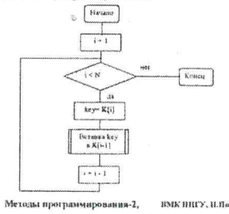
6.1. Алгоритм сортировки включением…

Идея похода – вставка нового значения в упорядоченный набор данных



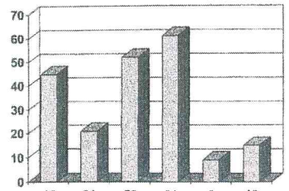
6.1. Алгоритм сортировки включением…

В ходе сортировки –упорядочиваемый набор данных разбивается на две части – упорядоченный (слева) и неупорядоченный (справа) блоки



6.1. Алгоритм сортировки включением…

- Массив до сортировки



6.1. Алгоритм сортировки включением…

- Упорядоченная часть массива состоит из одного элемента

6.1. Алгоритм сортировки включением…

- Массив отсортирован

6.1. Алгоритм сортировки включением…

Оценка сложности

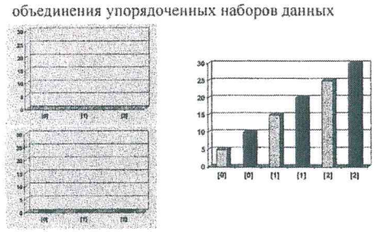
- $T_{min} = N$ (при сортировке упорядоченного массива)
- $T_{max} = N^2$ (= 1+2+…+(N-1) – для каких данных ?)
- $T_{ср}$ = ?

Реализация:

Экспериментальная оценка сложности:
программа, приложение

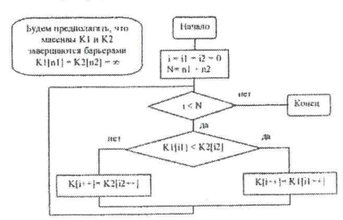
6.2. Алгоритм сортировки слиянием…

Идея подхода – возможность эффективного объединения упорядоченных наборов данных



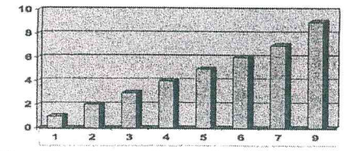
6.2. Алгоритм сортировки слиянием…

Слияние упорядоченных массивов

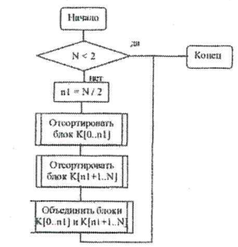


6.2. Алгоритм сортировки слиянием…

При наличии процедуры слияния **алгоритм сортировки** может быть определен *рекурсивно* – необходимо разбить упорядочиваемый набор на две равные по объему части, отсортировать их и объединить в единый массив



6.2. Алгоритм сортировки слиянием…



6.2. Алгоритм сортировки слиянием…

Оценка сложности

- ☑ N – количество сравнений при слиянии
- ☑ $\log_2 N$ – количество уровней рекурсии
- $T_{min} = T_{max} = T_{ср} = N \log_2 N$

Реализация:

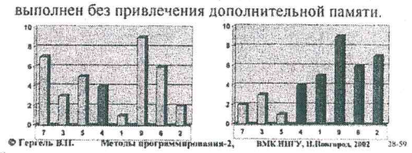
Экспериментальная оценка сложности:
программа, приложение

- Процедура слияния использует дополнительную память. **Пространственная сложность алгоритма (сложность по памяти)**

$$V = N$$

6.3. Алгоритм быстрой сортировки…

Идея подхода (*Hoare C.A.R.*)– использование процедуры разделения упорядочиваемого набора на две части, в одной из которых располагаются значения, меньшие некоторого порогового (*ведущего*) элемента массива, в другой – соответственно большие значения. Подобный способ разделения может быть выполнен без привлечения дополнительной памяти.

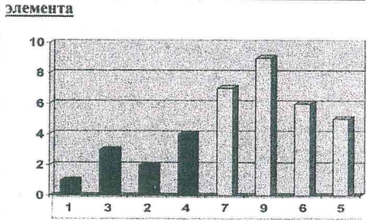


6.3. Алгоритм быстрой сортировки…

```
// Разделение массива с использованием ведущего элемента
key = K[0];   // ведущий элемента
i1=1; i2=N-1; // индексы левого (i1) и правого (i2) блоков
// цикл, пока разделяемые блоки не пересекутся
while ( i1 <= i2 ) {
  // пока K[i1] не превышает вед. эл-т, переход вправо
  while ( (i1<N) && (K[i1] <= key) ) i1++;
  // пока K[i2] меньше вед. эл-та, переход влево
  while ( K[i2] > key ) i2--;
  // перестановка значений, которые
  // приостановили разделение массива
  if ( i1 < i2 ) { kt = K[i1]; K[i1] = K[i2]; K[i2] = kt; }
}
// установка ведущего элемента между блоками
K[0] = K[i2]; K[i2] = key;
i    = i2;  // индекс ведущего элемента
```


6.3. Алгоритм быстрой сортировки…

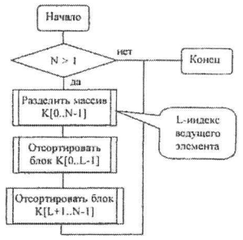
Разделение массива с использованием ведущего элемента



6.3. Алгоритм быстрой сортировки…

При наличии процедуры разделения **алгоритм сортировки** может быть определен *рекурсивно* – необходимо разбить упорядочиваемый набор на два блока с меньшими и большими значениями соответственно и затем последовательно отсортировать полученные блоки

6.3. Алгоритм быстрой сортировки…



6.3. Алгоритм быстрой сортировки…

Оценка сложности

- $T_{min} = N \log_2 N$
- $T_{max} = N^2$ (!!!)
- $T_{ср}$ = ?

Вероятность выбора любого ключа в качестве ведущего элемента является одинаковой

$$T(N) \le \alpha N + \frac{1}{N}\sum_{i=1}^{N-1}[T(i-1)+T(N-i)] = \alpha N + \frac{2}{N}\sum_{i=0}^{N-1}T(i)$$

Докажем по индукции, что

$T(N) \le KN \ln N$ ($K = 2\alpha + 2\beta$, $\beta = T(0) = T(1)$)

$$\Rightarrow T(N) \le \alpha N + \frac{2}{N}(T(0)+T(1)) + \frac{2}{N}\sum_{i=2}^{N-1} Ki \ln i$$

6.3. Алгоритм быстрой сортировки…

$$T.к. \frac{2}{N}\sum_{i=2}^{N-1} Ki \ln i \le \int_2^N x \ln x\,dx \le \frac{N^2 \ln N}{2} - \frac{N^2}{4}$$

$$\Rightarrow T(N) \le \alpha N + \frac{4\beta}{N} + \frac{2K}{N}\left(\frac{N^2 \ln N}{2} - \frac{N^2}{4}\right) = \alpha N + \frac{4\beta}{N} + KN \ln N - \frac{KN}{2}$$

$$T.к. \alpha N + \frac{4\beta}{N} \le \frac{KN}{2} \quad (N \le 2)$$

$$\Rightarrow T(N) \le KN \ln N \le KN \log_2 N$$

Оценка сложности $T_{ср} = 1.4(N+1)\log_2 N$

6.3. Алгоритм быстрой сортировки…

Реализация: ⋗

Экспериментальная оценка сложности:
программа, приложение

6.4. Оценка сложности сортировки…

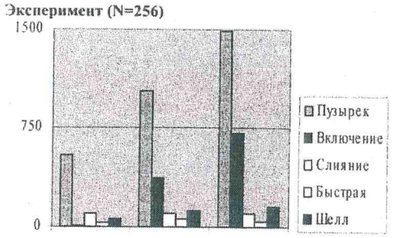
Эксперимент (N=256)*)

	Упор.	Случайн.	Упор. в обр. пор.
Пузырек	540	1026	1492
Включение	12	366	704
Слияние	99	102	99
Быстрая	31	60	37
Шелл	58	127	157

*) Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985.

6.4. Оценка сложности сортировки…

Эксперимент (N=256)



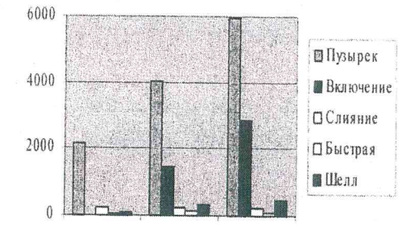
6.4. Оценка сложности сортировки…

Эксперимент (N=512)

	Упор.	Случайн.	Упор. в обр. пор.
Пузырек	2165	4054	5931
Включение	23	1444	2836
Слияние	234	242	232
Быстрая	69	146	79
Шелл	116	349	492

6.4. Оценка сложности сортировки…

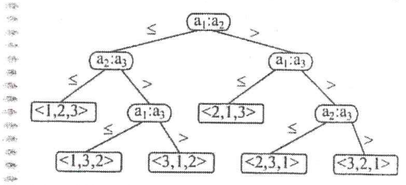
Эксперимент (N=512)



6.4. Оценка сложности сортировки…

Оценка минимальной сложности

Представим действия, выполняемые при сортировке данных, в виде разрешающего дерева (*decision tree*)



6.4. Оценка сложности сортировки…

Оценка минимальной сложности

Теорема 3.1. Высота любого разрешающего дерева, сортирующего N элементов, есть $\Omega(N\log_2 N)$

Доказательство.

- Количество листьев N! (количество перестановок из N элементов)
- Двоичное дерево высоты h имеет 2^h листьев

$$\Rightarrow N! \le 2^h$$

- Используя формулу Стирлинга $N! > (N/e)^N$, можно заключить

$$h \ge N\log_2 N - N\log_2 e = \Omega(N\log_2 N)$$

7. Заключение

- Упорядоченные таблицы представляют собой эффективный способ организации таблиц при большом количестве имеющихся записей
- Временная сложность поиска $\log_2 N$
- Временная сложность вставки и удаления N
- Временная сложность сортировки $N\log_2 N$
- При анализе вычислений следует учитывать *сложность алгоритмов по памяти*

↳ Упорядоченные таблицы целесообразно использовать для таблиц, в которых достаточно редко выполняются операции вставки и удаления

Вопросы для обсуждения

- Алгоритмы сортировки с линейной сложностью вычислений
- Возможные способы повышения эффективности операций вставки и удаления

Темы заданий для самостоятельной работы

- Рассмотрение дополнительных методов сортировки (пирамидальная сортировка)
- Проведение вычислительных экспериментов

Следующая тема

- Представление таблиц с использованием деревьев поиска


```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// sorttab.h - Copyright (c) Гергель В.П. 04.09.2000
//
// Упорядоченные Таблицы

#ifndef __SORTTAB_H
#define __SORTTAB_H

#include "scantab.h"

enum TSortMethod { INSERT_SORT, MERGE_SORT, QUICK_SORT };

class TSortTable : public TScanTable {
  protected:
    TSortMethod SortMethod; // индекс метода 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 QuickSort  (PTTabRecord *pMem, int DataCount); // быстрая сортировка
    void QuickSplit (PTTabRecord *pData, int Size, int &Pivot);
  public:
    TSortTable(int Size=TabMaxSize) : TScanTable(Size) {};
    TSortTable(const TScanTable &tab);
    TSortTable & operator =(const TScanTable &tab);    // присваивание
    TSortMethod GetSortMethod(void)    { return SortMethod; }
    void SetSortMethod(TSortMethod sm) { SortMethod = sm;   }
    // основные методы
    virtual PTDatValue FindRecord ( TKey k );                    // найти запись
    virtual void       InsRecord  ( TKey k, PTDatValue pVal ); // вставить
запись
    virtual void       DelRecord  ( TKey k );                    // удалить запись
};
#endif

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// sorttab.cpp - Copyright (c) Гергель В.П. 04.09.2000
//
// Упорядоченные Таблицы

#include "Sorttab.h"

// создание упорядоченной таблицы по просматриваемой таблице
TSortTable :: TSortTable(const TScanTable &tab) {
  *this = tab;
}
// присвание упорядоченной таблицы значения просматриваемой таблицы
TSortTable & TSortTable :: operator =(const TScanTable &tab) {
  if ( pRecs != NULL ) {
    for ( int i=0; i<DataCount; i++ ) delete pRecs[i];
    delete [] pRecs; pRecs = NULL;
  }
  TabSize   = tab.GetTabSize();
  DataCount = tab.GetDataCount();
  pRecs = new PTTabRecord[TabSize];
  for ( int i=0; i<DataCount; i++ )
    pRecs[i] = (PTTabRecord)tab.pRecs[i]->GetCopy();
  SortData();
  CurrPos = 0;
  return *this;
}
```

```
PTDatValue TSortTable :: FindRecord ( TKey k ) { // найти запись
  // двоичный поиск
  int i, i1=0, i2=DataCount-1; // границы области поиска
```

```
}              /*-----------------------------------------------*/

void TSortTable :: InsRecord ( TKey k, PTDatValue pVal ) { // вставить запись
  if ( IsFull() ) SetRetCode(TabFull);
  else {
    PTDatValue temp = FindRecord(k);
    if ( RetCode == TabOK ) SetRetCode(TabRecDbl);
    else {
      SetRetCode(TabOK);
      for ( int i=DataCount; i>CurrPos; i-- ) pRecs[i] = pRecs[i-1]; //
перепаковка
      pRecs[CurrPos] = new TTabRecord(k,pVal);
      DataCount++;
    }
  }
}              /*-----------------------------------------------*/

void TSortTable :: DelRecord ( TKey k ) { // удалить запись
```

```
}              /*-----------------------------------------------*/

void TSortTable :: SortData(void) { // сортировка данных
  Efficiency = 0;
  switch ( SortMethod ) {
    case INSERT_SORT:
      InsertSort(pRecs, DataCount); // сортировка включением
      break;
    case MERGE_SORT:
      MergeSort(pRecs, DataCount);  // сортировка слиянием
      break;
    case QUICK_SORT:
      QuickSort(pRecs, DataCount);  // быстрая сортировка
      break;
  }
}
```



```
// сортировка вставками
void TSortTable :: InsertSort(PTTabRecord *pRecs, int DataCount) {
  PTTabRecord pR;
  Efficiency = DataCount;
  for ( int i=1, j; i<DataCount; i++ ) { // блок массива до индекса i уже
упордочен
    pR = pRecs[i]; // добавляемый элемент
    for ( j=i-1; j>-1; j-- )
      if ( pRecs[j]->Key > pR->Key ) {
        pRecs[j+1] = pRecs[j]; // сдвиг вправо
        Efficiency++;
      } else break;
    pRecs[j+1] = pR; // вставка
  }
}

// сортировка слиянием
void TSortTable :: MergeSort(PTTabRecord *pRecs, int DataCount) {
  PTTabRecord *pData = pRecs;
  PTTabRecord *pBuff = new PTTabRecord[DataCount];
  PTTabRecord *pTemp = pBuff;
  MergeSorter(pData,pBuff,DataCount);
  if ( pData == pTemp ) // отсортированные данные находятся в буфере
    for ( int i=0; i<DataCount; i++ ) pBuff[i] = pData[i];
  delete pTemp;
}

// сортировка слиянием - служебный метод
void TSortTable :: MergeSorter(PTTabRecord * &pData,PTTabRecord * &pBuff,int
Size) {
  int n1 = (Size+1) / 2;
  int n2 = Size - n1;
  if ( Size > 2 ) {
    PTTabRecord *pDat2=pData+n1, *pBuff2=pBuff+n1;
    MergeSorter(pData,pBuff,n1);  // сортировка "половинок" массива
    MergeSorter(pDat2,pBuff2,n2);
  }
  MergeData(pData,pBuff,n1,n2);   // слияние упорядоченных "половинок" массива
}

// слияние упорядоченных массивов
void TSortTable :: MergeData(PTTabRecord * &pData,PTTabRecord * &pBuff,int n1,
int n2) {
// сливаемые массивы располагаются по указателю pData последовательно
// (n1 - размер первого массива, n2 - размер второго массива)
// pBuff - массив, в котором располагаются объединенные данные
// после слияния указатели pData и pBuff обмениваются значениями,
// т.е. pData указывает на объединенные данные, а pBuff - на область памяти
// с исходными двумя массивами (далее эта память может использоваться как
// рабочий буфер для последующих слияний упорядоченных данных
```

```
}
```

```
// быстрая сортировка
void TSortTable :: QuickSort(PTTabRecord *pRecs, int DataCount) {
  int pivot;  // индекс ведущего элемента
  int n1, n2; // размеры разделенных блоков данных
  if ( DataCount > 1 ) {
    QuickSplit(pRecs,DataCount,pivot); // разделение
    n1 = pivot + 1;
    n2 = DataCount - n1;
    QuickSort(pRecs,n1-1);  // сортировка разделенных блоков массива
    QuickSort(pRecs+n1,n2);
  }
}

// быстрая сортировка - выбор ведущего элемента и разделение данных
void TSortTable :: QuickSplit(PTTabRecord *pData, int Size, int &Pivot) {
  PTTabRecord pPivot = pData[0], pTemp; // pPivot - указатель на ведущий элемент
  int i1=1, i2=Size-1; // индексы левого (i1) и правого (i2) блоков
  while ( i1 <= i2 ) { // цикл, пока блоки не пересекутся
    // пока pData[i1]->Key не превышает ключа вед. эл-та, переход вправо
    while ( (i1<Size) &&  !(pData[i1]->Key > pPivot->Key) ) i1++;
    // пока pData[i2]->Key меньше ключа вед. эл-та, переход влево
    while ( pData[i2]->Key > pPivot->Key ) i2--;
    // перестановка элементов, на которых произошла остановка разделения
    if ( i1 < i2 ) {
      pTemp     = pData[i1];
      pData[i1] = pData[i2];
      pData[i2] = pTemp;
    }
  }
  // установка ведущего элемента между блоками
  pData[0]  = pData[i2];
  pData[i2] = pPivot;
  Pivot     = i2; // i2 - индекс позиции ведущего элемента
  Efficiency += Size;
}
```


Нижегородский Государственный Университет им. Н.И. Лобачевского

Общий курс:

Методы программирования - 2

Тема 3.3:

Представление таблиц с использованием деревьев поиска

Гергель В.П., профессор
кафедра МО ЭВМ ВМК

Содержание

Глава 3. *Организация доступа по имени*

3.3. Представление таблиц с использованием деревьев поиска

1. Общая характеристика подхода
2. Понятие деревьев поиска
3. Структура хранения
4. Реализация (схема наследования, алгоритмы обхода)
5. Выполнение операций (поиск, вставка, удаление, итератор)
6. Оценка эффективности

3.3А. Сбалансированные деревья поиска

1. Понятие сбалансированного дерева
2. Балансировка дерева при вставке новых узлов

Заключение

Вопросы для обсуждения

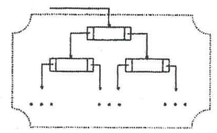
3.3. Деревья поиска

1. Общая характеристика подхода…

- ☑ Сложность вставки и удаления в упорядоченных таблицах вызвана использованием непрерывной памяти, что приводит, как следствие, к необходимости перепаковок данных
- ☑ Устранение перепаковок возможно только при использования списковой памяти, но в этом случае теряется возможность прямого доступа к данным
- ☞ Достижение эффективности двоичного поиска при списковой структуре хранения требует прямого доступа к звену со средней записью таблицы; из этого звена должна существовать возможность доступа (*указатели*) к средним звеньям левой и правой частей таблицы и т.д.

3.3. Деревья поиска

1. Общая характеристика подхода



"Вычисленная" структура хранения является *деревом поиска*

3.3. Деревья поиска

2. Понятие деревьев поиска…

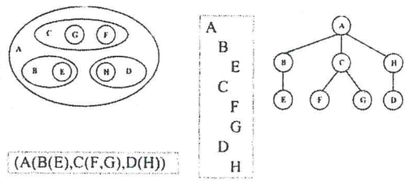
Определение 3.7. Связный граф без циклов называется *деревом*.

Определение 3.7'. Структура типа *дерева* (*древовидная структура*) с базовым типом T – это
- либо пустая структура,
- либо узел (вершина) со значением типа T, с которым связано конечное число древовидных структур (*поддеревьев*) с базовым типом T.

3.3. Деревья поиска

2. Понятие деревьев поиска…

Известны несколько различных *способов изображения* деревьев



2. Понятие деревьев поиска…

Определение 3.7-1. Узел y, который находится непосредственно под узлом x (т.е. есть ребро (x,y)), называется (непосредственным) *потомком* x, а x называется (непосредственным) *предком* y.

Определение 3.7-2. Узел, у которого нет предков, называется *корнем*.

Определение 3.7-3. Узел, у которого нет потомков, называется *листом*.

Определение 3.7-4. Число ветвей (ребер), которые нужно пройти, чтобы продвинуться от корня к узлу x,.называется *длиной пути* к x.

2. Понятие деревьев поиска…

Определение 3.7-5. Узлы с одинаковой длиной пути образуют уровень (ярус) дерева. Корень расположен на уровне 1, его потомки на уровне 2 и т.д. (принято изображать узлы дерева одного и того же уровня на одной горизонтальной прямой) .

Определение 3.7-6. Максимальный уровень дерева называется его *глубиной*.

Определение 3.7-7. Число непосредственных потомков узла называется *степенью узла*. Максимальная степень всех узлов есть *степень дерева*.

2. Понятие деревьев поиска…

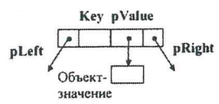
Определение 3.7-8. *Упорядоченное дерево* – это дерево, у которого ветви каждого узла упорядочены.

Определение 3.7-9. Упорядоченное дерево степени 2 называется *бинарным деревом*.

Определение 3.7-10. Если для любой вершины бинарного дерева значения в левом потомке меньше значения узла, а значение в правом потомке больше значения узла, то такое дерево называется ***деревом поиска***.

3. Структура хранения

```
class TTreeNode : public TTabRecord {
  protected: // указатели на поддеревья
    PTTreeNode pLeft, pRight;
  public:
    TTreeNode ( TKey k="", PTDatValue pVal=NULL,
      PTTreeNode pL=NULL, PTTreeNode pR=NULL );
    PTTreeNode  GetLeft  ( void ) const;
    PTTreeNode  GetRight ( void ) const;
};
```



4. Реализация – схема наследования

4. Реализация - алгоритмы обхода…

☑ Обработка дерева – выполнение необходимой операции для каждой узла дерева

↳ Реализация подобного типа действий предполагает умение *обхода* (*обхода*) дерева

Представим дерево в общем виде

T – top (корень)
L – left (левое поддерево)
R – right (правое поддерево)

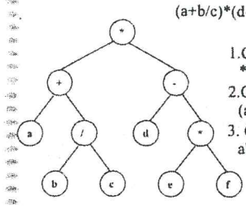
Тогда возможны следующие варианты обхода

1. T, L, R – сверху вниз
2. L, T, R – слева направо
3. L, R, T – снизу вверх
4. T, R, L – сверху вниз
5. R, T, L – справа налево
6. R, L, T – снизу вверх

4. Реализация - алгоритмы обхода…

Пример: Представление арифметического выражения в виде бинарного дерева

(a+b/c)*(d-e*f)



1.Сверху вниз TLR
*+a/bbc-d*ef - префиксная
2.Слева направо LTR
(a+b/c)*(d-e*f) - инфиксная
3. Снизу вверх LRT
abc/+def*-* - постфиксная

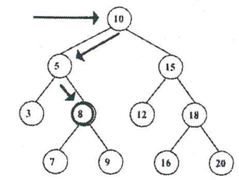
4. Реализация - алгоритмы обхода

Пример: Печать значений дерева поиска (схема LTR, рекурсия)

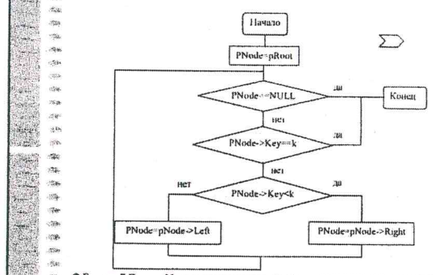
```
void TTreeTable :: PrintTreeTab(ostream
  &os, PTTreeNode pNode) {
   if ( pNode != NULL ) { // печать
  дерева с вершиной pNode
      PrintTreeTab(os,pNode->pLeft);
      pNode->Print(os); os << endl;
      PrintTreeTab(os,pNode->pRight);
   }
}
```


5. Выполнение операций – поиск…

Пример: Поиск записи с ключом **8**



5. Выполнение операций – поиск…

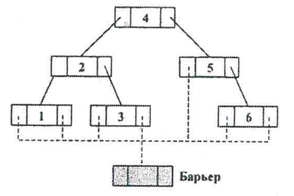


5. Выполнение операций – поиск…

```
// поиск в дереве поиска - рекурсия
PTTreeNode
 FindRecord(TKey k,PTTreeNode pNode){
 if ( pNode != NULL ) { // лист
  if ( pNode->Key < k ) // вправо
    pNode = FindRecord(k,pNode->Right);
  if ( pNode->Key > k ) // влево
    pNode = FindRecord(k,pNode->Left);
 }
 return pNode;
}
```


5. Выполнение операций – поиск…

Введение барьера



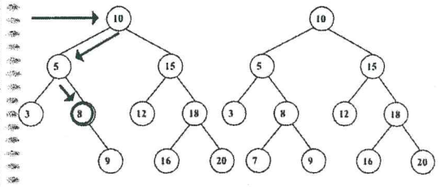
5. Выполнение операций – поиск

Введение барьера

```
// поиск в дереве поиска - рекурсия
PTTreeNode
 FindRecord(TKey k,PTTreeNode pNode){
 if ( pNode->Key < k ) // вправо
   pNode = FindRecord(k,pNode->Right);
 if ( pNode->Key > k ) // влево
   pNode = FindRecord(k,pNode->Left);
 return (pNode==pBarrier) ? NULL : pNode;
}
```

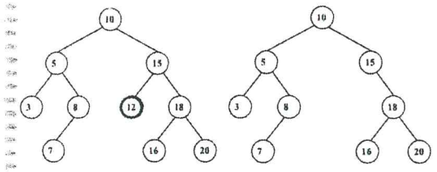

5. Выполнение операций – вставка…

Поиск до тупика и вставка (ключ 7)



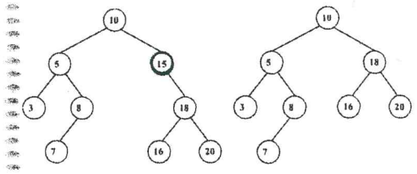
5. Выполнение операций – удаление…

Удаление листа (ключ 12)



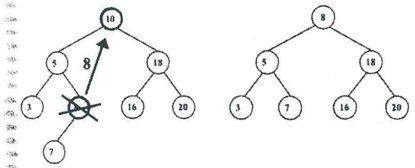
5. Выполнение операций – удаление…

Удаление узла с одним потомком (ключ 15)



5. Выполнение операций – удаление

Удаление узла с двумя потомком (ключ 10)



5. Выполнение операций – итератор…

- ☑ Схема обхода LTR, нерекурсивный вариант
- ☑ Неиспользованные узлы пути до текущей вершины запоминаются в стеке (*фиксатор пути*)

5. Выполнение операций – итератор…

- ☑ Инициализация (Reset) – переход к крайнему левому узлу

```
PTTreeNode pNode = pCurrent = pRoot;
CurrPos = 0;
while ( pNode != NULL ) {
  St.push(pNode);
  pCurrent = pNode;
  pNode=pNode->GetLeft();
}
```


5. Выполнение операций – итератор…

- ☑ Проверка завершения (IsTabEnded) – оценка номера текущего узла

```
return CurrPos >= DataCount;
```


5. Выполнение операций – итератор…

- ☑ Переход к следующему узлу (GoNext) от узла pCurrent (располагается на вершине стека)
  - Переход к правому потомку
  - Если правый потомок имеется, то переход к крайней левой вершине
  - Если правого потомка нет, выборка узла из стека

5. Выполнение операций – итератор

- ☑ Переход к следующему узлу (GoNext)

```
if (!IsTabEnded() && (pCurrent != NULL)) {
 pNode = pCurrent = pCurrent->GetRight();
 St.pop(); // исключение из стека
 while ( pNode != NULL ) {
   // переход к крайней левой вершине
   St.push(pNode); // добавить в стек
   pCurrent = pNode;
   pNode=pNode->GetLeft();
 }
 if ( (pCurrent==NULL) && !St.empty() )
   pCurrent = St.top();
 CurrPos++;
}
```


6. Оценка эффективности…

- $T_{min} = 1$
- $T_{max} = \log_2 N$ (при сбалансированном дереве)
- $T_{max} = N$ (при вырожденном дереве)
- $T_{ср} = ?$

6. Оценка эффективности…

Пусть даны N различных ключей со значениями 1,…,N и появление любого ключа равновероятно.

Пусть первый ключ равен i. Левое поддерево будет содержать $(i-1)$ узлов, правое поддерево - $(n-i)$ узел.

$a_N = \frac{1}{N}\sum_{i=1}^{N} a_N^i$ - средняя длина пути для дерева с N узлами,

где a_N^i есть средняя длина пути в дереве, в котором корень равен i.

Оценим

$a_N^i = (a_{i-1}+1)\frac{i-1}{N} + 1\frac{1}{N} + (a_{N-i}+1)\frac{N-i}{N}$

6. Оценка эффективности…

Тогда

$$a_N = \frac{1}{N}\sum_{i=1}^{N} a'_N = \frac{1}{N}\sum_{i=1}^{N}((a_{i-1}+1)\frac{i-1}{N} + 1\frac{1}{N} + (a_{N-i}+1)\frac{N-i}{N}) =$$

$$= \frac{1}{N}(N + \frac{1}{N}\sum_{i=1}^{N}[(i-1)a_{i-1} + (n-i)a_{N-i}]) =$$

$$= 1 + \frac{2}{N*N}\sum_{i=1}^{N}(i-1)a_{i-1} = 1 + \frac{2}{N*N}\sum_{i=1}^{N-1} ia_i$$

6. Оценка эффективности…

Из последнего выражения следует

(1) $a_N = 1 + \frac{2}{N*N}\sum_{i=1}^{N-1} ia_i = 1 + \frac{2}{N*N}(N-1)a_{N-1} + \frac{2}{N*N}\sum_{i=1}^{N-2} ia_i$

(2) $a_{N-1} = 1 + \frac{2}{(N-1)*(N-1)}\sum_{i=1}^{N-2} ia_i$ - умножим на $((N-1)/N)^2$

(3) $\frac{2}{N*N}\sum_{i=1}^{N-2} ia_i = (\frac{N-1}{N})^2(a_{N-1}-1)$ - подставим (3) в (1)

(4) $a_N = \frac{2}{N*N}((N^2-1)a_{N-1} + 2N - 1)$

6. Оценка эффективности…

Отсюда можно получить (*проверяется подстановкой*)

$a_N = 2\frac{N+1}{N}H_N - 3, \quad H_N = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N}$

$H_N = \gamma + \ln N + \frac{1}{12N^2} + \ldots$ (формула Эйлера, $\gamma \cong 0{,}577$)

$(N >> 1) \Rightarrow a_N \cong 2[\ln N + \gamma] - 3 = 2\ln N - c$

6. Оценка эффективности…

Пусть $a^*_N = \log_2 N$ есть средняя длина пути для идеально сбалансированного дерева

$\lim_{N\to\infty} \frac{a_N}{a^*_N} = \frac{2\ln N}{\log_2 N} = 2\ln 2 = 1{,}386$

Экспериментальная оценка сложности:
программа, приложение

1. Понятие сбалансированного дерева…

Определение 3.8. Дерево является *идеально сбалансированным*, если для каждого его узла количество узлов в левом и правом поддеревьях различаются не более, чем на 1

Определение 3.9. (Адельсон-Вельский, Ландис) Дерево является *сбалансированным*, если для каждого его узла высота левого и правого поддеревьев различаются не более, чем на 1 (*АВЛ-деревья*)

1. Понятие сбалансированного дерева

Свойства.

☑ Идеально сбалансированные деревья являются сбалансированными

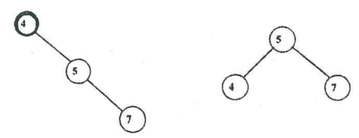
☑ Операции обработки сбалансированных деревьев (поиск, вставка, удаление) имеют сложность $\log_2 N$

Теорема 3.2. $h_{АВЛ} \le 1.45\, h_{ИСД}$

$\log_2(N+1) \le h(N) \le 1.4404 \log_2(N+2) - 0.328$

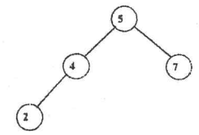
2. Балансировка при вставке - примеры…

Вставка ключа 7 **Балансировка ?**



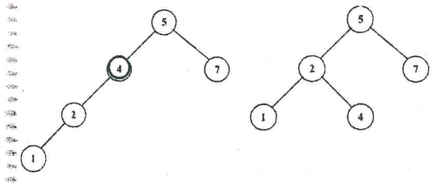
2. Балансировка при вставке - примеры…

Вставка ключа 2 **Балансировка ?**



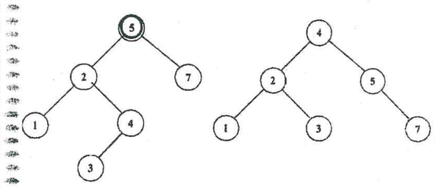
2. Балансировка при вставке - примеры…

Вставка ключа 1 **Балансировка ?**



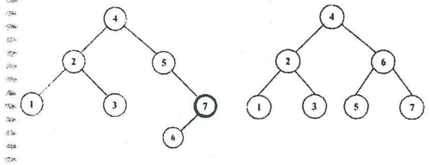
2. Балансировка при вставке - примеры…

Вставка ключа 3 **Балансировка ?**



2. Балансировка при вставке - примеры…

Вставка ключа 6 **Балансировка ?**



2. Балансировка при вставке – общий анализ…

☑ Пусть дано сбалансированное дерево, в котором рассмотрим поддерево с корнем в некотором узле r.

☑ Обозначим через L и R левую и правую части этого поддерева с высотами h_L и h_R соответственно.

☑ Пусть высота h_L после вставки в L нового узла увеличилась на 1.

☟ Возможны три ситуации:

1) $h_L = h_R$: после вставки высоты L и R разные, но условие балансировки не нарушено

2) $h_L < h_R$: после вставки $h_L = h_R$ и балансировка поддеревьев улучшается

3) $h_L > h_R$: после вставки условие балансировки нарушено, необходима балансировка

2. Балансировка при вставке – структура хранения

```
class TBalanceNode : public TTreeNode {
  protected:
    int Balance; // индекс балансировки
  public:
    TBalanceNode ();
    int  GetBalance(void) const;
    void SetBalance(int bal);
}
// Balance = h_R - h_L = (-1, 0, +1)
```


2. Балансировка при вставке – алгоритм

- ☑ Следовать по пути поиска, пока не окажется, что ключа нет в дереве
- ☑ Включить новый узел о определить новый показатель сбалансированности
- ☑ Пройти обратно по пути поиска и проверить показатель сбалансированности у каждого узла

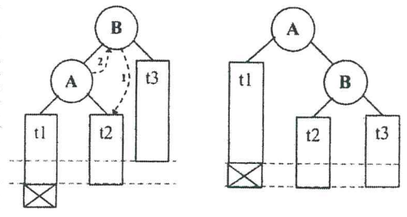
↳При необходимости балансировки после вставки (для примера, в левое поддерево) можно выделить две ситуации:

(1) высота левого поддерева левого узла больше (т.е. Balance = -1)

(1) высота левого поддерева левого узла меньше (т.е. Balance = +1)

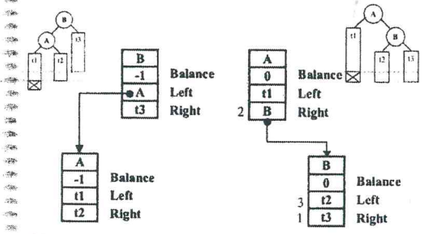
2. Балансировка при вставке – общая схема 1…

Однократный левый (LL) поворот



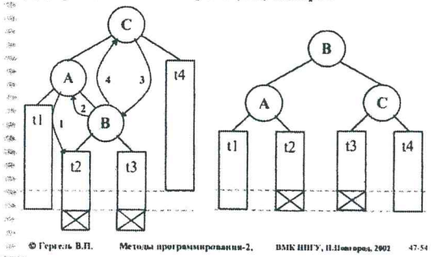
2. Балансировка при вставке – общая схема 1…

Однократный левый (LL) поворот



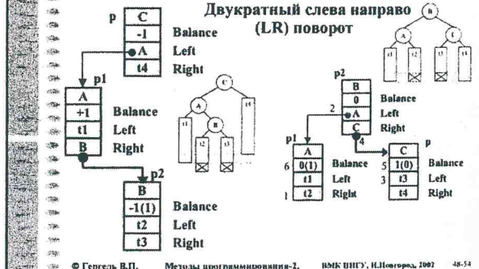
2. Балансировка при вставке – общая схема 2…

Двукратный слева направо (LR) поворот



2. Балансировка при вставке – общая схема 2…

Двукратный слева направо (LR) поворот



2. Балансировка при вставке – реализация

3. Оценка сложности

- ☑ Эмпирическая проверка показывает, что $h_{АВЛ} = \log_2 N + c$ (c≈0.25)
- ☑ В среднем балансировка при каждом втором включении, причем однократный и двукратный поворот равновероятны

Экспериментальная оценка сложности:
программа, приложение

7. Заключение

- Представление таблиц при помощи деревьев поиска обеспечивает сложность в среднем $\log_2 N$ для всех операций обработки (поиска, вставки и удаления)
- Максимальная сложность обработки деревьев поиска имеет порядок N
- Сбалансированные деревья поиска обеспечивает сложность порядка $\log_2 N$ для любых вариантов исходных данных
- Идеально сбалансированные деревья требуют больших накладных расходов для балансировки

Вопросы для обсуждения

- Сложность алгоритмов для разных классов исходных данных
- Методы оценки сложности алгоритмов
- Возможные способы дальнейшего повышения эффективности операций обработка таблиц

Темы заданий для самостоятельной работы

- Балансировка деревьев поиска при удалении узлов
- Проведение вычислительных экспериментов

Следующая тема

- Таблицы с вычислимым входом

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// treenode.h Copyright (c) Гергель В.П. 04.09.2000
//
// Таблицы - базовый (абстрактный) класс объектов-значений для деревьев

#ifndef __TREENODE_H
#define __TREENODE_H

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

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 { return pLeft;  }
    PTTreeNode  GetRight ( void ) const { return pRight; }
    virtual  TDatValue * GetCopy(); // изготовить копию
    friend class TTreeTable;
    friend class TBalanceTree;
};
#endif
// end of treenode.h
```

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// treenode.cpp Copyright (c) Гергель В.П. 04.09.2000, 16.01.2003
//
// Таблицы - базовый (абстрактный) класс объектов-значений для деревьев

#include "treenode.h"

TDatValue * TTreeNode :: GetCopy() { // изготовить копию
  TTreeNode *temp = new TTreeNode(Key,pValue,NULL,NULL);
  return temp;
}
// end of treenode.cpp
```

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// treetab.h - Copyright (c) Гергель В.П. 04.09.2000
//
// Таблицы со структурой хранения в виде деревьев поиска

#ifndef __TREETAB_H
#define __TREETAB_H

#include <stack>
#include "ttable.h"
#include "treenode.h"

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 PrintTreeTab (ostream &os, PTTreeNode pNode); // Печать
    void DrawTreeTab  (PTTreeNode pNode, int Level);   // Печать с ярусами
    void DeleteTreeTab(PTTreeNode pNode);              // Удаление
  public:
    TTreeTable() : TTable() { CurrPos=0; pRoot=pCurrent=NULL; ppRef=NULL; }
    ~TTreeTable() { DeleteTreeTab(pRoot); }
    // 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       DelRecord  ( TKey k ); // удалить запись
    // навигация
    virtual TKey GetKey( void ) const;        // ключ текущей записи
    virtual PTDatValue GetValuePtr( void ) const; // указатель на значение
    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 GoNext для последней записи таблицы)
    // Печать таблицы
    friend ostream& operator<<(ostream &os,TTreeTable &tab );
    void Draw(void); // печать дерева (рисунок слева направо)
    void Show(void); // печать дерева (рисунок сверху вниз)
protected:
  string tk[20];
  int tl[20], pos;
  void PutValues(PTTreeNode pNode,int Level); // служебный метод lkz Show
};
#endif
```

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// treetab.cpp - Copyright (c) Гергель В.П. 04.09.2000, 16.01.2003
//
// Таблицы со структурой хранения в виде деревьев поиска

#include "treetab.h"

int TTreeTable :: IsFull() const { // заполнена ?
  PTTreeNode pNode = new TTreeNode();
  int temp = pNode == NULL;
  delete pNode;
  return temp;
}             /*--------------------------------------------*/

PTDatValue TTreeTable :: FindRecord ( TKey k ) { // найти запись
  PTTreeNode pNode = pRoot;
  ppRef = &pRoot; // адрес указателя на вершину-результата
  Efficiency = 0;
  while ( pNode != NULL ) {
    Efficiency++;
    if ( pNode->Key == k ) break;
    if ( pNode->Key <  k ) ppRef = &pNode->pRight;
    else ppRef = &pNode->pLeft;
    pNode = *ppRef;
  }
  SetRetCode(TabOK); return (pNode==NULL) ? NULL : pNode->pValue;
}             /*--------------------------------------------*/

void TTreeTable :: InsRecord ( TKey k, PTDatValue pVal ) { // вставить запись
  if ( IsFull() ) SetRetCode(TabFull);
  else if ( FindRecord(k) != NULL ) SetRetCode(TabRecDbl);
  else {
```

```
    SetRetCode(TabOK);
    *ppRef = new TTreeNode(k,pVal);
    DataCount++;
  }
}               /*-----------------------------------------------*/

void TTreeTable :: DelRecord ( TKey k ) { // удалить запись

}               /*-----------------------------------------------*/

// навигация
TKey TTreeTable :: GetKey( void ) const { // значение ключа текущей записи
  return ( pCurrent==NULL) ? string("") : pCurrent->Key;
}

PTDatValue TTreeTable :: GetValuePtr( void ) const { // указатель на значение
  return ( pCurrent==NULL) ? NULL : pCurrent->pValue;
}

int TTreeTable :: Reset ( void ) { // установить на первую запись
  PTTreeNode pNode = pCurrent = pRoot;
  while ( !St.empty() ) St.pop(); // очистка стека
  CurrPos = 0;
  while ( pNode != NULL ) { // переход к крайней левой вершине
    St.push(pNode); pCurrent = pNode; pNode=pNode->GetLeft();
  }
  SetRetCode(TabOK);
  return IsTabEnded();
}

int TTreeTable :: IsTabEnded ( void ) const { // таблица завершена ?
  return CurrPos >= DataCount;
}

int TTreeTable :: GoNext ( void ) { // переход к следующей записи

}
```



```
// методы печати

ostream& operator<<(ostream &os,TTreeTable &tab ) {
//  cout << "Печать таблицы" << endl;
  cout << "Table printing" << endl;
  tab.PrintTreeTab(os,tab.pRoot);
  return os;
}               /*-----------------------------------------------*/
void TTreeTable :: Draw(void) { // печать дерева (рисунок слева направо)
  //  cout << "Печать таблицы" << endl;
  cout << "Table printing" << endl;
  DrawTreeTab(pRoot,0);
}               /*-----------------------------------------------*/
// запись ключей в массив в порядке возрастания с запоминаем номеров ярусов
void TTreeTable :: PutValues(PTTreeNode pNode,int Level) {
  if ( (pNode != NULL) && ( pos<20) ){
    PutValues(pNode->pLeft,Level+1);
    tk[pos] = pNode->Key;
    tl[pos] = Level;
    pos++;
    PutValues(pNode->pRight,Level+1);
  }
}
void TTreeTable :: Show(void) { // печать дерева (рисунок сверху вниз)
  int maxl=0, i, j, k, pn;
  pos = 0;
  PutValues(pRoot,0);
  for ( i=0; i<pos; i++ ) if ( maxl < tl[i] ) maxl = tl[i];
  //  cout << "Печать таблицы" << endl;
  cout << "Table visualization" << endl;
  for ( i=0; i<maxl+1; i++ ) { // номер яруса
    pn = 0;
    for ( j=0; j<pos; j++ ) { // печать ключей яруса i
      if ( tl[j] == i ) {
        for ( k=0; k<2*(j-pn); k++ ) cout << " ";
        cout << tk[j]; pn = j+1;
      }
    }
    cout << endl;
  }
}               /*-----------------------------------------------*/
// служебные методы
void TTreeTable :: PrintTreeTab(ostream &os, PTTreeNode pNode) {
  if ( pNode != NULL ) { // печать дерева с вершиной pNode
    PrintTreeTab(os,pNode->pLeft);
    pNode->Print(os); os << endl;
    PrintTreeTab(os,pNode->pRight);
  }
}               /*-----------------------------------------------*/
void TTreeTable :: DrawTreeTab(PTTreeNode pNode,int Level) {
  if ( pNode != NULL ) { // печать таблицы с соблюдением ярусов
    DrawTreeTab(pNode->pRight,Level+1);
    for ( int i=0; i<2*Level; i++ ) cout << " ";
    pNode->Print(cout); cout << endl;
    DrawTreeTab(pNode->pLeft,Level+1);
  }
}               /*-----------------------------------------------*/
void TTreeTable :: DeleteTreeTab(PTTreeNode pNode) {
  if ( pNode != NULL ) { // удаление дерева с вершиной pNode
    DeleteTreeTab(pNode->pLeft);
    DeleteTreeTab(pNode->pRight);
    delete pNode;
  }
}               /*-----------------------------------------------*/
```

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// balnode.h Copyright (c) Гергель В.П. 17.09.2000
//
// Таблицы - базовый класс объектов-значений для сбалансированных деревьев

#ifndef __BALNODE_H
#define __BALNODE_H

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

#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) {}
    virtual TDatValue * GetCopy(); // изготовить копию
    int  GetBalance(void) const { return Balance; }
    void SetBalance(int bal)    { Balance = bal;  }
  protected:
    virtual void Print(ostream &os) {
      TTreeNode::Print(os);
      os << " " << Balance;
    }
    friend class TBalanceTree;
};
typedef TBalanceNode *PTBalanceNode;
#endif
// end of balnode.h

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// balnode.cpp Copyright (c) Гергель В.П. 17.09.2000
//
// Таблицы - базовый класс объектов-значений для сбалансированных деревьев

#include "balnode.h"

TDatValue * TBalanceNode :: GetCopy() { // изготовить копию
  TBalanceNode *temp = new TBalanceNode(Key,pValue,NULL,NULL,BalOK);
  return temp;
}
// end of balnode.cpp

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// baltree.h - Copyright (c) Гергель В.П. 23.09.2000, 16.01.2003
//
// Сбалансированные деревьев поиска (AVL-деревья)

#ifndef __BALTREE_H
#define __BALTREE_H

#include "treetab.h"
#include "balnode.h"
#define HeightOK  0
#define HeightInc 1
```



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

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// baltree.cpp - Copyright (c) Гергель В.П. 23.09.2000, 16.01.2003
//
// Сбалансированные деревьев поиска (AVL-деревья)

#include <math.h>
#include <conio.h>
#include "baltree.h"


void TBalanceTree :: InsRecord (  TKey k, PTDatValue pVal  ) { // вставить
запись
  if ( IsFull() ) SetRetCode(TabFull);
  else InsBalanceTree((PTBalanceNode)pRoot,k,pVal);
}             /*--------------------------------------------*/

// вставить запись - балансировка
int TBalanceTree::InsBalanceTree(PTBalanceNode &pNode,TKey k,PTDatValue pVal) {
  int HeighIndex = HeightOK;
  if ( pNode == NULL ) { // вставка вершины
    SetRetCode(TabOK);
    pNode = new TBalanceNode(k,pVal);
    HeighIndex = HeightInc;
    DataCount++;
  }
  else if ( k < pNode->GetKey() ) {
    if ( InsBalanceTree(PTBalanceNode(pNode->pLeft),k,pVal) == HeightInc )
      // после вставки высота левого поддерева увеличилась - балансировка
      HeighIndex = LeftTreeBalancing(pNode);
  }
  else   if ( k > pNode->GetKey() ) {
    if ( InsBalanceTree(PTBalanceNode(pNode->pRight),k,pVal) == HeightInc )
      // после вставки высота правого поддерева увеличилась - балансировка
      HeighIndex = RightTreeBalancing(pNode);
  }
  else { // совпадение ключей
    SetRetCode(TabRecDbl);
    HeighIndex = HeightOK;
  }
  return HeighIndex;
}             /*--------------------------------------------*/
```

```
// балансировка после вставки в левое поддерево
int TBalanceTree :: LeftTreeBalancing(PTBalanceNode &pNode) {
  int HeighIndex = HeightOK;
  switch ( pNode->GetBalance() ) { // проверка предыдущей балансировки
    case BalRight: pNode->SetBalance(BalOK);  // в поддереве был перевес справа
         HeighIndex = HeightOK; break;        // устанавливается равновесие
    case BalOK:    pNode->SetBalance(BalLeft);// в поддереве было равновесие
         HeighIndex = HeightInc; break;       // устанавливается перевес слева
    case BalLeft: // в поддереве был перевес слева - необходима балансировка
         PTBalanceNode p1, p2;
         p1 = PTBalanceNode(pNode->pLeft);
         if ( p1->GetBalance() == BalLeft ) { // случай 1 - однократ. LL-поворот
           pNode->pLeft = p1->pRight; // 1
           p1->pRight   = pNode;      // 2
           pNode->SetBalance(BalOK);  // 3
           pNode = p1;
         }
         else { // случай 2 - двукратный LR-поворот
           p2 = PTBalanceNode(p1->pRight);
           p1->pRight   = p2->pLeft;  // 1
           p2->pLeft    = p1;         // 2
           pNode->pLeft = p2->pRight; // 3
           p2->pRight   = pNode;      // 4
           if ( p2->GetBalance()==BalLeft ) pNode->SetBalance(BalRight);
             else pNode->SetBalance(BalOK); // 5
           if ( p2->GetBalance()==BalRight) p1->SetBalance(BalLeft);
             else p1->SetBalance(BalOK);    // 6
           pNode = p2;
         }
         pNode->SetBalance(BalOK); HeighIndex = HeightOK;
  }
  return HeighIndex;
}              /*-----------------------------------------------*/
// балансировка после вставки в правое поддерево
int TBalanceTree :: RightTreeBalancing(PTBalanceNode &pNode) {
```

```
}              /*-----------------------------------------------*/
```


Нижегородский Государственный Университет им. Н.И. Лобачевского

Общий курс:

Методы программирования - 2

Тема 3.4:

Таблицы с вычислимым входом

Гергель В.П., профессор
кафедра МО ЭВМ ВМК

Содержание

Глава 3. ***Организация доступа по имени***
3.3. Таблицы с вычислимым входом
1. Общая характеристика подхода
2. Основные понятия и определения
3. Способы построения функций расстановки
4. Открытое перемешивание (вставка, поиск, удаление, итератор)
5. Реализация (схема наследования)
6. Оценка эффективности
7. Разрешение коллизий – метод цепочек

Заключение
Вопросы для обсуждения

3.4. Таблицы с вычислимым входом

1. Общая характеристика подхода

- ☑ При поиске данных в таблицах во многих случаях можно предварительно оценить место расположения искомых записей (например, слово "янтарь" располагается в конце словаря и, следовательно, словарь целесообразно раскрыть ближе к последней странице)
- ↳ Первоначально забракованный способ аналитического задания отношения "иметь имя" может быть использован для разработки способа представления таблиц (!)

3.4. Таблицы с вычислимым входом

2. Основные понятия и определения…

- ☑ Пусть для ключей записей имеется правило преобразования к значениям целого типа. Используем получаемые значения для определения места расположения записей

Пример 3.2. Пусть ключ есть строка не более чем из 6 символов. Каждый символ может быть преобразован в целое число (код символа). Тогда и ключ может рассматриваться как значение целого типа

$$A \sim 41_{16},\ B \sim 42_{16},\ \Rightarrow AB \sim 16706_{10}$$

Всего разных имен может быть $N=2^{48}$, тогда положив размер памяти для таблицы $M=2^{48}$, мы сможем организовать ***прямой доступ*** к записям

3.4. Таблицы с вычислимым входом

2. Основные понятия и определения…

- ☑ Организация прямого доступа в рассмотренном виде приводит к неэффективному использованию памяти
  - требуется большой памяти (M>>1)
  - количество используемых ключей, как правило, существенно меньше теоретически возможного набора (Л.Н. Толстой "Война и мир" содержит всего 500 имен)
- ↳ Размер памяти для представления таблицы значительно меньше возможного количества имен (M<<N) и, как результат, получаемые по ключам числовые значения необходимо преобразовывать к диапазону номеров (адресов) строк памяти

3.4. Таблицы с вычислимым входом

2. Основные понятия и определения

Определение 3.10. Функция преобразования значения ключа к номеру (адресу) строки памяти для хранения записи

$$H : K \to L \quad (L=\{0,\ldots,M-1\})$$

называется *функцией* (*хеширования*, *перемешивания*, *рассеивания*) *расстановки* (hash - мешанина, путаница)

Определение 3.11. Таблицы, представление которых организуется при использовании функции расстановки, называются *таблицы с вычислимыми адресами* (*хеш-таблицы*, *перемешиваемые таблицы*)

3. Способы построения функция расстановки…

☑ При M<N функции расстановки является *взаимно-неоднозначной* (*неинъективной*)

$$\exists\, k_1, k_2 \in K : h(k_1) = h(k_2)$$

↳ Тем самым, при использовании функции расстановки могут возникать ситуации, когда получаемый функцией номер строки памяти для расположения записи уже является использованным

Определение 3.12. Ситуация, когда для расположения записи функцией расстановки определяется уже занятая строка памяти называется *относительным переполнением* (*коллизией*)

3. Способы построения функция расстановки

Требования (редкое возникновение коллизий)

- Быстрое вычисление
- Равномерное распределение имен
- Равномерное распределение часто используемых имен
- Равномерное распределение близких имен

Примеры

- Код левого (правого) символа ключа
- Числовой код ключа
- Представление ключа в виде нескольких полей, используемых для получения целого значения
- $h(k) = \eta(k) \bmod M$ $(M \neq 2^k)$

4. Открытое перемешивание - вставка…

Вставка (начальный вариант)

1. s = h(key) // применение функции расстановки
2. ЕСЛИ строка s свободна, ТО { K[s]=key; Останов }
3. ЕСЛИ K[s] == key, ТО { Дублирование; Останов }
4. (!) Коллизия { Изменение s и переход к п. 2 }

? Как разрешать ситуации коллизии ?

↳ Возможное решение – использование строк s+1, s+2,… (такой способ может провести к появлению *сгущений* в структуре хранения таблицы)

↳ Уменьшение эффекта сгущений может быть достигнуто при применении способа *открытого* или *линейного перемешивания*

$$s' = (s+p) \bmod M \quad (1 \le p < M)$$

4. Открытое перемешивание - вставка…

Рассмотрим выполнение процедуры вставки

Пусть N=6, s=3, p=2

? Как обеспечить нахождение свободных строк ?

↳ Возможное решение состоит в выборе взаимно-простых значений для M и p

0	
1	
2	
3	
4	
5	

☑ В более общем виде правило разрешения коллизии может быть представлено как функция вторичного перемешивания

$$s' = h'(s)$$

4. Открытое перемешивание - вставка…

Теорема 3.2. Алгоритм открытого перемешивания при взаимно-простых M и p гарантирует нахождение свободных строк структуры хранения таблицы

Доказательство. Рассмотри множество G={0,1,…,M-1} с операцией a⊕b=(a+b)mod M.

Свойства операции:

– G замкнута относительно ⊕
– операция ассоциативна и коммутативна
– ∃ нулевой и обратные элементы

⇒ Множество G с операцией ⊕ является группой

Выделим подмножество G'={0, a, a ⊕ a,…}. Такое множество G' с операцией ⊕ тоже является группой (группы такого вида называются *циклическими*)

4. Открытое перемешивание – вставка…

Обозначим a⊕a⊕…⊕a через n_a (a - число повторений)

Если n>0, то минимальное значение n, при котором n_a=0, называется *порядком* элемента a и обозначается |a| (т.е. порядок определяет количество итераций открытого перемешивания, после которого начнется повторение строк)

Целое значение в операции (n_a) / M получится только при n=M (т.к. a<M и для взаимно-простых a и M). Но это означает также, что n_a=0, и, тем самым, |a|=M. Отсюда следует G=G'.

4. Открытое перемешивание - вставка…

☑ При разрешении коллизии просматриваемые строки могут рассматриваться как список, в котором порядок следования определяется при помощи алгоритмического правила. Тем самым, удаление записи в середине подобного списка не должно разрушать связность записей. Это может быть достигнуто *специальной маркировкой* строк с удаленными записями.

↳ Строка структуры хранения имеет *три возможных состояния – свободное, занятое, пустое* (пустое состояние строки возникает после удаления хранимой в строке записи)

4. Открытое перемешивание - вставка

Вставка (окончательный вариант)

1. Если n==M, ТО { Переполнение; Останов }
2. f=-1 // f – номер первой найденной пустой строки
3. s = h(key) // применение функции расстановки
4. ЕСЛИ s занята и K[s]==key, ТО {Дублир.; Останов }
5. ЕСЛИ s пустая и (f<0), ТО { f = s }
6. ЕСЛИ s свободна и (f < 0), ТО { K[s]=key; Останов }
7. ЕСЛИ s свободна и (f >-1), ТО { K[f]=key; Останов }
8. (!) Коллизия {s = (s+p) mod M и переход к п. 4 }

4. Открытое перемешивание - поиск

Поиск

1. f=-1 // f – номер первой найденной пустой строки
2. s = h(key) // применение функции расстановки
3. ЕСЛИ s занята и K[s]==key, ТО { Останов }
4. ЕСЛИ s пустая и (f<0), ТО { f = s }
5. ЕСЛИ s свободна, ТО { Останов }
6. (!) Коллизия { s = (s+p) mod M и переход к п. 3 }

4. Открытое перемешивание - удаление

Удаление

1. Поиск записи
2. ЕСЛИ запись найдена, ТО { Отметить строку как пустую }

5. Реализация – схема наследования

6. Оценка эффективности…

- $T_{min} = 1$
- $T_{max} = N$
- $T_{cp} = ?$

6. Оценка эффективности…

Пусть ключи равновероятны и функция хеширования равномерно рассеивает ключи по структуре хранения (M – количество строк памяти, n – количество записей)

Оценим среднее число сравнений при вставке $(k+1)$ ключа

$p_1 = \frac{M-n}{M}$ - вероятность попадания на пустую строку на 1 сравнении

$p_2 = \frac{n}{M}\frac{M-n}{M-1}$ - вероятность попадания на пустую строку на 2 сравнении

$p_3 = \frac{n}{M}\frac{n-1}{M-1}\frac{M-n}{M-2}$

…

$p_i = \frac{n}{M}\frac{n-1}{M-1}\frac{n-2}{M-2}\cdots\frac{n-i+2}{M-i+2}\frac{M-n}{M-i+1}$

6. Оценка эффективности…

Тогда

$$E_{n+1} = \sum_{i=1}^{n+1} i p_i = 1\cdot\frac{M-n}{M} + 2\cdot\frac{n}{M}\frac{M-n}{M-1} + \ldots + (n+1)\left(\frac{n}{M}\frac{n-1}{M-1}\frac{n-2}{M-2}\cdots\frac{1}{M-n+1}\right) = \frac{M+1}{M-n+1}$$

6. Оценка эффективности…

☑Число сравнений при вставке равно числу сравнений при поиске

Оценим среднее количество сравнений при поиске в таблице с n записями

$E = \frac{1}{n}\sum_{i=1}^{n} E_i = \frac{M+1}{n}\sum_{i=1}^{n}\frac{1}{M-i+2} = \frac{M+1}{n}(H_{M+1} + H_{M-n+1})$

где $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$ - есть гармоический ряд

$H_n \cong \gamma + \ln n$. (формула Эйлера, $\gamma \cong 0{,}577$)

Введем $\alpha = n/(M+1)$ - *индекс заполненности*. Тогда

$E = \frac{1}{\alpha}(\ln(M+1) - \ln(M-n+1)) = \frac{1}{\alpha}\ln\frac{M+1}{M-n+1} = \frac{1}{\alpha}\ln(1-\alpha)$

6. Оценка эффективности

Эффективность зависит не от количества записей, а от степени заполненности структуры хранения (!!!)

α	E	О.П.
0.1	1.05	1.06
0.25	1.15	1.17
0.5	1.39	1.50
0.75	1.85	2.50
0.9	2.56	5.50
0.95	3.15	10.5
0.95	4.66	

Для открытого перемешивания сложность поиска оценивается как

$E=(1-\alpha/2)/(1-\alpha)$

Экспериментальная оценка сложности: программа, приложение

7. Разрешение коллизий – метод цепочек…

☑ Замечания к открытому перемешиванию как способу разрешения коллизий

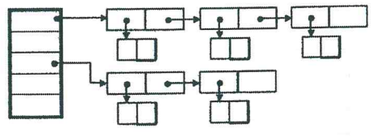
- Размер памяти для таблицы фиксирован
- Хранение записей без упорядоченности по ключам

Широко используемый подход для разрешения коллизий – *метод цепочек*, когда все записи, для которых функция хеширования определяет одно и тоже значение, представляются в виде линейного списка

☑ Открытое перемешивание еще называют *закрытым хешированием*, метод цепочек – *открытое хеширование*

7. Разрешение коллизий – метод цепочек…

Структура хранения



7. Разрешение коллизий – метод цепочек…

Реализация

7. Разрешение коллизий – метод цепочек

☒ Оценка сложности выполнения операций для таблиц с вычислимым входом при использовании метода цепочек ?

Экспериментальная оценка сложности: программа, приложение

7. Заключение

- Использование хеширование позволяет разрабатывать эффективные способы представления таблиц
- Эффективность обработки таблиц с вычислимым входом зависит не от количества записей, а от степени заполненности структуры хранения
- Метод цепочек обеспечивает получение структуры хранения таблицы с динамическим распределением памяти

Вопросы для обсуждения

- Сравнение таблиц с вычислимым входом с другими способами представления таблиц
- Способы построения функций хеширования
- Сравнение открытого перемешивания и метода цепочек для разрешения коллизий

Темы заданий для самостоятельной работы

- Уплотнение структуры хранения таблиц при открытом перемешивании
- Проведение вычислительных экспериментов

Следующая тема

- Графы и методы их обработки

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// hashtab.h - Copyright (c) Гергель В.П. 16.09.2000
//
// Таблицы - базовый класс для таблиц с вычислимым входом

#ifndef __HASHTAB_H
#define __HASHTAB_H

#include "ttable.h"

class TTabRecord;

class THashTable : public TTable {
  protected:  // хеш-функция
    virtual unsigned long HashFunc (const TKey key);
  public:
    THashTable() : TTable() {}
};
#endif
```

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// hashtab.cpp - Copyright (c) Гергель В.П. 16.09.2000
//
// Таблицы - базовый класс для таблиц с вычислимым входом

#include "hashtab.h"                    string

unsigned long THashTable :: HashFunc (const TKey key) {
  unsigned long hashval = 0;
  int Len = key.length();
  for ( int i=0; i<Len; i++ )
    hashval = ( hashval << 3 ) + key[i];
  return hashval;
}
```

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// arrhash.h - Copyright (c) Гергель В.П. 16.09.2000
//
// Таблицы с вычислимым входом - Открытое перемешивание

#ifndef __ARRHASH_H
#define __ARRHASH_H

#include "hashtab.h"

#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 своб. строка, обнаруженная при поиске
    int CurrPos;        // строка памяти при завершении поиска
    PTTabRecord  pMark; // маркер для индикации строк с удаленными записями
    // функция открытого перемешивания
    int GetNextPos ( int pos ) { return ( pos + HashStep ) % TabSize; }
```



```
  public:
    TArrayHash(int Size=TabMaxSize, int Step=TabHashStep);
   ~TArrayHash();
    // информационные методы
    virtual int IsFull() const { return DataCount >= TabSize; } // заполнена ?
    // 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 GoNext для последней записи таблицы)
    // доступ
    virtual TKey GetKey( void ) const;      // ключ текущей записи
    virtual PTDatValue GetValuePtr( void ) const; // указ-ль на значение
};
#endif
```

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// arrhash.cpp - Copyright (c) Гергель В.П. 16.09.2000, 18.01.2003
//
// Таблицы с вычислимым входом - Открытое перемешивание

#include "arrhash.h"

TArrayHash :: TArrayHash (int Size, int Step) : THashTable() {
  pRecs = new PTTabRecord[Size]; TabSize=Size; HashStep = Step;
  for ( int i=0; i<TabSize; i++ ) pRecs[i] = NULL;
  pMark = new TTabRecord(string(""),NULL);
}             /*-----------------------------------------*/

TArrayHash :: ~TArrayHash () { // деструктор
  for ( int i=0; i<TabSize; i++ )
    if ( (pRecs[i]!=NULL) && (pRecs[i]!=pMark) ) delete pRecs[i];
  delete [] pRecs;
  delete pMark;
}

// основные методы
PTDatValue TArrayHash :: FindRecord ( TKey k ) { // найти запись
 PTDatValue pValue = NULL;
 FreePos=-1; Efficiency=0;
 CurrPos = HashFunc(k) % TabSize;
 for ( int i=0; i<TabSize; i++ ) {
  Efficiency++;
  if ( pRecs[CurrPos] == NULL ) break; // свободная строка - конец поиска
  else if (pRecs[CurrPos] == pMark )    // пустая строка - запоминаем первую
          { if (FreePos==-1) FreePos=CurrPos; }
  else if (pRecs[CurrPos]->Key == k)    // нашли ключ
          { pValue = pRecs[CurrPos]->pValue; break;     }
  CurrPos = GetNextPos ( CurrPos );     // открытое перемешивание
 }
 if ( pValue == NULL ) SetRetCode(TabNoRec); else SetRetCode(TabOK);
 return pValue;
}             /*-----------------------------------------*/
```

```
void TArrayHash :: InsRecord (  TKey k, PTDatValue pVal ) { // вставить запись




}             /*-----------------------------------------------*/

void TArrayHash :: DelRecord ( TKey k ) { // удалить запись
  PTDatValue temp = FindRecord(k);           // поиск ключа




}             /*-----------------------------------------------*/

// навигация
int TArrayHash :: Reset ( void ) { // установить на первую запись




}             /*-----------------------------------------------*/

int TArrayHash :: IsTabEnded ( void ) const { // таблица завершена ?
  return CurrPos >= TabSize;
}             /*-----------------------------------------------*/

int TArrayHash :: GoNext ( void ) { // переход к следующей записи
  if ( !IsTabEnded() ) {
    while ( ++CurrPos < TabSize ) // поиск занятой строки
      if ( (pRecs[CurrPos]!=NULL) && (pRecs[CurrPos]!=pMark) ) break;
  }
  return IsTabEnded();
}             /*-----------------------------------------------*/

// доступ
TKey TArrayHash :: GetKey( void ) const { // значение ключа текущей записи
  return ( (CurrPos<0) || (CurrPos>=TabSize) ) ? string("") : pRecs[CurrPos]->Key;
}

PTDatValue TArrayHash :: GetValuePtr( void ) const { // указатель на значение
  return ( (CurrPos<0) || (CurrPos>=TabSize) ) ? NULL : pRecs[CurrPos]->pValue;
}

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// listhash.h - Copyright (c) Гергель В.П. 16.09.2000
//
// Таблицы с вычислимым входом - Метод цепочек

#ifndef __LISTHASH_H
#define __LISTHASH_H
```



```
#include "datlist.h"
#include "hashtab.h"

#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);
   ~TListHash();
    // информационные методы
    virtual int IsFull() const; // таблица заполнена ?
    // 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 GoNext для последней записи таблицы)
    // доступ
    virtual TKey GetKey( void ) const;            // ключ текущей записи
    virtual PTDatValue GetValuePtr( void ) const; // указатель на значение
};
#endif

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// listhash.cpp - Copyright (c) Гергель В.П. 16.09.2000, 19.01.2003
//
// Таблицы с вычислимым входом - Метод цепочек

#include "listhash.h"

TListHash :: TListHash (int Size) : THashTable() {
  pList = new PTDatList[Size]; TabSize=Size; CurrList=0;
  for ( int i=0; i<TabSize; i++ ) pList[i] = new TDatList;
}             /*-----------------------------------------------*/
TListHash :: ~TListHash() {
  for ( int i=0; i<TabSize; i++ ) delete pList[i];
  delete [] pList;
}             /*-----------------------------------------------*/

int TListHash :: IsFull() const { // таблица заполнена ?
  PTDatLink pLink = new TDatLink();
  int temp = (pLink == NULL);
  delete pLink;
  return temp;
}             /*-----------------------------------------------*/
PTDatValue TListHash :: FindRecord ( TKey k ) { // найти запись
 PTDatValue pValue = NULL;
 CurrList = HashFunc(k) % TabSize;   // функция расстановки
 PTDatList pL = pList[CurrList]; Efficiency=0;
 for ( pL->Reset(); !pL->IsListEnded(); pL->GoNext() ) // поиск по списку
  if (PTTabRecord(pL->GetDatValue())->Key == k)
    { pValue = PTTabRecord(pL->GetDatValue())->pValue; break; }
 Efficiency = pL->GetCurrentPos()+1; // номер тек. поз. = к-ву итераций поиска
 if ( pValue == NULL ) SetRetCode(TabNoRec); else SetRetCode(TabOK);
 return pValue;
}             /*-----------------------------------------------*/
```

```
void TListHash :: InsRecord  ( TKey k, PTDatValue pVal ) { // вставить запись




}             /*-------------------------------------------*/

void TListHash :: DelRecord ( TKey k ) { // удалить запись
  PTDatValue temp = FindRecord(k);          // поиск в таблице




}             /*-------------------------------------------*/

// навигация
int TListHash :: Reset ( void ) { // установить на первую запись
  CurrList = 0;
  pList[CurrList]->Reset();
  return IsTabEnded();
}             /*-------------------------------------------*/

int TListHash :: IsTabEnded ( void ) const { // таблица завершена ?
  return CurrList >= TabSize;
}             /*-------------------------------------------*/

int TListHash :: GoNext ( void ) { // переход к следующей записи




}             /*-------------------------------------------*/

// доступ
TKey TListHash :: GetKey( void ) const { // значение ключа текущей записи
  if ( (CurrList<0) || (CurrList>=TabSize) ) return string("");
  PTTabRecord pRec = PTTabRecord(pList[CurrList]->GetDatValue());
  return (pRec==NULL ) ? string("") : pRec->Key;
}             /*-------------------------------------------*/

PTDatValue TListHash :: GetValuePtr( void ) const { // указатель на значение
  if ( (CurrList<0) || (CurrList>=TabSize) ) return NULL;
  PTTabRecord pRec = PTTabRecord(pList[CurrList]->GetDatValue());
  return (pRec==NULL ) ? NULL : pRec->pValue;
}             /*-------------------------------------------*/
```


Нижегородский Государственный Университет им. Н.И. Лобачевского

Общий курс:

Методы программирования - 2

Тема 5:

Автоматизация управления ЭВМ и операционные системы

Гергель В.П., профессор
кафедра МО ЭВМ ВМК

Содержание

Глава 5. *Автоматизация управления ЭВМ и операционные системы*
5.1. Функции ОС по управлению ЭВМ
1. Понятие операционной системы
2. Функции ОС
5.2. Модель управления процессами и ресурсами
1. Концепция процесса
2. Модель выполняемого набора программ как системы процессов
3. *Понятие* ресурса
4. Описание состояния вычислительной системы
5. Правила изменения состояний
6. Модель вычислительной системы
7. Обнаружение и исключение тупиков
Заключение и Вопросы для обсуждения

5.1. Функции ОС по управлению ЭВМ

1. Понятие операционной системы...

Проблемы использования современных ЭВМ

- Сложность эффективного управления устройствами
  - Обеспечение независимости функционирования устройств
  - Введение устройств управления для отдельных функциональных подсистем ЭВМ
  - Организация взаимодействия на основе механизма событий и прерываний
  - Аппаратная поддержка обработки прерываний (регистр состояний , операции сохранения и восстановления состояния)
- Трудоемкость использования сложных устройств
  - Разработка программного обеспечения для снижения сложности использования устройств (драйверы)
- Неравномерность загрузки устройств при однопрограммном режиме использования ЭВМ

5.1. Функции ОС по управлению ЭВМ

1. Понятие операционной системы

Определение 5.1. Под операционной системой обычно понимается комплекс взаимосвязанных системных программ, выполняемый на компьютере для снижения сложности эффективного использования ЭВМ

5.1. Функции ОС по управлению ЭВМ

2. Функции операционной системы

- Управление работой устройств ЭВМ
- Обеспечение средств более простого использования ресурсов для пользователей (программистов)
- Распределение ресурсов ЭВМ при многопрограммном режиме работы компьютера

5.2. Модель управления процессами и ресурсами

1. Концепция процесса...

- Основой для построения моделей функционирования набора одновременно выполняемых программ является понятие *процесса*
- Представление множества одновременно выполняемых программ в виде набора процессов позволяет:
  - выполнить анализ проблем организации *взаимодействия процессов*, обычно возникающих при потреблении ресурсов,
  - определить моменты и способы обеспечения *синхронизации и взаимоисключения процессов* при использовании неразделяемых ресурсов
  - изучить условия возникновения или доказать отсутствие *тупиков* в ходе выполнения программ из-за нехватки ресурсов

1. Концепция процесса…

- Процесс может пониматься как "*некоторая последовательность команд, претендующая наравне с другими процессами программы на использование процессора для своего выполнения*".
- Конкретизация понятия процесса зависит от целей исследования параллельных программ. Для анализа проблем организации взаимодействия процессов процесс можно рассматривать как последовательность команд

$$p_n = (i_1, i_2, \cdots, i_n)$$

(для простоты изложения материала будем предполагать, что процесс описывается единственной командной последовательностью)

1. Концепция процесса…

Динамика развития процесса определяется моментами времен начала выполнения команд (*траектория процесса*)

$$t(p_n) = t_p = (\tau_1, \tau_2, ..., \tau_n)$$

где τ_j, $1 \le j \le n$, есть время начала выполнения команды τ_j.

Предположения:

- Команды процесса исполняются строго последовательно,
- Команды в ходе своей реализации не могут быть приостановлены (т.е. являются неделимыми)
- Команды имеют одинаковую длительность выполнения, равную 1

$$\forall i, \quad 1 \le i < n \quad \Rightarrow \quad \tau_{i+1} \ge \tau_i + 1$$

1. Концепция процесса

- $\tau_{i+1} = \tau_i + 1$ - процесс *активен*,
- $\tau_{i+1} \ge \tau_i + 1$ - процесс *приостановлен* (т.е. находится в состоянии *ожидания* или *блокировки*)

2. Модель выполняемого набора программ как системы процессов…

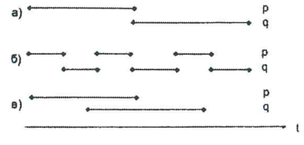
- Понятие процесса может быть использовано в качестве основного *конструктивного элемента* для построения модели выполняемого набора программ как системы *взаимодействующих процессов*.
- Подобное представление набора программ позволяет:
  - получить более компактные (поддающиеся анализу) вычислительные схемы многопрограммного режима работы ЭВМ,
  - выделить в явном виде моменты взаимодействия процессов при потреблении ресурсов ЭВМ,
  - Оценивать эффективность применяемых методов распределения ресурсов

2. Модель выполняемого набора программ как системы процессов…

Возможные типовые варианты соотношений временных траекторий на примере двух одновременно выполняемых процессов p и q состоят в следующем :

- выполнение процессов осуществляется строго последовательно, т.е. процесс q начинает свое выполнение только после полного завершения процесса p (*однопрограммный режим* работы ЭВМ),
- выполнение процессов может осуществляться одновременно, но в каждый момент времени могут исполняться команды только какого либо одного процесса (*режим разделения времени* или *многопрограммный режим* работы ЭВМ),
- *параллельное выполнение* процессов, когда одновременно могут выполняться команды нескольких процессов (данный режим исполнения процессов осуществим только при наличии в вычислительной системе нескольких процессоров).

2. Модель выполняемого набора программ как системы процессов



3. Понятие ресурса

- Понятие *ресурса* обычно используется для обозначения любых объектов вычислительной системы, которые могут быть использованы процессом для своего выполнения. В качестве ресурса может рассматриваться процесс, память, программы, данные и т.п.
- Различают следующие категории ресурсов:
  - *выделяемые* (монопольно используемые, неперераспределяемые) ресурсы;
  - *повторно распределяемые ресурсы*;
  - *разделяемые* ресурсы;
  - *многократно используемые* (реентерабельные) ресурсы.

4. Описание состояния вычислительной системы…

Состояние вычислительной системы может быть представлено в виде ориентированного графа (V,E) со следующей интерпретацией и условиями:

- Множество V разделено на два взаимно пересекающихся подмножества P и R, представляющие *процессы*

$$P=(p_1, p_2, ..., p_n)$$

и *ресурсы*

$$R=(R_1, R_2, ..., R_m)$$

ВС.

- Граф является "двудольным" по отношению к подмножествам вершин P и R, т.е. каждое ребро $e \in E$ соединяет вершину P с вершиной R. Если ребро имеет вид $e=(p_i, R_j)$, то e есть ребро *запроса* и интерпретируется как запрос от процесса p_i на единицу ресурса R_j. Если ребро e имеет вид $e=(R_j, p_i)$, то e есть ребро *назначения* и выражает назначение единицы ресурса R_j процессу p_i.

4. Описание состояния вычислительной системы…

- Для каждого ресурса $R_j \in R$ существует целое $k_j \ge 0$, обозначающее количество единиц ресурса R_j.
- Пусть $|(a,b)|$ - число ребер, направленных от вершины a к вершине b. Тогда при принятых обозначениях для ребер графа должны выполняться условия:
  - Может быть сделано не более k_j назначений (распределений) для ресурса R_j, т.е.

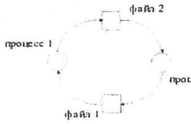
$$\sum |(R_j, p_i)| \le k_j \quad 1 \le j \le m ;$$

  - Сумма запросов и распределений относительно любого процесса для конкретного ресурса не может превышать количества доступных единиц, т.е.

$$|(R_j, p_i)| + |(p_i, R_j)| \le k_j, \quad 1 \le j \le n, \; 1 \le j \le m .$$

4. Описание состояния вычислительной системы…

Граф, построенный с соблюдением всех перечисленных правил, называется как *граф "процесс-ресурс"*.



Ресурс 1 (файл 1) выделен процессу 1, который, в свою очередь, выдал запрос на ресурс 2 (файл 2). Процесс 2 владеет ресурсом 2 и нуждается для своего продолжения в ресурсе 1.

5. Правила изменения состояний…

Состояние ВС, представленное в виде графа "процесс-ресурс", изменяется только в результате *запросов*, *освобождений* или *приобретений* ресурсов каким-либо из процессов программы.

- **Запрос.** Если программа находится в состоянии S и процесс p_i не имеет невыполненных запросов, то p_i может запросить любое число ресурсов. Тогда программа переходит в состояние T

$$S \xrightarrow{i} T$$

Состояние T отличается от S только дополнительными ребрами запроса p_i от к затребованным ресурсам.

5. Правила изменения состояний…

- **Приобретение.** Операционная система может изменить состояние программы S на состояние T в результате операции приобретения ресурсов процессом p_i, тогда и только тогда, когда p_i имеет запросы на выделение ресурсов и все такие запросы могут быть удовлетворены, т.е. если

$$\forall R_j : (p_i, R_j) \in E \quad \Rightarrow \quad (p_i, R_j) + \sum_t |(R_j, p_t)| \le k_j$$

Граф T идентичен S за исключением того, что все ребра запроса (p_i, R_j) для p_i обратны ребрам (R_j, p_i), что отражает выполненное распределение ресурсов.

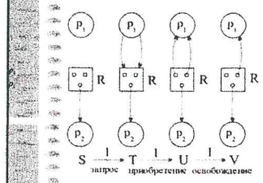
5. Правила изменения состояний…

• **Освобождение**. Процесс p_i может вызвать переход из состояния S в состояние T с помощью освобождения ресурсов тогда и только тогда, когда p_i не имеет запросов, а имеет некоторые распределенные ресурсы, т.е.

$$\forall R_j : (p_i, R_j) \notin E \quad \exists R_j : (R_j, p_i) \in E$$

В этой операции p_i может освободить любое непустое подмножество своих ресурсов. Результирующее состояние T идентично исходному состоянию S за исключением того, что в T отсутствуют некоторые ребра приобретения из S (из S удаляются ребра (R_j, p_i) каждой освобожденной единицы ресурса R_j).

5. Правила изменения состояний…



Для примера показаны состояния ВС с одним ресурсом емкости 3 и двумя процессами после выполнения операций запроса, приобретения и освобождения ресурсов для первого процесса.

5. Правила изменения состояний

При рассмотрении переходов вычислительной системы из состояния в состояние важно отметить, что *поведение процессов является недетерминированным* – при соблюдении приведенных выше ограничений выполнение любой операции любого процесса возможно в любое время

6. Модель вычислительной системы…

• Под *вычислительной системой* будем понимать модель

$$\langle \Sigma, P \rangle$$

где Σ есть множество состояний ВС $(S,T,U,\ldots)$, а P представляет множество процессов $(p_1, p_2, \ldots, p_n)$.

• *Процесс* $p_i \in R$ есть частичная функция, отображающая состояния ВС в непустые подмножества состояний

$$p_i : \Sigma \to \{\Sigma\}$$

где $\{\Sigma\}$ есть множество всех подмножеств Σ.

• Обозначим множество состояний, в которые может перейти ВС при помощи процесса p_i (*область значений процесса* p_i) при нахождении программы в состоянии S через $p_i(S)$. Возможность перехода ВС из состояния S в состояние T в результате некоторой операции над ресурсами в процессе p_i (т.е. $T \in p_i(S)$) будем пояснять при помощи записи

$$S \xrightarrow{i} T$$

6. Модель вычислительной системы

• Обобщим данное обозначение для указания достижимости состояния T из состояния S в результате выполнения некоторого произвольного количества переходов в программе

$$S \xrightarrow{*} T \quad \Leftrightarrow \quad (S = T) \vee (\exists p_i \in P : S \xrightarrow{i} T) \vee (\exists p_i, P, U \in \Sigma : S \xrightarrow{i} U, U \xrightarrow{*} T)$$

7. Обнаружение и исключение тупиков…

Проблема тупиков является значительной при организации параллельных вычислений:

• Сложность диагностирования состояния тупика (система выполняет длительные расчеты или "зависла" из-за тупика),

• Необходимость определенных специальных действий для выхода из тупика,

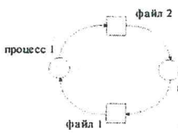
• Возможность потери данных при восстановлении системы при устранении тупика.

7. Обнаружение и исключение тупиков…

В самом общем виде *тупик (клинч, дедлок, взаимная блокировка, смертельное объятие)* может быть определен как ситуация, в которой один или несколько процессов ожидают какого-либо события, которое никогда не произойдет.

Состояние тупика может наступить не только вследствие логических ошибок, допущенных при разработке параллельных программ, но и в результате возникновения тех или иных событий в вычислительной системе (выход из строя отдельных устройств, нехватка ресурсов и т.п.).

7. Обнаружение и исключение тупиков…



Ситуация тупика: первому процессу для продолжения работы требуется файл второго процесса и одновременно второму процессу необходим файл первого процесса

7. Обнаружение и исключение тупиков…

Рассмотрим один из аспектов проблемы тупика – анализ причин возникновения тупиковых ситуаций при использовании разделяемых ресурсов. Могут быть выделены следующие **необходимые условия тупика**:

– процессы требуют предоставления им права монопольного управления ресурсами, которые им выделяются (*условие взаимоисключения*);

– процессы удерживают за собой ресурсы, уже выделенные им, ожидая в то же время выделения дополнительных ресурсов (условие *ожидания* ресурсов);

– ресурсы нельзя отобрать у процессов, удерживающих их, пока эти ресурсы не будут использованы для завершения работы (условие *неперераспределяемости*);

- существует кольцевая цепь процессов, в которой каждый процесс удерживает за собой один или более ресурсов, требующихся следующему процессу цепи (условие *кругового ожидания*).

7. Обнаружение и исключение тупиков…

С учетом построенной модели и введенных обозначений можно выделить ряд ситуаций, возникающих при выполнении программы и представляющих интерес при рассмотрении проблемы тупика:

• *процесс* p_i *заблокирован* в состоянии S, если программа не может изменить свое состояние при помощи этого процесса, т.е. если $p_i(S)=\varnothing$;

• *процесс* p_i находится в *тупике* в состоянии S, если этот процесс является заблокированным в любом состоянии T, достижимом из состояния , т.е.

$$\forall T : S \xrightarrow{*} T \Rightarrow p_i(T) =$$

7. Обнаружение и исключение тупиков…

Как результат, для обеспечения отсутствия тупиков необходимо исключить возникновение, по крайней мере, одного из рассмотренных условий.

Модель программы в виде графа "процесс-ресурс" может быть использована для обнаруживания ситуации кругового ожидания.

7. Обнаружение и исключение тупиков…

• *состояние* S называется *тупиковым*, если существует процесс p_i, находящийся в тупике в этом состоянии;

• *состояние* S есть *безопасное состояние*, если любое состояние T, достижимое из S, не является тупиковым.

Пример: состояния U и V являются безопасными, состояния S и T и W не являются безопасными, а состояние W есть состояние тупика.

7. Обнаружение и исключение тупиков…

• Рассмотренная модель программы может быть использована для определения возможных состояний программы, обнаружения и недопущения тупиков.

• В качестве возможных теоретических результатов такого анализа может быть приведена теорема [Шоу].

Теорема 5.1. Граф "процесс-ресурс" для состояния программы с ресурсами единичной емкости указывает на состояние тупика тогда и только тогда, когда он содержит цикл.

7. Обнаружение и исключение тупиков…

Дополнительная информация по данному разделу курса может быть получена, например, в:

Шоу А. Логическое проектирование операционных систем. - М.: Мир, 1981.

Заключение

• Операционная система обеспечивает возможность более простого использования ЭВМ (автоматизация управления устройствами, обеспечение удобного виртуального уровня работы с аппаратурой компьютера, эффективное распределение ресурсов)

• Моделирование работы вычислительной системы как комплекса процессов и ресурсов

• Обнаружение и исключение тупиковых ситуаций при функционировании вычислительной системы

Вопросы для обсуждения

• Возможные способы реализации модели вычислительной системы "процесс-ресурс"

• Обнаружение тупиков при множественной емкости ресурсов

• Способы устранения тупиков

Темы заданий для самостоятельной работы

• Построить множество состояния для системы из двух процессов и двух ресурсов единичной емкости

• Рассмотреть возможность построения модели системы с потребляемыми ресурсами (возможность увеличения емкости ресурса, использования ресурса без освобождения)

Следующая тема

• Представление графовых моделей на ЭВМ

Нижегородский Государственный Университет им. Н.И. Лобачевского

Общий курс:

Методы программирования - 2

Тема 5:

Автоматизация управления ЭВМ и ОС.
5.3. Представление графовых моделей на ЭВМ

Гергель В.П., профессор
кафедра МО ЭВМ ВМК

Содержание

Глава 5. ***Автоматизация управления ЭВМ и операционные системы***
5.3. Представление графовых моделей на ЭВМ
1. Понятие графа
2. Выбор структуры хранения графа
3. Реализация структуры хранения графа
4. Алгоритмы обхода (поиск в глубину и в ширину)
5. Поиск кратчайшего пути
Заключение и Вопросы для обсуждения

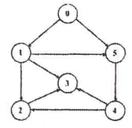
1. Понятие графа…

Определение 5.2. *Ориентированный граф* ***G*** представляет собой набор

$$G = (V, R)$$

где

- $V = \{ v_1, v_2, \ldots, v_n \}$ есть множество *вершин* графа,
- $R = \{ (i,j): i,j \in V \}$ есть множество *дуг* (*ребер*) графа



1. Понятие графа

Понятия теории графов

- Вершина-родитель (предок) и вершина-потомок
- Входящие/исходящие дуги
- Путь, длина пути, размер (глубина) графа

Дополнительные понятия

- Размеченный граф (вершины имеют метки-значения)
- Взвешенный граф (дугам графа приписывается вес)
- Длина пути с учетом весов дуг
- Кратчайший путь

2. Выбор структуры хранения графа

- Использование матрицы смежности (для неориентированных графов можно применить верхние треугольные матрицы)
- Представление наборов исходящих дуг в виде множеств смежных вершин
- Использование списков исходящих дуг для вершин графа

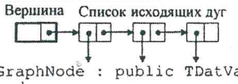
Оценка походов:

- объем используемый памяти
- эффективность выполнения основных операций обработки (вставка/удаление вершин/вершин, поиск вершины/дуги, реализация обходов)

Используем для учебного изучения *структуру хранения на основе списков исходящих дуг*

3. Реализация структуры хранения графа…

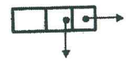
Структура хранения вершины




```
class TGraphNode : public TDatValue {
 protected:
  string   Name;  // имя вершины графа
  int      Value; // метка (значение)
  int      Index; // индекс (номер)
  TDatList Links; // список исходящих дуг
 public:
  // вставка/исключение дуг
  void InsLink(TGraphNode *pGn, int val);
  void DelLink(string NodeName);
};
```

3. Реализация структуры хранения графа...

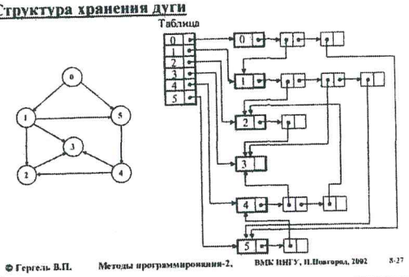
Структура хранения дуги



```
class TGraphLink : public TDatValue {
 protected:
  int Value; // вес дуги
  // начальная и конечная вершины
  PTGraphNode pFirst, pLast;
};
```


3. Реализация структуры хранения графа...

Структура хранения дуги



3. Реализация структуры хранения графа

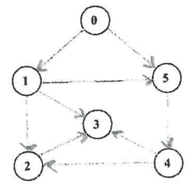
Структура хранения графа

```
class TGraph : public TDatValue {
 protected:
  int NodeNum;            // количество вершин
  int LinkNum;            // количество дуг
  TTreeTable Nodes;       // множество вершин графа
 public:
  TGraph ();
  int GetNodeNum (void) { return NodeNum; }
  int GetLinkNum (void) { return LinkNum; }
  // вставка вершин и дуг графа
  void InsNode ( string nm, int val=0 );
  void InsLink ( string fn, string ln, int val=1);
};
```


4. Алгоритмы обхода...

Поиск в глубину (depth-first search)

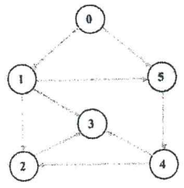
Обработка очередной вершины графа продолжается рекурсивной процедурой обработки смежных вершин



4. Алгоритмы обхода...

Поиск в ширину (breadts-first search)

Для каждой вершины сначала выполняется обработка непосредственно смежных вершин



4. Алгоритмы обхода...

- Для исключения циклов и запоминания набора уже обработанных вершин можно использовать множество TSet
- Для запоминания набора достигнутых, но еще не обработанных вершин, можно использовать структуры:
  - Стек – для алгоритма поиска в глубину
  - Очередь – для алгоритма поиска в ширину

4. Алгоритмы обхода (итератор)...

```
TSearchMode Mode;       // способ обхода
PTGraphNode pCurrNode; // текущая вершина
// достигнутые, но не обработанные вершины
TDataRoot *pStream;
// множество вершин, достигнутых в ходе обхода
TBitField *pFound;
```


4. Алгоритмы обхода (итератор)...

Инициализация (Reset)

- Инициализация структур
- Вставка первой вершины в поток pStream и ее отметка в множестве pFound
- Выполнение метода GoNext

4. Алгоритмы обхода (итератор)...

Проверка завершения (IsGraphEnded)

```
return pCurrNode==NULL; // текущее звено ?
```


4. Алгоритмы обхода (итератор)

Переход к следующей вершине графа (GoNext)

- Получить вершину из потока pStream
- Поместить смежные вершины, если они еще не достигнуты, в поток pStream
- Отметить смежные вершины в множестве pFound

Пример: программа, приложение

5. Поиск кратчайшего пути...

Алгоритм Дейкстры (Dijkstra)

- В ходе работы алгоритма определяются расстояния кратчайших путей от заданной вершины до всех достижимых вершин графа
- Для запоминания длин найденных путей используется массив pDist (если до вершины нет найденного пути, соответствующее значение массива устанавливается в ∞)
- Вершины, уже вошедшие в построенные кратчайшие пути, фиксируются при помощи множества pFound

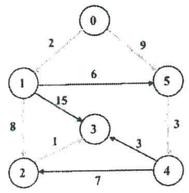
5. Поиск кратчайшего пути...

Алгоритм Дейкстры (Dijkstra)

- На каждой итерации алгоритма строится кратчайший путь до вершины, до которой этот путь еще не был определен (т.е. до вершины, не принадлежащей множеству pFound); данная вершина определяется минимальным расстоянием в массиве pDist среди вершин, до которых еще не найден кратчайший путь
- Найденная вершина фиксируется в множестве pFound; далее из этой вершина строятся пути до вершин, не вошедших в pFound и, если этот длина этих путей меньше, чем значения в массиве pDist, значения длин путей в pDist корректируется

5. Поиск кратчайшего пути…

Алгоритм Дейкстры (Dijkstra)



0	1	2	3	4	5
0	2	10	11	11	8

5. Поиск кратчайшего пути…

Алгоритм Дейкстры (Dijkstra)

Для восстановления найденных кратчайших путей для каждой вершины определим массив pPred, в котором будем запоминать для вершин номера предшествующих по кратчайшему пути вершин

Пример: программа, приложение

5. Поиск кратчайшего пути…

Обоснование алгоритма Дейкстры

Теорема 5.2. Пусть $G=(V,E)$ – взвешенный ориентированный граф с неотрицательной весовой функцией $w:E\to R$ и исходной вершиной s. Тогда после применения алгоритма Дейкстры

$$\forall v\in V \Rightarrow d[v] = \delta_{min}(s,v)$$

Доказательство.

Покажем, что на любой итерации цикла

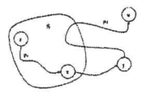
1. Для вершин $v\in S$ (S – множество уже обработанных вершин) уже выполняется утверждение теоремы, т.е. $d[v] = \delta_{min}(s,v)$, и, кроме того, к этим вершинам существуют кратчайшие пути только из вершин S.
2. Для вершин $v\in Q=V\setminus S$ значение $d[v]$ равно наименьшему весу пути из s в v, если учитывать только те *особые пути*, в которых все вершины (кроме последней) лежат в S (если таких путей нет, то $d[v] = \infty$)

5. Поиск кратчайшего пути…

Обоснование алгоритма Дейкстры

Для начальной итерации, когда в S только одна вершина s, свойства 1 и 2 справедливы. Покажем выполнимость этих утверждений для любой итерации.

Пусть u есть вершина из $Q=V\setminus S$ с минимальным значением d[u]. Покажем, что особый путь из s в u веса d[u] является кратчайшим для u.



Пусть существует другой путь из s в u. Рассмотрим первую вершину $y\in S$. По свойству 1 и предположения индукции вес пути из s в y не меньше d[y] и, кроме того, $d[y]\geq d[u]$ (условие выбора u). Это означает, что вершину u можно добавить в S и свойство 1 не нарушится.

5. Поиск кратчайшего пути…

Обоснование алгоритма Дейкстры

Расширение множества $S'=S\cup\{u\}$ сопровождается уточнением расстояний до вершин множества Q

$$\forall v\in Q \Rightarrow d'[v] = \min(d[v], d[u]+w(u,v))$$

Оба значения, из которых определяется величина d'[v], есть веса особых путей из s в v. Покажем, что вес любого особого пути из s в u не меньше d'[v]. На самом деле, пусть $x\in S'$ есть вершина, предшествующая v. Тогда либо $x\in S$ и тогда вес пути не меньше d'[v] по свойству 1, либо x=u и необходимое условие следует из правила вычисления d'[v].

Итак, условия 1 и 2 выполняются на всех итерациях цикла, в том числе, и по завершении работы алгоритма Дейкстры.

5. Поиск кратчайшего пути…

Оценка сложности алгоритма Дейкстры

- Количество операций пересчета весов путей до вершин графа пропорционально числу вершин N
- Количество итераций алгоритма также совпадает с N

$$\Rightarrow T_{max} = O(N^2)$$

Если для представления вектора весов d использовать приоритетную очередь на основе двоичной кучи, то можно получить оценку

$$\Rightarrow T_{max} = O(M \log N),$$

где M есть общее количество вершин в графе.

Заключение

- Графы является одной из самых важных моделей описания сложных структур
- Для представления графов на ЭВМ можно использовать матрицы смежности или списковые структуры хранения
- Для обхода графов основными алгоритмами являются поиски в глубину или в ширину
- Поиск кратчайших путей в графе может быть выполнен при помощи алгоритма Дейкстры

Вопросы для обсуждения

- Сравнение способов представления графов на ЭВМ
- Оценка способов обхода графов при решении задач
- Сложность задачи поиска кратчайшего пути между двумя заданными вершинами графа

Темы заданий для самостоятельной работы

- Расширить класс реализации графов TGraph методами удаления вершин/дуг
- Расширить набор методов обработки графов (поиск циклов, построение минимального остовного дерева, определение связности)
- Разработать структуру хранения для неориентированных графов

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// graphnode.h Copyright (c) Гергель В.П. 09.02.2003
//
// Графы - класс для представления вершин графа

#ifndef __GRAPHNODE_H
#define __GRAPHNODE_H

#include <iostream.h>
#include "datvalue.h"
#include "datlist.h"

class TGraphLink;   // класс для представления дуги графа

class TGraphNode : public TDatValue {
  protected:
    string   Name;  // имя вершины графа
    int      Value; // метка (значение) вершины
    int      Index; // индекс (номер) вершины
    TDatList Links; // список исходящих дуг
  public:
    TGraphNode ( string nm="", int val=0 ) { Name=nm; Value=val; Index=0; }
    void   SetName  (string nm="") { Name=nm;  }
    string GetName  (void)      { return Name;  }
    void   SetValue (int val=0){ Value = val;  }
    int    GetValue (void)      { return Value; }
    void   SetIndex (int ind=0){ Index = ind;  }
    int    GetIndex (void)      { return Index; }
    // вставить дугу веса val к вершине pGn
    void   InsLink  (TGraphNode *pGn, int val=1);
    void   DelLink  (string nm );    // удалить дугу к вершине с именем nm
    virtual  TDatValue * GetCopy();  // изготовить копию
  protected:
    virtual void Print(ostream &os); // печать вершины
  friend class TGraph;
};
#endif
typedef TGraphNode * PTGraphNode;
// end of tgraphnode.h

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
// graphnode.cpp Copyright (c) Гергель В.П. 09.02.2003
//
// Графы - класс для представления вершин графа

#include "graphnode.h"
#include "graphlink.h"

// вставить дугу веса val к вершине pGn
void TGraphNode :: InsLink (PTGraphNode pGn, int val) {
  PTGraphLink pLink = new TGraphLink(val,this,pGn);
  Links.InsFirst(pLink);
}
// удалить дугу к вершине с именем nm
void TGraphNode :: DelLink  (string nm ) {
  PTGraphLink pLink;
  for ( Links.Reset(); !Links.IsListEnded(); Links.GoNext() ) {
    pLink = (PTGraphLink) Links.GetDatValue();
    if ( pLink->GetLastNode()->GetName() == nm ) {
      Links.DelCurrent(); break;
    }
  }
}
```



```
TDatValue * TGraphNode :: GetCopy() { // изготовить копию
  PTGraphNode pNode = new TGraphNode(Name,Value);
  for ( Links.Reset(); !Links.IsListEnded(); Links.GoNext() ) {
    pNode->Links.InsFirst(Links.GetDatValue());
  }
  return pNode;
}

void TGraphNode :: Print(ostream &os) { // печать вершины
  PTGraphLink pLink = NULL;
  os << "Vertex = " << Name << ", Label = " << Value << ", Index = " << Index;
  os << "; Edges = ";
  for ( Links.Reset(); !Links.IsListEnded(); Links.GoNext() ) {
    if ( pLink != NULL ) os << ", ";
    pLink = (PTGraphLink) Links.GetDatValue();
    os << pLink->GetLastNode()->GetName();
  }
}
// end of graphnode.cpp

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// graphlink.h Copyright (c) Гергель В.П. 09.02.2003
//
// Графы - класс для представления дуг графа

#ifndef __GRAPHLINK_H
#define __GRAPHLINK_H

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

class TGraphLink : public TDatValue {
  protected:
    int Value; // вес дуги
    PTGraphNode pFirst, pLast; // указатели на начальную и конечную вершины
  public:
    TGraphLink ( int val=1, PTGraphNode pF=NULL, PTGraphNode pL=NULL ) {
      Value=val; pFirst=pF; pLast=pL;
    }
    void        SetValue (int val=0) { Value = val;  }
    int         GetValue (void)      { return Value; }
    void        SetFirstNode (PTGraphNode pF=NULL) { pFirst=pF; }
    PTGraphNode GetFirstNode (void)              { return pFirst; }
    void        SetLastNode  (PTGraphNode pL=NULL) { pLast=pL;  }
    PTGraphNode GetLastNode  (void)              { return pLast;  }
    virtual  TDatValue * GetCopy() {  // изготовить копию
      return new TGraphLink(Value,pFirst,pLast);
    }
  protected:
    virtual void Print(ostream &os) { // печать дуги
      os << "Edge: Weight = " << Value;
      os << "; Vertexes: out = " << pFirst->GetName();
      os << ", in = " << pLast->GetName();
    }
};
#endif
typedef TGraphLink * PTGraphLink;
// end of tgraphlink.h
```

```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// graphpath.h - Copyright (c) Гергель В.П. 12.02.2003
//
// Класс для описания пути в графе

#ifndef __GRAPHPATH_H
#define __GRAPHPATH_H

#include "datlist.h"
#include "graphnode.h"

class TGraphPath : protected TDatList {
  protected:
    int Distance; // длина пути
  public:
    TGraphPath() { Distance=0; }
    // доступ
    int  GetDistance ( void )  { return Distance; }          // длина пути
    void SetDistance ( int d ) { Distance = d;    }
    PTGraphNode GetNode ( TLinkPos mode = CURRENT ) const { // значение
      return (PTGraphNode)TDatList::GetDatValue(mode);
    }
    virtual int IsEmpty()  const { return pFirst==pStop; }   // путь пуст ?
    int GetPathLen()       const { return TDatList::GetListLength(); } // к-во вершин
    // навигация
    TDatList::SetCurrentPos;          // установить текущую вершину
    TDatList::GetCurrentPos;          // получить номер текущей вершины
    TDatList::Reset;                  // установить на начало пути
    int IsPathEnded ( void ) const {  // путь завершен ?
      return TDatList::IsListEnded();
    }
    TDatList::GoNext;                 // сдвиг вправо от текущей вершины
                // (=1 после применения GoNext для последней вершины пути)
    // вставка вершины
    void InsNode(PTGraphNode pNode) { // вставить вершину в путь
      TDatList::InsLast(pNode);
    }
    void DelPath ( void ) {           // удалить путь
      TDatList::DelList(); Distance=0;
    }
    friend ostream& operator<<(ostream &os,TGraphPath &Path) {
      os << "Path printing" << endl;
      os << "  Distance =        " << Path.Distance << endl;
      os << "  Number of edges = " << Path.GetPathLen() << endl;
      for ( Path.Reset(); !Path.IsPathEnded(); Path.GoNext() )
        os << *Path.GetNode() << endl;
      return os;
    }

};
typedef TGraphPath *PTGraphPath;
#endif
```



```
// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// graph.h Copyright (c) Гергель В.П. 09.02.2003
//
// Графы

#ifndef __GRAPH_H
#define __GRAPH_H

#include <iostream.h>
#include "dataroot.h"
#include "bitfield.h"
#include "graphlink.h"
#include "graphpath.h"
#include "treetab.h"

enum TSearchMode { BREADTH, DEPTH };

class TGraph : public TDatValue {
  protected:
    int NodeNum;            // количество вершин
    int LinkNum;            // количество дуг
    TSearchMode Mode;       // способ обхода
    TTreeTable Nodes;       // множество вершин графа (стр-ра хранения - таблица)
    TDataRoot *pStream;     // поток вершин для итератора
    TBitField *pFound;      // множество достигнутых вершин для итератора
    PTGraphNode pCurrNode; // указатель текущей вершины при обходе
  public:
    TGraph () {
      NodeNum = LinkNum = 0; Mode=BREADTH;
      pStream=NULL; pFound=NULL; pCurrNode=NULL;
    }
    ~TGraph () { delete pStream; delete pFound; }
    int GetNodeNum (void) { return NodeNum; }
    int GetLinkNum (void) { return LinkNum; }
    void SetSearchMode(TSearchMode md=BREADTH) { Mode=md; }
    TSearchMode GetSearchMode(void)         { return Mode; }
    void InsNode ( string nm, int val=0 );    // вставка вершины
    void InsLink ( string fn, string ln, int val=1 ); // вставка дуги
    virtual  TDatValue * GetCopy();           // создание копии
    void NodeNumeration();                    // нумерация вершин
    // итератор
    int Reset       (void);         // установить на первую вершину
    int IsGraphEnded(void) const;   // обход завершен ?
    int GoNext      (void);         // переход к следующей вершины
    PTGraphNode GetCurrNode(void) { return pCurrNode; } // доступ
    PTGraphPath GetShortestPath(string fn, string ln);  // поиск кратчайшего пути
  protected:
    virtual void Print(ostream &os);          // печать графа
};
#endif
typedef TGraph * PTGraph;
// end of tgraph.h

// ННГУ, ВМК, Курс "Методы программирования-2", C++, ООП
//
// graph.cpp Copyright (c) Гергель В.П. 09.02.2003
//
// Графы

#include <limits.h>
#include "datstack.h"
#include "datqueue.h"
#include "graph.h"
```

```
void TGraph :: InsNode ( string nm, int val ) { // вставка вершины
  PTDatValue pN = Nodes.FindRecord(nm);
  if ( pN == NULL ) Nodes.InsRecord(nm,new TGraphNode(nm,val));
  NodeNum++;
}
void TGraph :: InsLink ( string fn, string ln, int val ) { // вставка дуги



}
TDatValue * TGraph :: GetCopy() {  // создание копию
  TGraph *pGr = new TGraph();
  for ( Nodes.Reset(); !Nodes.IsTabEnded(); Nodes.GoNext() ) {
    PTGraphNode pN1 = (PTGraphNode)Nodes.GetValuePtr();
    pGr->Nodes.InsRecord(pN1->GetName(),pN1);
  }
  pGr->NodeNum = NodeNum;
  pGr->LinkNum = LinkNum;
  return pGr;
}
void TGraph :: NodeNumeration() { // нумерация вершин
  int i;
  for ( Nodes.Reset(), i=0; !Nodes.IsTabEnded(); Nodes.GoNext(), i++ ) {
    PTGraphNode pN = (PTGraphNode)Nodes.GetValuePtr();
    pN->SetIndex(i);
  }
}
// итератор
int TGraph :: Reset(void) { // установить на первую вершину
  if ( pStream != NULL ) delete pStream;
  if ( pFound  != NULL ) delete pFound;
  if ( Mode == BREADTH ) pStream = new TQueue(NodeNum);
  else pStream = new TStack(NodeNum);
  pFound = new TBitField(NodeNum);
  NodeNumeration(); Nodes.Reset();
  PTGraphNode pN = (PTGraphNode)Nodes.GetValuePtr();
  pStream->Put(pN); pFound->SetBit(pN->GetIndex());
  return GoNext();
}
int TGraph :: IsGraphEnded(void) const { // обход завершен ?
  return pCurrNode==NULL;
}
int TGraph :: GoNext(void) { // переход к следующей вершины
  if ( pStream->IsEmpty() ) pCurrNode=NULL;
  else {
    pCurrNode = (PTGraphNode)pStream->Get();
    pCurrNode->Links.Reset();
    while ( !pCurrNode->Links.IsListEnded() ) { // цикл по дугам вершины









    }
  }
  return pCurrNode==NULL;
}
void TGraph :: Print(ostream &os) { // печать графа
  os << "Graph printing" << endl;
  for ( Nodes.Reset(); !Nodes.IsTabEnded(); Nodes.GoNext() )
    os << *Nodes.GetValuePtr() << endl;
}
```



```
// поиск кратчайшего пути
PTGraphPath TGraph :: GetShortestPath(string fn, string ln) {
  PTGraphPath pPath = NULL;
  PTGraphNode pF = (PTGraphNode)Nodes.FindRecord(fn);  // начальная вершина
  PTGraphNode pL = (PTGraphNode)Nodes.FindRecord(ln);  // конечная вершина
  if ( (pF != NULL) && (pL != NULL) ) {
    NodeNumeration();
    int Findex = pF->GetIndex();
    int Lindex = pL->GetIndex();
    // подготовка вектора указателей на вершины
    int i, d, dmin, kmin, ind, id;
    PTGraphNode *pGraphNodes = new PTGraphNode[NodeNum];
    for ( Nodes.Reset(), i=0; !Nodes.IsTabEnded(); Nodes.GoNext(), i++ )
      pGraphNodes[i] = (PTGraphNode)Nodes.GetValuePtr();
    //  подготовка данных для алгоритма Дейкстры
    PTGraphLink pLink;
    PTGraphNode pN, pNd;
    if ( pFound  != NULL ) delete pFound;
    pFound = new TBitField(NodeNum); // множество обработанных вершин
    int *pDist = new int[NodeNum];   // вектор расстояний путей
    int *pPred = new int[NodeNum];   // предыдущая вершина на кратчайшем пути
    for ( i=0; i<NodeNum; i++ ) pDist[i] = INT_MAX;
    pDist[Findex] =  0; // начальная вершина
    pPred[Findex] = -1;
    // цикл алгоритма Дейкстры
    for ( i=0; i<NodeNum; i++ ) {
      // определение следующей вершины с кратчайшим путем
      dmin = INT_MAX;
      for ( int k=0; k<NodeNum; k++ )
        if ( !pFound->GetBit(k) && (pDist[k]<dmin) ) { dmin=pDist[k]; kmin=k; }
      if ( dmin==INT_MAX ) break; // остались только изолированные вершины
      // включение найденной вершины в множество обработанных
      PTGraphNode pN = pGraphNodes[kmin];
      pFound->SetBit(kmin);
      if ( kmin==Lindex ) break;
      // пересчет расстояний до еще не обработанных вершин с учетом вершины pN
      for ( pN->Links.Reset(); !pN->Links.IsListEnded(); pN->Links.GoNext() ) {
        pLink = (PTGraphLink) pN->Links.GetDatValue(); // очередная дуга
        d     = pLink->GetValue();     // вес дуги
        pNd   = pLink->GetLastNode(); // вершина, в которую входит дуга
        ind   = pNd->GetIndex();       // индекс вершины
        if ( pDist[kmin]+d < pDist[ind] ) {  // новый более короткий путь
          pDist[ind] = pDist[kmin]+d; pPred[ind] = kmin;
        }
      }
    }
    // запоминание длины кратчйшего пути



    do { // заполнение пути
      pPath->InsNode(pGraphNodes[ind]); ind = pPred[ind];
    } while ( ind >= 0 );
    // освобождение памяти
    delete pGraphNodes; delete pDist; delete pPred;
    delete pFound; pFound=NULL;
  }
  return pPath;
}
```

Вопросы для контроля по общему курсу "ЭВМ и программирование"

Введение

1. Проблема доказательства правильности программ
2. Способы снижения сложности программного обеспечения

***Тема* 1. Структуры действия и структуры данных**

1. Рекурсивное описание вычислительного процесса и структуры данных.
2. Структуры данных и математические структуры.
3. Переменные структуры и схемы структуры.
4. Понятие экземпляра, схемы структуры.
5. Линейные структуры данных
6. Структура машинной памяти. Вектор памяти как образ линейной структуры.
7. Практическая работа 1: Структура хранения множеств
8. Практическая работа 2: Структуры хранения для матриц специального вида
9. Динамические структуры.
10. Практическая работа 3: Структуры хранения динамических структур типа стек
11. Практическая работа 4: Структуры хранения динамических структур типа очередь
12. Сравнение структур хранения линейных и динамических структур.
13. Статическое и динамическое распределение памяти.
14. Управление памятью путем перепаковки структур хранения.
15. Практическая работа 5: Структура хранения нескольких стеков в общей памяти.
16. Роль гипотез о росте структур при разработке систем управления памятью путем перепаковки.
17. Оценка параметров модели в ходе выполнения программ (адаптация).
18. Линейный список.
19. Способы реализации списков на языках высокого уровня.
20. Управление свободной памятью при использовании сцепления.
21. Практическая работа 6: Реализация структуры хранения нескольких стеков с использованием списков на языке высокого уровня.
22. Сравнение непрерывной и списковой структур хранения.
23. Динамическое распределение памяти в языке C/C++ (выделение и освобождение памяти).
24. Реализация стека с использованием динамически распределяемой памяти.
25. Пример использования стеков: поразрядная сортировка.
26. Пример использования стеков: преобразование арифметических выражений в польскую форму записи.
27. Практическая работа 7: Разработка общего представления линейного списка для обеспечения списковой структуры хранения.
28. Общая характеристика стандартной библиотеки шаблонов.

***Тема* 2. Динамические структуры и конструирование математических моделей.**

1. Система для арифметических действий над полиномами (представление полиномов, управление памятью, выполнение операций).
2. Представление многочленов от нескольких переменных. Исключение хранения мономов с нулевыми коэффициентами.
3. Схема наследования программ для обеспечения структуры хранения полиномов.
4. Реализация программ для обеспечения работы с линейным циклическим списком.
5. Структура класса для представления на ЭВМ полиномов от нескольких переменных.
6. Алгоритм сложения многочленов от нескольких переменных.
7. Представление текста связным списком.
8. Операторы объединения списков и расчленения списка.
9. Алгоритм обхода иерархического списка.
10. Копирование списка.

11. Сборка мусора.
12. Плексы как представление рисунков, состоящих из точек и соединяющих их отрезков.
13. Алгоритм обхода плекса.
14. Алгоритм вставки линии.
15. Плекс, как представление арифметического выражения.

***Тема* 3. Организация доступа по имени.**

1. Организация доступа по имени. Таблицы. Поиск по ключу (просмотр и двоичный поиск).
2. Упорядоченные таблицы. Алгоритм сортировки включением.
3. Упорядоченные таблицы. Алгоритм сортировки слиянием.
4. Упорядоченные таблицы. Алгоритм быстрой сортировки.
5. Представление таблиц с использованием деревьев поиска.
6. Деревья поиска. Алгоритмы обхода.
7. Деревья поиска. Алгоритмы поиска и вставки.
8. Деревья поиска. Алгоритм удаления.
9. Сбалансированные и идеально сбалансированные деревья поиска. Алгоритм балансировки при вставке.
10. Таблицы с вычислимым входом. Запись и поиск при переполнении (способ открытого перемешивания).

***Тема* 4. Проблемное языковое обеспечение.**

1. Определение формального языка.
2. БНФ-форма задания грамматики формального языка.
3. Представление грамматик с помощью синтаксических диаграмм. Порождение языковых цепочек в результате обхода диаграмм.
4. Контекстно-свободные грамматики (терминалы и нетерминалы, правила вывода).
5. Распознавание операторов формального языка.
6. Пример описания грамматики языка арифметических выражений.

***Тема* 5. Автоматизация управления ЭВМ и операционные системы.**

1. Понятие процесса и ресурса в операционной системе.
2. Понятия состояния операционной системы. Граф "процесс-ресурс".
3. Модель управления процессами и ресурсами в операционной системе в форме асинхронного конечного автомата.
4. Условие наличия тупика в системе с ресурсами единичной емкости
5. Способы представления графов
6. Реализация структуры хранения графов
7. Алгоритм обхода графов. Поиск в глубину
8. Алгоритм обхода графов. Поиск в ширину
9. Задача поиска кратчайших путей. Алгоритм Дейкстры