Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Оренбургский государственный университет»

Кафедра управления и информатики в технических системах

И.И. Стрекалова

СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Рекомендовано Редакционно-издательским советом федерального государственного бюджетного образовательного учреждения высшего профессионального образования «Оренбургский государственный университет» в качестве методических указаний для студентов, обучающихся по программам высшего профессионального образования по направлению подготовки 230100.62 Информатика и вычислительная техника

Оренбург
2012

УДК 004.421(076.5)
ББК 32.973-018я7
С 84

Рецензент – доцент, кандидат педагогических наук А.Е. Шухман

Стрекалова, И.И.
С 84 Структуры и алгоритмы обработки данных: методические указания / И.И. Стрекалова; Оренбургский гос. ун-т. – Оренбург: ОГУ, 2012. – 107 с.

Методические указания содержат 13 лабораторных работ, примерные темы курсовых работ, рекомендуемую литературу по структурам и алгоритмам обработки данных и являются основным учебным руководством при выполнении лабораторных работ студентами.

Методические указания предназначены для выполнения лабораторных работ по дисциплине «Структуры и алгоритмы обработки данных» студентами по направлению подготовки 230100.62 Информатика и вычислительная техника по профилю «Автоматизированные системы обработки информации и управления».

УДК 004.421(076.5)
ББК 32.973-018я7

Содержание

Введение

Дисциплина «Структуры и алгоритмы обработки данных» относится к вариативной части профессионального цикла по направлению подготовки 230100.62 Информатика и вычислительная техника. Она базируется на следующих дисциплинах: «Программирование», «Информатика», «Дискретная математика», «Математическая логика и теория алгоритмов».

Цель данной дисциплины – изучить понятия и классификации алгоритмов обработки данных, трудоемкости алгоритмов и методов ее оценки, научиться выработке критериев и оценке трудоемкости алгоритмов с учетом критериев, на примере реализаций и задач на языке Turbo Pascal.

Понятие «алгоритм обработки данных» в компьютерных науках используется для описания метода решения задачи, который в дальнейшем, возможно, реализовать в выбранной среде программирования. Тщательная разработка алгоритма является весьма эффективной частью процесса решения задачи в любой области применения. При разработке алгоритма для реальной задачи значительные усилия должны быть потрачены на осознание степени ее сложности, выяснение ограничений на входные данные, разбиение задачи на менее трудоемкие подзадачи.

Алгоритм не должен быть привязан к конкретной реализации. В силу разнообразия используемых средств программирования, их требований к аппаратным ресурсам и платформенной зависимости сходные по структуре, но различные в реализации, алгоритмы могут выдавать отличающиеся по эффективности результаты. При этом некоторые среды программирования содержат встроенные библиотечные функции, реализующие базовые алгоритмы обработки данных (например, в MS Visual Studio 2010 в библиотеки C++ входит функция быстрой сортировки массивов данных). Чтобы решения были переносимыми и оставались актуальными, не рекомендуется их ориентировать на процедурную реализацию среды. Поэтому главным в рассматриваемом подходе является выбор метода решения с учетом специфики задачи. Адаптация к среде осуществляется позднее.

Выбор того или иного метода обработки данных определяется не только сложностью задачи. Учитывать необходимо и массовость применения разработанного кода: при однократном или редком обращении к реализации предпочтительнее бывают простые алгоритмы, которые несложны в разработке. При этом, однако, допускается увеличение времени работы программы.

Массовое использование алгоритмов обработки данных требует поиска наилучшего алгоритма решения. Такой процесс бывает весьма сложен, так как требует выработки критериев оценки и применения математических методов для получения количественных характеристик. Направление компьютерных наук, занимающееся изучением оценки эффективности алгоритмов, называется анализом алгоритмов.

Курс дисциплины рассчитан на 54 часа лабораторных занятий. Формой итогового контроля являются экзамен, который проводится в третьем семестре и курсовая работа, которая дифференцированно оценивается в четвертом семестре.

В данном издании содержатся 13 лабораторных работ, примерные темы курсовых работ и рекомендуемая литература для самостоятельного изучения некоторых разделов дисциплины «Структуры и алгоритмы обработки данных». В начале каждой лабораторной работы даны краткие теоретические сведения, необходимые для решения всех последующих примеров и задач. Приведенные примеры типовых практических задач даются с подробными объяснениями. Также к каждой лабораторной работе представлены содержание отчета и примерный перечень вопросов, которые необходимы для защиты. Под символом «*» представлены дополнительные задания повышенной сложности.

Все примеры в данных методических указаниях приводятся на языке Turbo Pascal.

1 Организационно-методические указания

1. Перед началом лабораторной работы проводится консультация по методике выполнения лабораторных работ по данной дисциплине.

2. Объем каждой лабораторной работы, подготовка и порядок выполнения построены таким образом, чтобы все студенты выполнили работу и сдали отчеты.

3. Студенты готовятся к выполнению очередной работы заблаговременно.

4. Студенты обязаны изучить технику безопасности при работе на лабораторных установках до 1000 В.

5. Готовясь к лабораторному занятию, студент обязан изучить необходимый теоретический материал, пользуясь настоящими указаниями и рекомендованной литературой, произвести необходимые расчеты, заполнить соответствующую часть отчета и дать ответы на контрольные вопросы.

6. Неподготовленные студенты к выполнению лабораторной работы не допускаются.

7. Студенты, не сдавшие отчет во время занятия, сдают его в назначенное преподавателем время.

8. Студент, не выполнивший лабораторную работу, выполняет ее в согласованное с преподавателем время.

9. Каждая лабораторная работа выполняется студентами самостоятельно. Все студенты предъявляют индивидуальные отчеты. Допускается предъявление отчета в виде электронного документа.

10. Проверка знаний студентов производится преподавателем во время лабораторного занятия и при сдаче отчета.

11. При сдаче отчета студент должен показать знание теоретического материала в объеме, определяемом контрольными вопросами, а также пониманием физической сущности выполняемой работы.

2 Лабораторная работа №1. Работа с динамической памятью. Линейные однонаправленные списки. Основные операции с линейными однонаправленными списками. Использование основных операций со списком для решения прикладных задач

Лабораторная работа №1 предназначена для изучения структур данных «линейные однонаправленные списки», а также основных операций над ними.

Разработать программы, в соответствии с заданиями в пункте 2.2 и представить отчет в соответствии с приложением В.

Последовательность выполнения лабораторной работы №1 представлена в приложении Б.

Заканчивается работа защитой с оценкой «зачтено» или «незачтено» в виде экспресс-опроса в рамках теоретического и практического материала, отнесенного к пунктам 2.1 и 2.2. Примерный перечень вопросов приведен в пункте 2.3.

На выполнение и защиту лабораторной работы отводится 4 академических часа.

2.1 Краткие теоретические сведения

2.1.1 Линейные однонаправленные списки

Список – это набор связанных однотипных элементов, в котором каждый элемент каким-то образом определяет следующий за ним элемент. Различают линейные и нелинейные (циклические) списки. Линейные списки различают на:

– линейные однонаправленные (в этом списке любой элемент имеет один указатель на следующий элемент в списке или является пустым указателем у последнего элемента);

– линейные двунаправленные (в этом линейном списке любой элемент имеет два указателя, один из которых указывает на следующий элемент в списке или является пустым указателем у последнего элемента, а второй – на предыдущий элемент в списке или является пустым указателем у первого элемента;

Элемент списка представлен записью, содержащей поле с данными Data и указатель Next на следующую запись. Тип данных указателя описывается в том же описании типов до описания записи. Указатель у последнего элемента списка считается равным nil. Для работы со списком используется указатель на его первый элемент.

```
type
   PList = ^List;
   List = record
          Data : Datatype;
          Next : PList;
          end;
Var L:PList;  {указатель на первый элемент}
    T:Plist  {указатель на текущий элемент}
```

2.1.2 Основные операции со списком

Пусть в списке уже есть числа 1,2,3,4. Наглядно список изображен на рисунке 2.1.

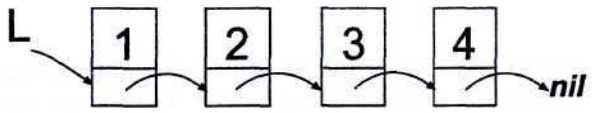


Рисунок 2.1 – Линейный однонаправленный список

Добавление в начало списка.

Начинается создание списка с пустого указателя. Далее элементы добавляются в начало списка, а именно в результате добавления числа 0 должен получиться следующий список, представленный на рисунке 2.2.

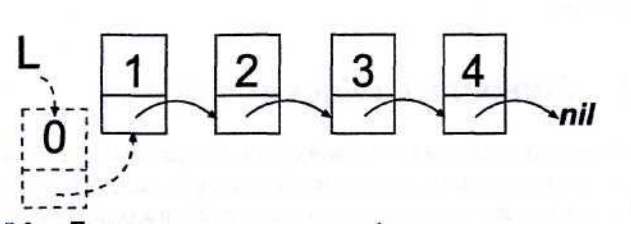
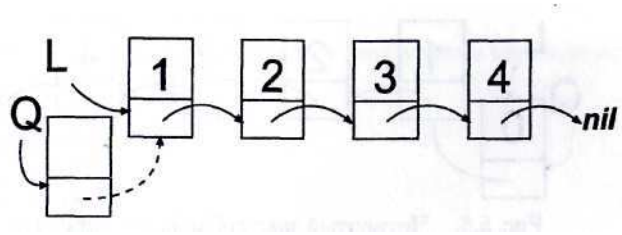


Рисунок 2.2 – Линейный список после добавления нового элемента в начало списка

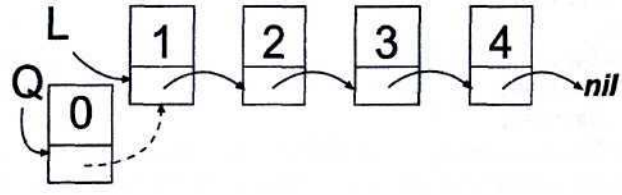
На первом шаге, показанный на рисунке 2.3, необходимо создать новый элемент. Пусть на него будет указывать временный указатель Q.




```
New(Q)
```

Рисунок 2.3 – Первый шаг добавления элемента в линейный список

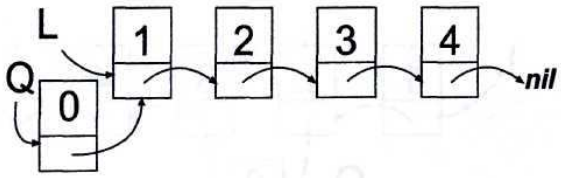
Далее заполняем информационной поле значением 0, как показано на рисунке 2.4.




```
Q^.data:=0
```

Рисунок 2.4 – Второй шаг добавления элемента в линейный список

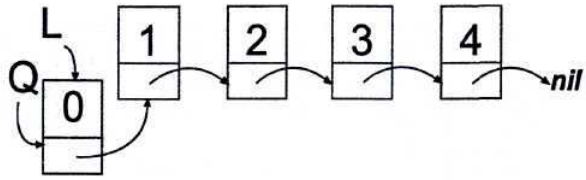
На третьем шаге формируем ссылку у нового элемента, как показано на рисунке 2.5.




```
Q^.next:=L
```

Рисунок 2.5 – Третий шаг добавления элемента в линейный список

И в последнюю очередь, в указатель L записываем ссылку на новый элемент (значение указателя Q), как показано на рисунке 2.6.




```
L:=Q
```

Рисунок 2.6 – Последний шаг добавления элемента в линейный список

Иногда требуется добавить элемент в нужное место, например, после элемента, на который указывает T. Окончательный список изображен на рисунке 2.7.

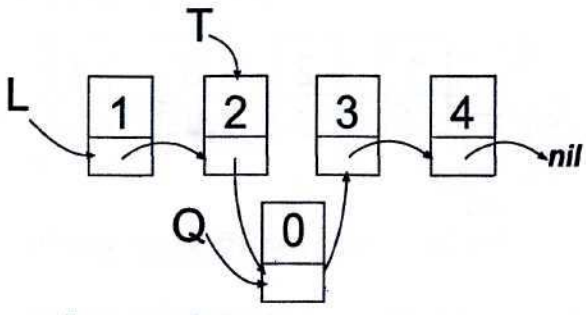


Рисунок 2.7 – Добавление нового элемента после определенного элемента

```
Q^.data:=0;
Q^.next:=T^.next;
T^.next:=Q;
T:=Q
```

Просмотр элементов.

Просмотр элементов списка осуществляется последовательно, начиная с первого элемента (головы). Для перемещения вдоль списка используется вспомогательный указатель, который будет последовательно указывать на элементы списка, пока не дойдет до его конца.

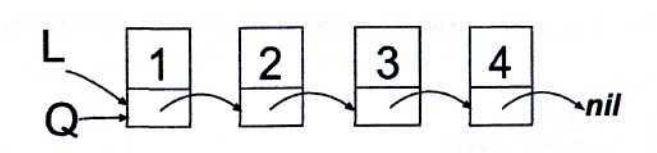
Поиск элемента с заданным ключом.

Операция поиска элемента в списке заключается в последовательном просмотре всех элементов списка до тех пор, пока текущий элемент не будет содержать заданное значение, или пока не будет достигнут конец списка. В последнем случае фиксируется отсутствие искомого элемента в списке (функция принимает значение false).

Удаление элемента.

Для удаления элемента из головы списка нужно выполнить операции, обратные операциям при добавлении в список. Вспомогательный указатель Q нужен здесь для того, чтобы удалить запись из памяти.

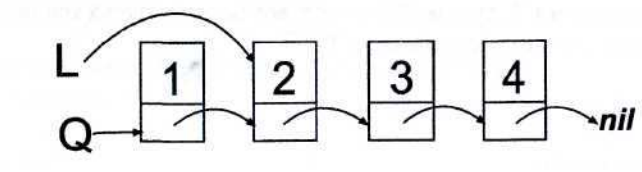
Сначала устанавливаем значение указателя Q, как показано на рисунке 2.8.




```
Q:=L
```

Рисунок 2.8 – Первый шаг при удалении первого элемента из линейного списка

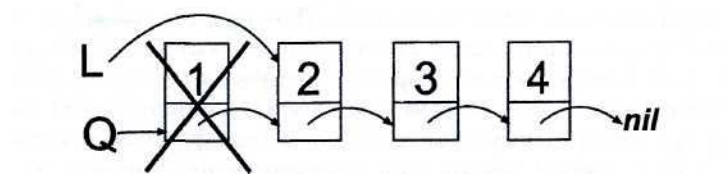
Затем устанавливаем указатель списка на второй элемент в списке, как показано на рисунке 2.9.




```
L:=L^.next
```

Рисунок 2.9 – Второй шаг при удалении первого элемента из линейного списка

Теперь осталось только удалить первый элемент из памяти, как показано на рисунке 2.10.




```
Dispose (Q)
```

Рисунок 2.10 – Заключительный этап при удалении первого элемента из линейного списка

Для удаления элемента из списка с заданным ключом N необходимо получить указатель на этот элемент X и указатель на предыдущий элемент P, который должен быть связан с элементом, следующим за указателем, как показано на рисунке 2.11.

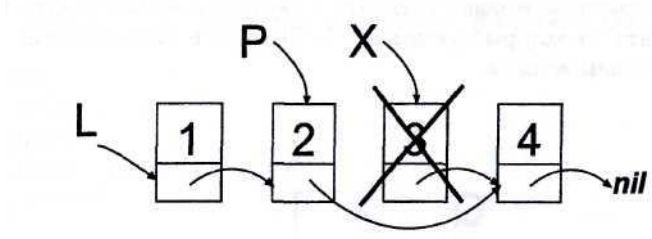


Рисунок 2.11 – Удаление элемента из линейного списка с заданным ключом

```
P^.next:=X^.next;
Temp:=X^.data;
```

```
Dispose (X)
```

2.2 Задания на лабораторную работу

1) Реализовать программу, выполняющую стандартный набор операций с линейным однонаправленным списком:

- вставка элемента в линейный однонаправленный список (в начало, середину, конец);
- просмотр линейного однонаправленного списка;
- поиск элемента в линейном однонаправленном списке;
- удаление элемента из линейного однонаправленного списка (из начала, середины, конца).

Требования:

- все действия должны быть оформлены как процедуры или функции;
- добавлению/удалению должна предшествовать проверка возможности выполнения этих операций;
- главная программа реализует следующий набор действий:
  а) инициализация пустого линейного однонаправленного списка;
  б) организация диалогового цикла с пользователем;

2) * вычислите среднее арифметическое элементов непустого списка;

3) * требуется просмотреть список и удалить элементы, у которых информационные поля равны 4.

2.3 Примерный перечень вопросов

1) Что такое динамическая структура данных?

2) Что такое список?

3) Какие виды списков существуют?

4) Какие основные операции выполняются над списком?

5) Особенности выполнения операций вставки первого и не первого элемента.

6) Особенности выполнения операций удаления первого и не первого элемента.

3 Лабораторная работа №2. Работа с динамической памятью. Линейные двунаправленные списки. Основные операции с линейными двунаправленными списками. Использование основных операций со списком для решения прикладных задач

Лабораторная работа №2 предназначена для изучения структур данных «линейные двунаправленные списки», а также основных операций над ними.

Разработать программы, в соответствии с заданиями в пункте 3.2 и представить отчет в соответствии с приложением В.

Последовательность выполнения лабораторной работы №2 представлена в приложении Б.

Заканчивается работа защитой с оценкой «зачтено» или «незачтено» в виде экспресс-опроса в рамках теоретического и практического материала, отнесенного к пунктам 3.1 и 3.2. Примерный перечень вопросов приведен в пункте 3.3.

На выполнение и защиту лабораторной работы отводится 4 академических часа.

3.1 Краткие теоретические сведения

3.1.1 Линейные двунаправленные списки

В этом линейном списке любой элемент имеет два указателя, один из которых указывает на следующий элемент в списке или является пустым указателем у последнего элемента, а второй – на предыдущий элемент в списке или является пустым указателем у первого элемента, как показано на рисунке 3.1.

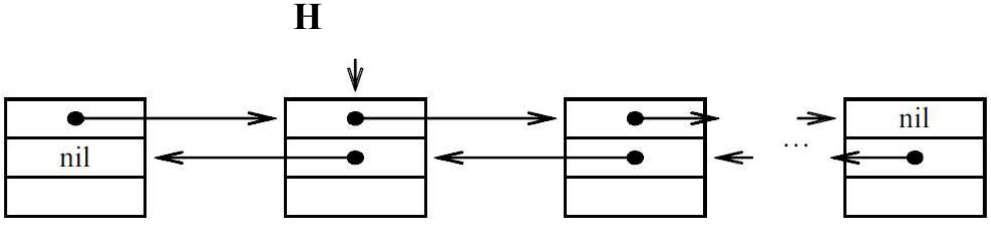


Рисунок 3.1 – Линейный двунаправленный список

Следует обратить внимание на то, что в отличие от однонаправленного списка здесь нет необходимости обеспечивать позиционирование какого-либо указателя именно на первый элемент списка, так как благодаря двум указателям в элементах можно получить доступ к любому элементу списка из любого другого элемента, осуществляя переходы в прямом или обратном направлении. Однако часто бывает полезно иметь указатель на заголовок списка.

Для описания алгоритмов этих основных операций используем следующие объявления:

```
type
    PList = ^List;  {указатель на тип элемента}
    List = record   {тип элемента}
           Data : Datatype;  {поле данных элемента}
           Next,             {поле указателя на следующий элемент}
           Last : PList;  {поле указателя на предыдущий элемент}
           end;
Var Head:PList;  {указатель на голову списка}
    Current:Plist  {указатель на текущий элемент}
```

3.1.2 Основные операции со списком

Основные операции, осуществляемые с линейным двунаправленным списком, те же, что и с однонаправленным линейным списком:

– вставка элемента в линейный двунаправленный список (в начало, середину, конец);

– просмотр линейного двунаправленного списка;

– поиск элемента в линейном двунаправленном списке;

– удаление элемента из линейного двунаправленного списка (из начала, середины, конца).

3.2 Задания на лабораторную работу

Сформировать линейный двунаправленный список (ЛДС) с заданным указателем Head, работающий с типом данных Integer. Составить программу, которая:

– обеспечивает ввод данных типа Integer с клавиатуры;

– создает линейный двунаправленный список из введенных данных с клавиатуры;

– обеспечивает диалог посредством вывода информационных сообщений и вариантов выполнения дальнейших действий;

– в данной программе будут реализованы следующие возможности работы с ЛДС:

0 - Выход из программы

1 - Создание ЛДС

2 - Добавление элемента в начало списка

3 - *Добавление элемента в середину списка перед указанным значением

4 - *Добавление элемента в середину списка после указанного значения

5 - Добавление элемента в конец списка

6 - Удаление элемента в начале списка

7 - *Удаление элемента ЛДС, стоящего перед указанным значением списка

8 - *Удаление элемента ЛДС, стоящего после указанного значения списка

9 - *Удаление определенного элемента в списке

10 - Удаление элемента в конце списка

11 - *Очистка ЛДС

12 - Поиск элемента по его значению

13 - *Подсчет количества идентичных по содержанию элементов с указанным значением.

3.3 Примерный перечень вопросов

1) Что такое динамическая структура данных?

2) Что такое список?

3) Какие виды списков существуют?

4) Какие основные операции выполняются над списком?

5) Особенности выполнения операции вставки первого и не первого элемента в двунаправленный список.

6) Особенности выполнения операции удаления первого и не первого элемента.

4 Лабораторная работа №3. Работа с динамической памятью. Циклические однонаправленные списки. Основные операции с циклическими однонаправленными списками. Использование основных операций со списком для решения прикладных задач

Лабораторная работа №3 предназначена для изучения структур данных «циклические однонаправленные списки», а также основных операций над ними.

Разработать программы, в соответствии с заданиями в пункте 4.2 и представить отчет в соответствии с приложением В.

Последовательность выполнения лабораторной работы №3 представлена в приложении Б.

Заканчивается работа защитой с оценкой «зачтено» или «незачтено» в виде экспресс-опроса в рамках теоретического и практического материала, отнесенного к пунктам 4.1 и 4.2. Примерный перечень вопросов приведен в пункте 4.3.

На выполнение и защиту лабораторной работы отводится 4 академических часа.

4.1 Краткие теоретические сведения

4.1.1 Циклические однонаправленные списки

Циклические списки – это такие списки, в которых нельзя выделить крайние элементы. Они также, как линейные, различаются на однонаправленные и двунаправленные:

- циклический однонаправленный (похож на линейный однонаправленный список, но его последний элемент содержит указатель, связывающий его с первым элементом);
- циклический двунаправленный (в этом списке любой элемент имеет два указателя, один из которых указывает на следующий элемент в списке, а второй указывает на предыдущий).

Циклический однонаправленный список похож на линейный однонаправленный список, но его последний элемент содержит указатель, связывающий его с первым элементом, как показано на рисунке 4.1.

Для полного обхода такого списка достаточно иметь указатель на произвольный элемент, а не на первый, как в линейном однонаправленном списке. Понятие «первого» элемента здесь достаточно условно и не всегда требуется. Хотя иногда бывает полезно выделить некоторый элемент как «первый», путем установки на него специального указателя. Это требуется, например, для предотвращения «зацикливания» при просмотре списка.

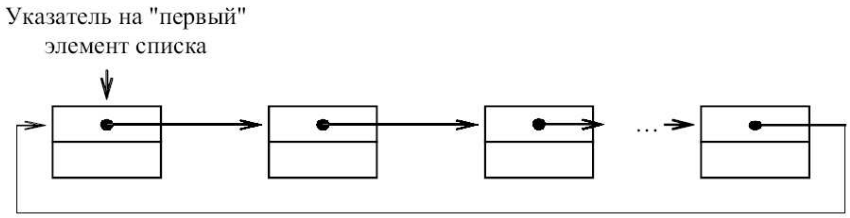


Рисунок 4.1 – Циклический однонаправленный список

Основные операции, осуществляемые с циклическим однонаправленным списком:

– вставка элемента в циклический однонаправленный список (в «начало», середину, «конец»);

– просмотр циклического однонаправленного списка;

– поиск элемента в циклическом однонаправленном списке;

– удаление элемента из циклического однонаправленного списка (из «начала», середины, «конца»).

Для описания алгоритмов этих основных операций используем те же объявления данных, что и для линейного однонаправленного списка:

```
type
    PList = ^List;  {указатель на тип элемента}
```

```
List = record   {тип элемента списка}
       Data : Datatype;  {поле данных списка}
       Next : PList;     {поле указателя на следующий элемент}
       end;
```

А также для работы с циклическим однонаправленным списком нужно объявить две глобальные переменные, которые будут являться указателями на голову и текущий элемент списка.

```
Var Head,     {указатель на первый элемент  списка}
    Current:PList; {указатель на текущий элемент списка}
```

4.1.2 Основные операции

Добавление элемента.

Вставка элемента, в отличие от линейного однонаправленного списка, реализуется с помощью одной процедуры. В качестве входных параметров передаются данные для заполнения создаваемого элемента, указатель на начало списка и указатель на текущий элемент в списке, после которого осуществляется вставка. Выходными параметрами процедур является указатель на начало списка (который возможно изменится) и указатель текущего элемента, который показывает на вновь созданный элемент.

Алгоритм добавления элемента:

1) выделяем в динамической памяти место для нового элемента;
2) заполняем его информационное поле;
3) проверяем список на пустоту:
   - если список пустой, то новый элемент образует цикл из одного элемента и становится первым в списке;
   - иначе вставляем элемент после текущего и передвигаем текущий указатель на новый элемент.

Порядок следования операторов присваивания процедуры очень важен. При неправильном переопределении указателей возможен разрыв списка или возможна потеря указателя на первый элемент, что приводит к потере доступа к части или всему списку.

Просмотр списка.

Операция просмотра списка заключается в последовательном просмотре всех элементов списка. В отличие от линейного однонаправленного списка здесь признаком окончания просмотра списка будет возврат к элементу, выделенным как «первый».

Поиск элемента.

Операция поиска элемента в списке заключается в последовательном просмотре всех элементов списка до тех пор, пока текущий элемент не будет содержать заданное значение или пока не достигнут «первый» элемент списка. В последнем случае фиксируется отсутствие искомого элемента в списке (функция принимает значение false). Входными параметрами являются значение, которое должен содержать искомый элемент и указатель на список. В качестве выходного параметра передается указатель, который устанавливается на найденный элемент или остается без изменений, если элемента в списке нет.

Удаление элемента.

Операция удаления элемента из циклического однонаправленного списка осуществляет удаление элемента, на который установлен указатель текущего элемента. После удаления указатель текущего элемента устанавливается на следующий за удаляемым элементом списка. Здесь не требуется отдельных алгоритмов удаления для крайних элементов списка, как это было в линейных списках, но в случае удаления «первого» элемента необходимо соответствующий указатель переместить на следующий элемент.

Алгоритм удаления элемента:

1) если входной параметр корректен, то

– проверяем, если удаляемый элемент не единственный в списке, то устанавливаем вспомогательный указатель на элемент, предшествующий удаляемому и удаляем указанный элемент;

– если удаляемый элемент первый, то указатель на голову списка переносим на следующий элемент и освобождаем память, которую занимал текущий элемент;

– иначе обнуляем указатель на голову списка и освобождаем память, которую занимал первый элемент.

4.2 Задания на лабораторную работу

Сформировать циклический однонаправленный список (ЦОС) с заданным указателем Head, работающий с типом данных Integer.

Составить программу, которая:

– обеспечивает ввод данных типа Integer с клавиатуры;

– создает циклический однонаправленный список из введенных данных с клавиатуры;

– обеспечивает диалог посредством вывода информационных сообщений и вариантов выполнения дальнейших действий;

– в данной программе будут реализованы следующие возможности работы с ЦОС:

0 - Выход из программы

1 - Создание ЦОС

2 - Добавление элемента в «начало» списка

3 - *Добавление элемента в середину списка перед указанным значением

4 - *Добавление элемента в середину списка после указанного значения

5 - Добавление элемента в «конец» списка

6 - Удаление элемента в «начале» списка

7 - *Удаление элемента списка, стоящего перед указанным значением списка

8 - *Удаление элемента списка, стоящего после указанного значения списка

9 - *Удаление определенного элемента в списке

10 - Удаление элемента в «конце» списка

11 - *Очистка списка

12 - Поиск элемента по его значению

13 - *Удаление элементов со значением кратным 3 из списка

14 - *Подсчет количества идентичных по содержанию элементов с указанным значением.

4.3 Примерный перечень вопросов

1) Что такое динамическая структура данных?
2) Что такое список?
3) Какие виды списков существуют?
4) Дать определение циклического списка.
5) Классификация циклических списков
6) Какие основные операции выполняются над циклическим списком?

5 Лабораторная работа №4. Стек. Использование основных операций со стеком для решения прикладных задач

Лабораторная работа №4 предназначена для изучения структур данных «стек», а также основных операций над ними.

Разработать программы, в соответствии с заданиями в пункте 5.2 и представить отчет в соответствии с приложением В.

Последовательность выполнения лабораторной работы №4 представлена в приложении Б.

Заканчивается работа защитой с оценкой «зачтено» или «незачтено» в виде экспресс-опроса в рамках теоретического и практического материала, отнесенного к пунктам 5.1 и 5.2. Примерный перечень вопросов приведен в пункте 5.3.

На выполнение и защиту лабораторной работы отводится 4 академических часа.

5.1 Краткие теоретические сведения

5.1.1 Стек. Способы реализации

Стек — это специальный тип списка, в котором все операции вставки и удаления выполняются только на одном конце, называемом вершиной (top). Стеки также иногда называют «магазинами», а в англоязычной литературе для обозначения стеков еще используется аббревиатура LIFO (last-in-first-out — последний вошел — первый вышел). Интуитивными моделями стека могут служить колода карт на столе при игре в покер, книги, сложенные в стопку, или стопка тарелок на полке буфета; во всех этих моделях взять можно только верхний предмет, а добавить новый объект можно, только положив его на верхний объект.

Стек можно реализовывать как статическую структуру данных в виде одномерного массива, а можно как динамическую структуру – в виде линейного списка. На рисунке 5.1 представлена динамическая и статическая реализация стека.

При реализации стека в виде статического массива необходимо резервировать массив, длина которого равна максимально возможной глубине стека, что приводит

к неэффективному использованию памяти. Однако работать с такой реализацией проще и быстрее.

При такой реализации дно стека будет располагаться в первом элементе массива, а рост стека будет осуществляться в сторону увеличения индексов. Одновременно необходимо отдельно хранить значение индекса элемента массива, являющегося вершиной стека.

Можно обойтись без отдельного хранения индекса, если в качестве вершины стека всегда использовать первый элемент массива, но в этом случае, при записи или чтении из стека, необходимо будет осуществлять сдвиг всех остальных элементов, что приводит к дополнительным затратам вычислительных ресурсов.

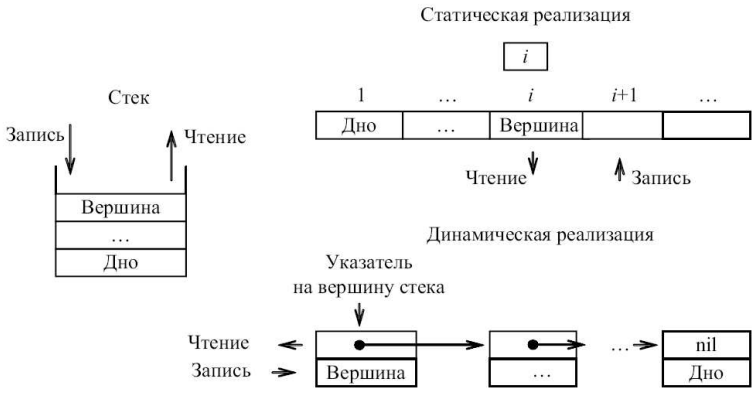


Рисунок 5.1 – Стек и его организация

Стек как динамическую структуру данных легко организовать на основе линейного списка. Поскольку работа всегда идет с заголовком стека, т. е. не требуется осуществлять просмотр элементов, удаление и вставку элементов в середину или конец списка, то достаточно использовать экономичный по памяти линейный однонаправленный список. Для такого списка достаточно хранить указатель вершины стека, который указывает на первый элемент списка.

Для описания динамического стека используем описание, аналогичное описанию линейного однонаправленного списка, где TypeData является типом элементов стека:

```
type
PElement = ^TypeElement;  {указатель на тип элемента}
TypeElement = record      {тип элемента списка}
        Data: TypeData;   {поле данных элемента}
        Next: PElement;   {поле указателя на следующий элемент}
        end;
Var
        top: PElement;  {указатель на первый элемент списка}
```

Если стек пустой, то `top` равно `nil`.

5.1.2 Основные операции со стеком

Запись в стек. Процедура записи элемента в стек должна содержать два параметра: первый (параметр – переменная) определяет указатель на начало стека, второй (параметр – значение) – записываемое значение.

Примечание – Запись в стек производится аналогично вставке нового элемента в начало списка.

Извлечение элемента из стека. В результате выполнения этой операции некоторой переменной x должно быть присвоено значение первого элемента стека и изменено значение указателя на начало списка. Процедура извлечения элемента из стека должна содержать два параметра: первый (параметр-переменная) определяет указатель на начало стека, второй (параметр – переменная) – извлекаемое значение.

Проверка на пустоту стека. Функция типа Boolean, имеющая один параметр – значение, который является указателем на начало стека, возвращает true, если параметр равно nil, иначе возвращает false.

Существуют вспомогательные операции, выполняемые над стеком: прочитать вершину стека, очистить стек, инициализация стека.

5.2 Задания на лабораторную работу

1) Используя операции со стеком, написать программу, проверяющую своевременность закрытия скобок «(,), [,] ,{, }» в строке символов (строка состоит из одних скобок этих типов).

Для решения задачи нужно определить стек следующим образом:

```
type Stek = ^PStek;
        PStek = record
        data: char;
        next:Stek;
        end;
```

В процессе решения анализируются символы строки a: String. Если встречена одна из открывающихся скобок, то она записывается в стек. При обнаружении закрывающейся скобки, соответствующей скобке, находящейся в вершине стека, последняя удаляется. При несоответствии скобки выдается сообщение об ошибке, которое фиксируется в логической переменной.

2) Написать программу вычисления значения выражения, представленного в обратной польской записи (в постфиксной записи). Выражение состоит из цифр от 1 до 9 и знаков операции.

Обычная запись:	Обратная польская запись:
*(b+c) *d*	*b c + d **
*a+(b+c) *d*	*a b c + d * +*

Просматривая строку, анализируем очередной символ, если это:

– цифра, то записываем ее в стек;

– знак, то читаем два элемента из стека, выполняем математическую операцию, определяемую этим знаком, и заносим результат в стек.

После просмотра всей строки в стеке должен оставаться один элемент, он и является решением задачи.

Примечание – процедура Val (s,x,k) преобразует символьное представление цифры s в соответствующее числовое значение. При этом значение k=0, если такое преобразование возможно, а в противном случае k>0.

5.3 Примерный перечень вопросов

1) Что такое динамическая структура?
2) Что такое стек?
3) Особенности выполнения операций со стеком.
4) Основные операции со стеком.

6 Лабораторная работа №5. Хеширование данных. Выбор хеш-функции. Разрешение коллизий. Анализ сложности хеширования с разными типами адресации

Лабораторная работа №5 предназначена для изучения построения функции хеширования и алгоритмов хеширования данных, а также для приобретения навыков разработки и применения алгоритмов открытого и закрытого хеширования при решении задач на языке Turbo Pascal.

Разработать программы, в соответствии с заданиями в пункте 6.2 и представить отчет в соответствии с приложением В.

Последовательность выполнения лабораторной работы №5 представлена в приложении Б.

Заканчивается работа защитой с оценкой «зачтено» или «незачтено» в виде экспресс-опроса в рамках теоретического и практического материала, отнесенного к пунктам 6.1 и 6.2. Примерный перечень вопросов приведен в пункте 6.3.

На выполнение и защиту лабораторной работы отводится 4 академических часа.

6.1 Краткие теоретические сведения

6.1.1 Хеширование данных

Для ускорения доступа к данным в таблицах можно использовать предварительное упорядочивание таблицы в соответствии со значениями ключей.

Для сокращения времени доступа к данным в таблицах используется так называемое случайное упорядочивание или хеширование. При этом данные организуются в виде таблицы при помощи хеш-функции h, используемой для вычисления адреса по значению ключа.

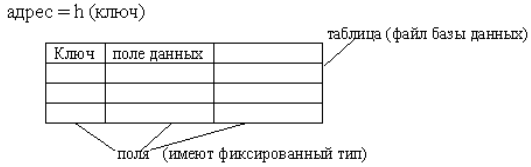


Рисунок 6.1 – Организация хеш-таблицы

Идеальной хеш-функцией является такая hash-функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса.

$$k1 \neq k2 \Rightarrow h(k1) \neq h(k2)$$

Подобрать такую функцию можно в случае, если все возможные значения ключей заранее известны. Такая организация данных носит название совершенное хеширование. В случае заранее неопределенного множества значений ключей и ограниченной длины таблицы, подбор совершенной функции затруднителен. Поэтому часто используют хеш-функции, которые не гарантируют выполнение условия.

При заполнении таблицы возникают ситуации, когда для двух неодинаковых ключей функция вычисляет один и тот же адрес. Данный случай носит название коллизия, а такие ключи называются ключи-синонимы.

6.1.2 Методы разрешения коллизий

Для разрешения коллизий используются различные методы, которые в основном сводятся к методам цепочек (открытое хеширование) и открытой адресации (внутреннее хеширование).

Рисунок 6.2 – Разновидности методов разрешения коллизий

Методом цепочек, показанный на рисунке 6.3, называется метод, в котором для разрешения коллизий во все записи вводятся указатели, используемые для организации списков цепочек переполнения. В случае возникновения коллизии при заполнении таблицы в список для требуемого адреса хеш-таблицы добавляется еще один элемент.

Поиск в хеш-таблице с цепочками переполнения осуществляется следующим образом. Сначала вычисляется адрес по значению ключа. Затем осуществляется последовательный поиск в списке, связанном с вычисленным адресом.

Процедура удаления из таблицы сводится к поиску элемента и его удалению из цепочки переполнения.

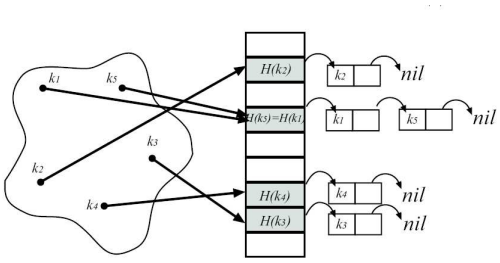


Рисунок 6.3 – Разрешение коллизий при добавлении элементов методом цепочек

Данный метод реализуется на основе линейного однонаправленного списка в динамической памяти, где информационное поле будет являться ключом.

```
Type Hash = ^PHash;
PHash = record
  Data: Datatype; {ключевое поле}
  Next:hash;
  End;
```

Объявление таблицы выглядит следующим образом:

```
Const
  n = …; {размерность хеш-таблицы}
var
  table = array [1..n] of hash; {хеш-таблица}
```

Метод открытой адресации состоит в том, чтобы, пользуясь каким-либо алгоритмом, обеспечивающим перебор элементов таблицы, просматривать их в поисках свободного места для новой записи.

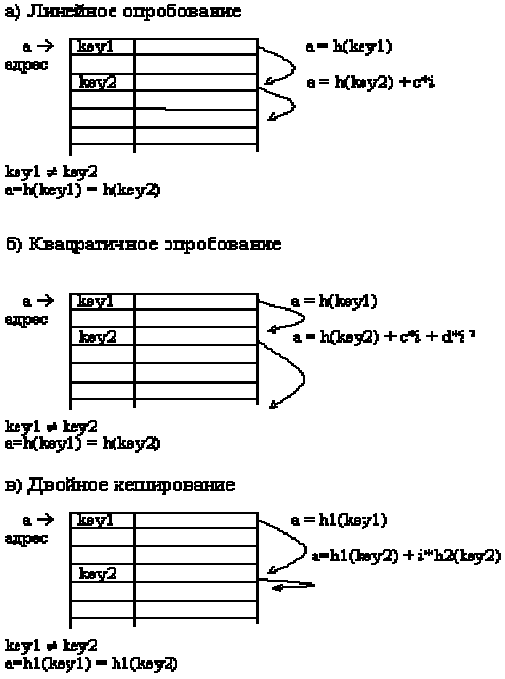


Рисунок 6.4 – Разрешение коллизий при добавлении элементов методами открытой адресации

6.2 Задания на лабораторную работу

1) Дан текстовый файл tel.txt, который представляет из себя телефонный справочник, содержащий строки следующего вида:

Петров

333333

Иванов

111111

…

Используя метод хеширования с открытой адресацией, написать программу поиска записи по первым трем буквам фамилии.

Указания: в качестве хеш-функции использовать:

- сумму кодов символов по модулю m;
- сумму квадратов кодов символов по модулю m.

В качестве метода адресации использовать:

- линейный;
- квадратичный;
- двойное хеширование.

Реализовать различные хеш-функции и методы открытой адресации.

2) То же условие задачи, что и в предыдущем задании, но использовать хеширование при помощи цепочек.

3) *Постройте хеш-таблицу для зарезервированных слов, используемого языка программирования (не менее 20 слов). При возникновении коллизий использовать любой из существующих методов их разрешения. Организовать поиск и добавление зарезервированного слова.

4) *В текстовом файле содержатся целые числа. Постройте хеш-таблицу из чисел файла. Осуществите поиск введенного целого числа в хеш-таблице. Сравните результаты количества сравнений при различном наборе данных в файле.

6.3 Примерный перечень вопросов

1) В чем заключается метод хеш-поиска?
2) Каков принцип построения хеш-таблиц?
3) Что такое хеш-таблица и как она используется?
4) Почему возможно возникновение коллизий?
5) Для чего используется хеш-функция и какие к ней предъявляются требования?
6) В каких ситуациях можно построить бесконфликтную хеш-таблицу?
7) Каковы методы устранения коллизий? Охарактеризуйте их эффективность в различных ситуациях.
8) Назовите преимущества открытого и закрытого хеширования.
9) В каком случае поиск в хеш-таблицах становится неэффективен?
10) Как выбирается метод изменения адреса при повторном хешировании?

7 Лабораторная работа №6. Реализация основных операций с двоичными деревьями поиска в динамической памяти

Лабораторная работа №6 предназначена для приобретения навыков работы с двоичными деревьями поиска в динамической памяти.

Разработать программы, в соответствии с заданиями в пункте 7.2 и представить отчет в соответствии с приложением В.

Последовательность выполнения лабораторной работы №6 представлена в приложении Б.

Заканчивается работа защитой с оценкой «зачтено» или «незачтено» в виде экспресс-опроса в рамках теоретического и практического материала, отнесенного к пунктам 7.1 и 7.2. Примерный перечень вопросов приведен в пункте 7.3.

На выполнение и защиту лабораторной работы отводится 4 академических часа.

7.1 Краткие теоретические сведения

7.1.1 Основные понятия, связанные с деревьями

Дерево — это совокупность элементов, называемых *узлами* (при этом один из них определен как *корень*), и отношений (родительский–дочерний), образующих иерархическую структуру узлов. Узлы могут являться величинами любого простого или структурированного типа, за исключением файлового. Узлы, которые не имеют ни одного последующего узла, называются *листьями*.

В *двоичном (бинарном) дереве* каждый узел может быть связан не более чем с двумя другими узлами. Рекурсивно двоичное дерево определяется так: *двоичное дерево* бывает либо пустым (не содержит ни одного узла), либо содержит узел, называемый *корнем*, а также два независимых поддерева — *левое поддерево* и *правое поддерево*.

Прямо из рекурсивного определения дерева можно вывести следующее ссылочное представление дерева в программе:

```
type
PTree = ^TTree;
TTree = record
  Data: DataType; {элемент дерева}
  Left, Right: PTree; {указатели на поддеревья}
end;
```

где Left, Right равны nil, если соответствующие поддеревья пусты.

Для обработки дерева достаточно знать адрес корневой вершины. Для хранения этого адреса надо ввести ссылочную переменную:

```
Var Tree: PTree
```

Высотой поддерева будем считать максимальную длину цепи y1, …, его вершин такую, что yi+1 – потомок yi для всех i. Высота пустого дерева равна нулю, высота дерева из одного корня – единице. Степенью вершины в дереве называется количество дуг, которое из нее выходит. Степень дерева равна максимальной степени вершины, входящей в дерево. При этом листьями в дереве являются вершины, имеющие степень нуль.

Двоичное дерево поиска может быть либо пустым, либо оно обладает таким свойством, что корневой элемент имеет большее значение узла, чем любой элемент в левом поддереве, и меньшее или равное, чем элементы в правом поддереве. Указанное свойство называется *характеристическим свойством двоичного дерева* поиска и выполняется для любого узла такого дерева, включая корень. Важное свойство такого дерева: все элементы его различны. Название двоичные деревья поиска получили по той причине, что скорость поиска в них примерно такая же, что и в отсортированных массивах: $O(n) = C \bullet \log_2 n$ (в худшем случае $O(n) = n$).

Основные операции над двоичными деревьями поиска:

– добавление элемента в дерево;

– удаление элемента из дерева;

– обход дерева (прямой, симметричный, обратный);

– поиск элементов в дереве.

Большинство операций над деревьями носят рекурсивный характер, поскольку дерево само по себе является рекурсивной структурой данных.

7.1.2 Основные операции с деревьями

Поиск элементов в дереве.

Алгоритм поиска в двоичном дереве очень прост:

Начиная с корневой вершины для каждого текущего поддерева надо выполнить следующие шаги:

– сравнить ключ вершины с заданным значением x;

– если заданное значение меньше ключа вершины, перейти к левому поддереву, иначе перейти к правому поддереву.

Поиск прекращается при выполнении одного из двух условий:

– либо, если найден искомый элемент;

– либо, если надо продолжать поиск в пустом поддереве, что является признаком отсутствия искомого элемента.

Добавление элемента в дерево.

Алгоритм добавления включает следующие шаги:

– выделение памяти для новой вершины;

– формирование информационной составляющей;

– формирование двух пустых ссылочных полей на будущих потомков;

– формирование в родительской вершине левого или правого ссылочного поля – адреса новой вершины.

Удаление элемента из дерева.

Теперь рассмотрим удаление вершины из двоичного дерева. По сравнению с добавлением удаление реализуется более сложным алгоритмом, поскольку добавляемая вершина всегда является терминальной, а удаляться может любая, в

том числе и нетерминальная. При этом может возникать несколько различных ситуаций.

Рассмотрим фрагмент двоичного с целыми ключами.

Ситуация 1. Удаляемая вершина не имеет ни одного потомка, т.е. является терминальной. Удаление реализуется очень легко: обнулением соответствующего указателя у родителя.

Ситуация 2. Удаляемая вершина имеет только одного потомка. В этом случае удаляемая вершина вместе со своим потомком и родителем образуют фрагмент линейного списка. Удаление реализуется простым изменением указателя у родительского элемента.

Ситуация 3. Пусть удаляемая вершина имеет двух потомков. Этот случай наиболее сложен, поскольку нельзя просто в родительской вершине изменить соответствующее ссылочное поле на адрес одного из потомков удаляемой вершины. Это может нарушить структуру дерева поиска.

Существует специальное правило для определения вершины, которая должна заменить удаляемую. Это правило состоит из двух взаимоисключающих действий:

– либо войти в левое поддерево удаляемой вершины и в этом поддереве спуститься как можно глубже, придерживаясь только правых потомков; это позволяет найти в дереве ближайшую меньшую вершину;

– либо войти в правое поддерево удаляемой вершины и спуститься в нем как можно глубже, придерживаясь только левых потомков; это позволяет найти ближайшую большую вершину.

Обход дерева.

Обход в прямом направлении:

– обработать корневую вершину текущего поддерева;

– перейти к обработке левого поддерева таким же образом;

– обработать правое поддерево таким же образом.

Симметричный обход:

– рекурсивно обработать левое поддерево текущего поддерева;

– обработать вершину текущего поддерева;

– рекурсивно обработать правое поддерево.

Обход в обратном направлении:

– рекурсивно обработать левое поддерево текущего поддерева;

– рекурсивно обработать правое поддерево;

– затем – вершину текущего поддерева.

7.2 Задания на лабораторную работу

1) Реализовать следующие операции с двоичными деревьями поиска:
   – формирование бинарного дерева;
   – обход (прямой, симметричный, обратный) бинарного дерева;
   – удаление заданной вершины из бинарного дерева;
   – поиск заданной вершины в бинарном дереве;
   – печать бинарного дерева на экран;
   – проверка пустоты бинарного дерева;
   – определение высоты бинарного дерева.

Каждую операцию описать в виде отдельной процедуры.

Главная программа должна:

– создать пустое дерево;

– организовать цикл для добавления вершины с вызовом соответствующей подпрограммы с последующим построчным выводом дерева (должна быть реализована двумя способами: заполнение случайными числами и заполнение заданными пользователем);

– предоставить возможность в любой момент вызвать подпрограмму удаления дерева с фактическим параметром;

– предоставить возможность в любой момент вызвать подпрограмму обхода дерева с фактическим параметром.

2) *Вершины дерева вещественные числа. Описать процедуру или функцию, которая:
   – вычисляет среднее арифметическое значение всех вершин дерева;

– добавляет в дерево вершину со значением, вычисленным в предыдущей процедуре (функции).

3) *Записи вершин дерева - вещественные числа. Описать процедуру, которая удаляет все вершины с отрицательными записями.

4) *Записи вершин дерева - вещественные числа. Описать процедуру или функцию, которая:

– находит максимальное или минимальное значение записей вершин непустого дерева;

– печатает записи из всех листьев дерева.

5) *Оператор мобильной связи организовал базу данных абонентов, содержащую сведения о телефонах, их владельцах и используемых тарифах, в виде бинарного дерева. Составьте программу, которая:

– обеспечивает начальное формирование базы данных в виде бинарного дерева;

– производит вывод всей базы данных;

– производит поиск владельца по номеру телефона;

– выводит наиболее востребованный тариф (по наибольшему числу абонентов).

7.3 Примерный перечень вопросов

1) С чем связана популярность использования деревьев в программировании?
2) Можно ли список отнести к деревьям? Ответ обоснуйте.
3) Какие данные содержат адресные поля элемента бинарного дерева?
4) Что такое дерево, двоичное дерево, поддерево?
5) Как рекурсивно определяется дерево?
6) Какие основные понятия связываются с деревьями?
7) Какие основные операции характерны при использовании деревьев?
8) Как программно реализуется алгоритм операции обхода дерева?

9) Как программно реализуется алгоритм операции добавления элемента в дерево?

10) Как программно реализуется алгоритм операции удаления элемента из дерева?

11) Как программно реализуется алгоритм операции поиска элемента в дереве?

8 Лабораторная работа №7. Реализация основных операций с рандомизованными деревьями в динамической памяти

Лабораторная работа №7 предназначена для приобретения навыков работы с рандомизованными деревьями в динамической памяти.

Разработать программы, в соответствии с заданиями в пункте 8.2 и представить отчет в соответствии с приложением В.

Последовательность выполнения лабораторной работы №7 представлена в приложении Б.

Заканчивается работа защитой с оценкой «зачтено» или «незачтено» в виде экспресс-опроса в рамках теоретического и практического материала, отнесенного к пунктам 8.1 и 8.2. Примерный перечень вопросов приведен в пункте 8.3.

На выполнение и защиту лабораторной работы отводится 4 академических часа.

8.1 Краткие теоретические сведения

8.1.1 Основные понятия, связанные с рандомизованными деревьями

На практике время выполнения основных операций с деревьями определяется высотой дерева. Высота будет минимальной $\Theta(\log n)$, если для каждого узла количество вершин в левом и правом поддереве отличается не более чем на единицу. Такое дерево называется идеально сбалансированным. На практике очень сложно добиться идеальной сбалансированности. Существует много способов добиться приближенной сбалансированности. Наиболее известны АВЛ-деревья, красно-черные деревья, 2-3 деревья и т.д. Во всех предложенных структурах поиск производится так же, как и в деревьях двоичного поиска, а при добавлении и удалении с помощью специальных операций, называемых вращениями, производится перестройка дерева, с целью добиться лучшей сбалансированности.

Рассмотрим наиболее простой вариант сбалансированных деревьев – рандомизованные деревья.

Заметим, что при добавлении в дерево данных, упорядоченных случайным образом, получается дерево, близкое к сбалансированному. При добавлении в рандомизованное дерево мы используем генератор случайных чисел для моделирования случайного порядка исходных данных. Так, если в дерево добавляется n элементов, то вероятность для каждого элемента оказаться при случайном упорядочении первым и попасть в корень дерева равна 1/n. В рандомизованном дереве мы с такой вероятностью помещаем элемент в корень дерева.

Для определения вероятности требуется знать количество элементов в каждом поддереве. Будем хранить это количество в корне поддерева в поле N и изменять при добавлении и удалении элементов. Получим следующую структуру данных.

```
Type TTree = ^TNode;
TNode = record
     X: DataType;
     L,R: TTree;
     N: Word;
end;
```

8.1.2 Основные операции с рандомизованными деревьями

Добавление элемента в дерево.

Процедура добавления будет рекурсивной: для пустого дерева она просто создаст новый узел, для непустого – рекурсивно добавит узел в корень левого или правого поддерева, а затем выполнит левое или правое вращение. Подробнее рассмотрим левое вращение. Пусть узел добавлен в корень левого поддерева, и его нужно вытащить в корень дерева. Обозначим указатель на корень дерева T, на корень левого поддерева S. Для сохранения свойства двоичного дерева поиска, нужно, чтобы правое поддерево S стало левым поддеревом T. Наглядно левое вращение представлено на рисунке 8.1.

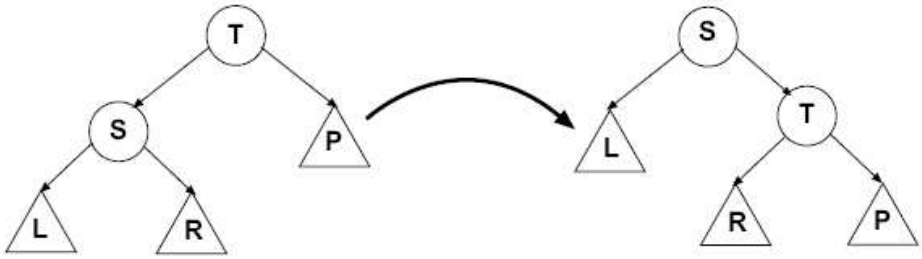


Рисунок 8.1 – Левое вращение

Удаление элемента из дерева.

Удаление элемента в рандомизованном дереве должно сохранять сбалансированность. Если одно из поддеревьев содержит больше элементов, то разумно, чтобы элемент для замены удаляемого извлекался с большей вероятностью именно из этого дерева.

Оно основано на операции объединения деревьев. После удаления элемента, мы должны объединить его левое и правое поддеревья. Объединение деревьев выполняется рекурсивно. Если одно из деревьев пусто, то результат объединения равен другому дереву. Если оба непусты, то мы в корень объединенного дерева помещаем либо корень левого дерева и присоединяем справа объединение его правого поддерева и второго поддерева, либо корень правого поддерева и аналогично другое дерево присоединяем слева. Пусть левое дерево содержит N_1 узлов, а правое N_2 узлов. Тогда корень объединения выбирается из левого дерева с вероятностью $\frac{N_1}{N_1 + N_2}$, из правого – с вероятностью $\frac{N_2}{N_1 + N_2}$. Наглядно объединение деревьев показано на рисунке 8.2.

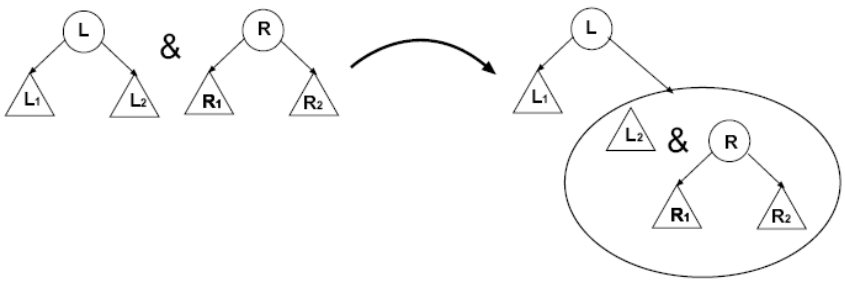


Рисунок 8.2 – Объединение деревьев

Поиск элементов в дереве.

Производится аналогично, как и с двоичными деревьями поиска.

Обход дерева.

Производится аналогично, как и с двоичными деревьями поиска.

8.2 Задания на лабораторную работу

Реализовать следующие операции с рандомизованными деревьями:

- добавление новой вершины;
- обход (прямой, симметричный, обратный);
- удаление заданной вершины;
- поиск заданной вершины;
- вывод дерева на экран.

Каждую операцию описать в виде отдельной процедуры или функции.

При запуске программы должно появляться меню, реализующее основные операции, выполняемые с рандомизованными деревьями.

8.3 Примерный перечень вопросов

1) Что такое рандомизованное дерево?
2) По какому правилу оно строится?
3) Как определяется количество узлов в рандомизованном дереве?
4) Какие основные операции характерны при использовании деревьев?
5) Как происходит добавление элемента в рандомизованное дерево?
6) Как происходит удаление элемента из рандомизованного дерева?

9 Лабораторная работа №8. Реализация основных алгоритмов на графах

Лабораторная работа №8 предназначена для приобретения навыков работы с графами в динамической памяти.

Разработать программы, в соответствии с заданиями в пункте 9.2 и представить отчет в соответствии с приложением В.

Последовательность выполнения лабораторной работы №8 представлена в приложении Б.

Заканчивается работа защитой с оценкой «зачтено» или «незачтено» в виде экспресс-опроса в рамках теоретического и практического материала, отнесенного к пунктам 9.1 и 9.2. Примерный перечень вопросов приведен в пункте 9.3.

На выполнение и защиту лабораторной работы отводится 6 академических часов.

9.1 Краткие теоретические сведения

9.1.1 Основные понятия, связанные с графами и способы представления

Графом называется пара $G = \langle V, E \rangle$, где V – некоторое множество, которое называют множеством *вершин* графа, а E – отношение на V ($E \subset V \times V$) - множество *ребер* графа. То есть все ребра из множества E соединяют некоторые пары точек из V.

Если отношение E симметричное (т.е. $(u,v) \in E \Leftrightarrow (v,u) \in E$), то граф называют *неориентированным*, в противном случае граф называют *ориентированным*. Фактически для каждого из ребер ориентированного графа указаны начало и конец, то есть пара (u, v) упорядочена, а в неориентированном графе (u, v) = (v, u).

Если в графе существует ребро (u, v), то говорят, что вершина v *смежна* с вершиной u (в ориентированном графе отношение смежности несимметрично).

Путем из вершины u в вершину v длиной k ребер называют последовательность ребер графа $U = <(u, v_1), (v_1, v_2), ..., (v_{k-1}, v)>$. Часто тот же путь

обозначают последовательностью вершин $< u, v_1, ..., v_{k-1}, v >$. Если для данных вершин *u*, *v* существует путь из *u* в *v*, то говорят, что вершина *v* *достижима* из *u*. Путь называется *простым*, если все вершины в нем различны. *Циклом* называется путь, в котором начальная вершина совпадает с конечной. При этом циклы, отличающиеся лишь номером начальной точки, отождествляются.

Граф называется *связанным*, если для любой пары его вершин существует путь из одной вершины в другую.

Если каждому ребру графа приписано какое-то число (*вес*), то граф называют *взвешенным*.

Количество вершин графа G называется его порядком. Граф порядка n называется *помеченным (нагруженным) графом*, если его вершины перенумерованы целыми числами 1,2,...,n.

Степень (валентность) вершины – это количество ребер, инцидентных этой вершине (количество смежных с ней вершин).

Маршрутом в графе называется чередующаяся последовательность вершин и ребер $v_0, e_1, v_1, e_2, v_2, ..., e_k, v_k$, в которой любые два соседних элемента инциденты.

При программировании вершины графа обычно сопоставляют числам от 1 до *N*, где $N = |V|$ - количество вершин графа, и рассматривают $V = \{1, 2, ..., N\}$. Ребра нумерую числами от 1 до *M*, где $M = |E|$. Каждому ребру и каждой вершине сопоставлен вес – целое положительное число.

Существуют следующие способы задания графов:

– матрица смежности.

Матрица смежности это двумерный массив размером N*N, где N – количество вершин в графе:

```
Type
  Matrix = array [1..N, 1..N] of Integer;
Var
  A: Matrix;
```

Для невзвешенного графа $A[i][j] = true$ (или 1), если $(i, j) \in E$ и $A[i][j] = false$ (или 0) в противном случае. Для взвешенного графа $A[i][j]$ равно весу соответствующего ребра, а отсутствие ребра в ряде задач удобно обозначать бесконечностью. Для неориентированных графов матрица смежности всегда симметрична относительно главной диагонали ($i = j$). С помощью матрицы смежности легко проверить, существует ли в графе ребро, соединяющее вершину i с вершиной j;

– матрица инцидентности.

Матрица инцидентности это двумерный массив размером N*M, где N – количество вершин в графе, M – количество ребер в графе:

```
Type
  Matrix = array [1..N, 1..M] of Integer;
Var
  A: Matrix;
```

В случае *неориентированного графа* столбец, соответствующий ребру {x,y}, содержит 1 в строках, соответствующих вершинам x и y, и 0 - в остальных строках. Для *ориентированного графа*, столбец, соответствующий дуге (x,y)∈E, содержит:

а) (-1) в строке, соответствующей вершине x;

б) 1 в строке, соответствующей вершине y;

в) 0 во всех остальных строках;

– матрица весов.

Матрица весов это двумерный массив размером N*N, где N – количество вершин в графе:

```
Type
  Matrix = array [1..N, 1..N] of Integer;
Var
```

```
    A: Matrix;
```

Для взвешенного графа $A[i][j]$ равно весу соответствующего ребра, а отсутствие ребра в ряде задач удобно обозначать бесконечностью.

– список ребер.

Список ребер – это одномерный массив размером N, содержащий список пар вершин, инцидентных с одним ребром графа:

```
Type
  TBranchList = array [1..N] of record
    Node1,          {1-я вершина, инцидентная ребру}
    Node2: string;  {2-я вершина, инцидентная ребру}
    Weight: Integer;  {вес ребра}
  end;
Var
    A: TBranchList;
```

– список смежных вершин.

Списки смежных вершин – это одномерный массив размером N, содержащий для каждой вершины указатели на списки смежных с ней вершин:

```
Type
  PNode=^Node;
  Node = record          {смежная вершина}
     Name: string;       {имя смежной вершины}
     Weight: integer;    {вес ребра}
     Next: PNode;        {следующая смежная вершина}
   end;
Graph = array [1..N] of record
```

```
    NodeWeight: integer;  {вес вершины}
    Name: string;         {имя вершины}
    List: PNode;          {указатель на список смежных}
end;
var
    A: Graph;
```

Графическое представление описанных выше реализаций графа показано на рисунке 9.1.

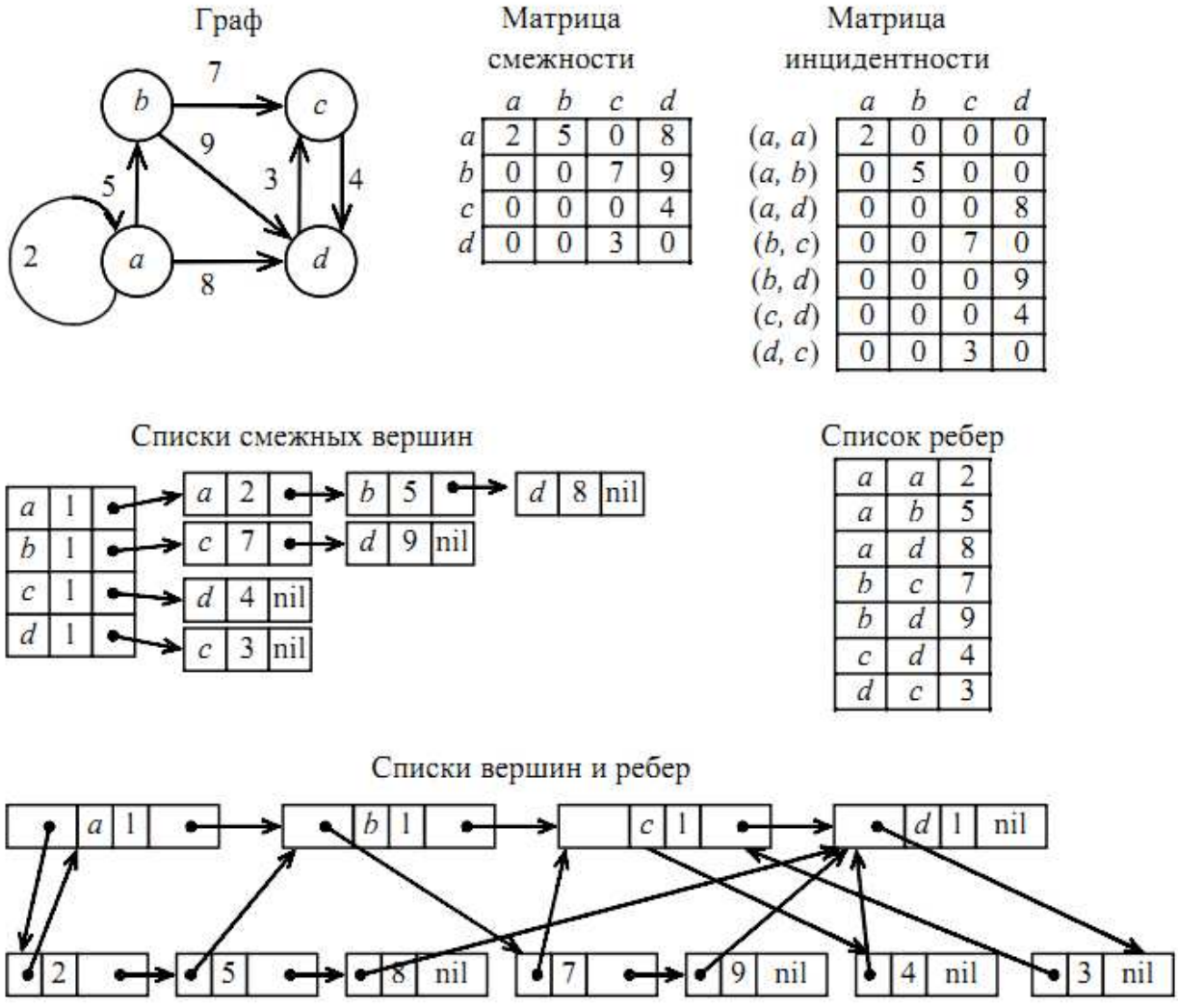


Рисунок 9.1 – Граф и его реализации

9.1.2 Алгоритмы на графах

Алгоритмы обхода графов.

Существует множество алгоритмов на графах, в основе которых лежит систематический перебор вершин графа, причем такой, что каждая вершина просматривается (посещается) в точности один раз. Поэтому важной задачей является нахождение «хороших» методов такого перебора.
Под обходом графа (поиском на графе) мы будем понимать процесс систематического просмотра всех вершин графа с целью отыскания вершин, удовлетворяющих некоторому условию.

Обход (поиск) в глубину.

Идея алгоритма: из текущей вершины движемся в первую вершину, смежную с текущей, в которой мы еще не были, если таковая есть. Если таковой нет, то возвращаемся в вершину, из которой мы попали в текущую. Если же таковой нет, и мы оказались в исходной вершине (возвращаться некуда), то это означает, что перебор вершин графа закончен.

Для программной реализации используются следующие структуры: Nnew - массив признаков. Номера просмотренных вершин графа запоминаются в стеке St, указатель стека - переменная yk.

Обход (поиск) в ширину.

Идея алгоритма: метод поиска в ширину получается из программы поиска в глубину, если мы заменим стек возврата на очередь. Эта простая замена модифицирует порядок обхода вершин так, что обход идет равномерно во все стороны, а не вглубь как при поиске в глубину.

Алгоритмы поиска циклов в графе.

Нахождение эйлерова цикла.

Определение: Если граф имеет цикл (не обязательно простой), содержащий все ребра графа по одному разу, то такой цикл называется *эйлеровым* циклом.

Идея алгоритма: начиная с произвольной вершины, строим путь, удаляя ребра и запоминая вершины в стеке, до тех пор, пока множество смежности очередной

вершины не окажется пустым, что означает, что путь удлинить нельзя. Заметим, что при этом мы с необходимостью придем в ту вершину, с которой начали. В противном случае это означало бы, что вершина v имеет нечетную степень, что невозможно по условию. Таким образом, из графа были удалены ребра цикла, а вершины цикла были сохранены в стеке S. Заметим, что при этом степени всех вершин остались четными. Далее вершина v выводится в качестве первой вершины эйлерова цикла, а процесс продолжается, начиная с вершины, стоящей на вершине стека.

Нахождение гамильтонова цикла.

Определение: Граф называется *гамильтоновым*, если в нем имеется цикл, содержащий каждую вершину этого графа. Сам цикл также называется гамильтоновым.

Идея алгоритма: в основе лежит перебор с возвратом (backtracking). Начинаем поиск решения, например, с первой вершины графа. Предположим, что уже найдены первые k компонент решения. Рассматриваем ребра, выходящие из последней вершины. Если есть такие рёбра, что идут в ранее не просмотренные вершины, то включаем эту вершину в решение и помечаем ее как просмотренную. Получена (k+1) компонента решения. Если такой вершины нет, то возвращаемся к предыдущей вершине и пытаемся найти ребро, выходящее из нее в другую вершину. Решение получено при просмотре всех вершин графа и возможности достичь из последней первой вершины. Решение (цикл) выводится и продолжается процесс нахождения следующих циклов.

Алгоритм нахождения кратчайших путей в графе.

Алгоритм Дейкстры.

Определение: Дан простой взвешенный граф $G(V,E)$ без петель и дуг отрицательного веса. Найти кратчайшие пути от некоторой вершины a графа G до всех остальных вершин этого графа.

Идея алгоритма: метка самой вершины u полагается равной 0, метки остальных вершин — бесконечности. Это отражает то, что расстояния от u до других вершин пока неизвестны. Все вершины графа помечаются как

непосещенные. Если все вершины посещены, алгоритм завершается. В противном случае из еще не посещенных вершин выбирается вершина u, имеющая минимальную метку. Мы рассматриваем всевозможные маршруты, в которых u является предпоследним пунктом. Вершины, соединенные с вершиной u ребрами, назовем соседями этой вершины. Для каждого соседа рассмотрим новую длину пути, равную сумме текущей метки u и длины ребра, соединяющего u с этим соседом. Если полученная длина меньше метки соседа, заменим метку этой длиной. Рассмотрев всех соседей, пометим вершину u как посещенную и повторим шаг.

9.2 Задания на лабораторную работу

Реализовать основные алгоритмы на графах, описанные в пункте 9.1.

9.3 Примерный перечень вопросов

1) Что такое граф? Что такое ребро и дуга графа?
2) Что такое ориентированный граф и неориентированный граф?
3) Какие вершины называют смежными? Какие ребра называют смежными? Что означает слово инцидентные?
4) Что такое вес вершины, вес ребра?
5) Какие способы представления графов существуют?
6) В чем разница между алгоритмами поиска в ширину и поиска в глубину?
7) Описать алгоритм нахождения кратчайшего пути.
8) Описать алгоритмы нахождения эйлерова и гамильтонова цикла.

10 Лабораторная работа № 9. Сортировка методом прямого включения и ее модификации

Лабораторная работа №9 предназначена для приобретения навыков применения сортировки методом прямого включения и ее модификаций на языке программирования Turbo Pascal.

Разработать программы, в соответствии с заданиями в пункте 10.2 и представить отчет в соответствии с приложением В.

Последовательность выполнения лабораторной работы №9 представлена в приложении Б.

Заканчивается работа защитой с оценкой «зачтено» или «незачтено» в виде экспресс-опроса в рамках теоретического и практического материала, отнесенного к пунктам 10.1 и 10.2. Примерный перечень вопросов приведен в пункте 10.3.

На выполнение и защиту лабораторной работы отводится 4 академических часа.

10.1 Краткие теоретические сведения

При обработке данных в ЭВМ важно знать и информационное поле элемента, и его размещение в памяти машины. Для этих целей используется сортировка. Итак, сортировка – это расположение данных в памяти в регулярном виде по их ключам. Регулярность рассматривают как возрастание значения ключа от начала к концу в массиве.

Различают следующие типы сортировок:

– внутренняя сортировка - это сортировка, происходящая в оперативной памяти машины;

– внешняя сортировка - сортировка во внешней памяти.

Если сортируемые записи занимают большой объем памяти, то их перемещение требует больших затрат. Для того, чтобы их уменьшить, сортировку

производят в таблице адресов ключей, делают перестановку указателей, т.е. сам массив не перемещается. Это метод сортировки таблицы адресов.

При сортировке могут встретиться одинаковые ключи. В этом случае при сортировке желательно расположить после сортировки одинаковые в том же расположении, что и ключи в исходном файле. Это устойчивая сортировка.

Эффективность сортировки можно рассмотреть с нескольких критериев:

- время, затрачиваемое на сортировку;
- объем оперативной памяти, требуемой для сортировки;
- время, затраченное программистом на написание программы.

Рассмотрим второй критерий подробнее: мы можем подсчитать количество сравнений при выполнении сортировки или количество перемещений.

Предположим, что число сравнений определяется формулой:

$$C = 0{,}01n^2 + 10n$$

Если n<1000, то второе слагаемое больше.

Если n>1000, то первое слагаемое больше.

То есть, при малых n порядок сравнения равен 0 , а при больших n он равен n.

Различают следующие методы сортировки:

- строгие (прямые) методы;
- улучшенные методы.

Рассмотрим преимущества прямых методов:

- программы этих методов легко понимать, и они коротки. Напомним, что сами программы также занимают память;
- прямые методы особенно удобны для объяснения характерных черт основных принципов большинства сортировок;
- усложненные методы требуют небольшого числа операций, но эти операции обычно сами более сложны, и поэтому для достаточно малых n прямые методы оказываются быстрее, хотя при больших n их использовать, конечно, не следует.

Методы сортировки «на том же месте» можно разбить в соответствии с определяющими их принципами на 3 категории:

- сортировки с помощью включения (by insertion);
- сортировки с помощью выделения (by selection);
- сортировки с помощью обменов (by exchange).

Сортировка методом прямого включения.

Такой метод широко используется при игре в карты. Элементы (карты) мысленно делятся на уже «готовую» последовательность a[1],...,a[i-1] и исходную последовательность. При каждом шаге, начиная с i=2 и увеличивая i каждый раз на единицу, из исходной последовательности извлекается i-й элемент и перекладывается в готовую последовательность, при этом он вставляется на нужное место.

Алгоритм сортировки прямым включением таков:

```
for x:=2 to n do
  x:=a[i]
  включение x на соответствующее место среди a[1]...a[i]
end
```

В реальном процессе поиска подходящего места удобно, чередуя сравнения x сравнивается с очередным элементом a[j], а затем либо x вставляется на свободное место, либо a[j] сдвигается (передается) вправо и процесс «уходит» влево. Обратите внимание, что процесс просеивания может закончиться при выполнении одного из двух следующих различных условий:

- найден элемент a[j] с ключом, меньшим, чем ключ x;
- достигнут левый конец готовой последовательности.

Листинг процедуры метода прямого включения выглядит следующим образом:

```
Procedure PraminSort(var a: mass; n: integer);
Var i, j, x: integer;
Begin
```

```
For i:=2 to n do
begin x: = a[i]; a[0]: =a[i]; j: = i;
While x< a[j-1] do
begin a[j]: = a[j-1]; j: = j-1 end;
a[j]: = x;
end;
End;
```

Среднее количество перемещений:

$$M_{среднее} = \frac{n^2 + 9n - 10}{4}$$

Среднее количество сравнений:

$$C_{среднее} = \frac{n^2 + n - 2}{4}$$

Усовершенствованием сортировки методом прямого включения является сортировка Шелла.

В 1959 г. Д. Шеллом было предложено усовершенствование сортировки с помощью прямого включения.

Идея метода: все элементы массива разбиваются на группы таким образом, что в каждую группу входят элементы, отстоящие друг от друга на некоторое число позиций L. Элементы каждой группы сортируются. После этого все элементы вновь объединяются и сортируются в группах, при этом расстояние между элементами уменьшается. Процесс заканчивается после того, как будет проведено упорядочивание элементов с расстоянием между ними, равным 1.

Проиллюстрируем метод Шелла на следующем массиве:

17, 2, 34, 20, 11, 13, 31, 29

Сначала рассмотрим вариант, когда первоначальное значение L равно половине числа элементов в массиве, а каждое последующее значение вдвое меньше предыдущего. Заметим, что обмениваются элементы, которые отстоят на величину шага. Если при сравнении 2-х элементов обмена не произошло, то места

сравниваемых элементов сдвигаются вправо на одну позицию. Если обмен произошёл, то происходит сдвиг соответствующих сравниваемых элементов на L.

Исходный массив:

17 2 34 20 11 13 31 29

L=4. Полученный массив после сортировки:

11 2 31 20 17 13 34 29

L=2. Полученный массив:

11 2 17 13 31 20 34 29

L=1. Полученный массив:

2 11 13 17 20 29 31 34

Фрагмент программы сортировки Шелла:

```
L: = n div 2; {начальное расстояние между сортируемыми элементами}
Repeat {внешний цикл по L}
i: = L+1;
Repeat
j: = i – L;
Repeat
if a[i] > a[j + L] then
begin
R: = a[j]; a[j]: = a[j + L]; a[j + L]: = R; j: = j – L
end
else goto 1;
Until j < 1;  1:  i: = i + 1;
Until i > n;  L: = L div 2;
Until L: =0;
```

10.2 Задания на лабораторную работу

В ремонтной мастерской находятся несколько (N) машин. О них имеются следующие сведения: номер, марка, имя владельца, дата последнего ремонта (число, месяц, год), день, к которому машина должна быть отремонтирована (число, месяц, год).

Требуется (согласно варианту) :

1) Расположить по алфавиту имена владельцев и, соответственно, вывести информацию об их машинах.

2) *Исходя из того, что машина, дата окончания ремонта которой раньше, должна ремонтироваться в первую очередь, вывести порядок ремонта автомобилей.

3) *Вывести по возрастанию даты конца ремонта всех машин, которые ранее не ремонтировались.

4) Вывести по алфавиту в обратном порядке владельцев автомобилей марки «Мерседес».

5) *Вывести по алфавиту марки машин, которые должны быть отремонтированы раньше всех (дата конца ремонта меньше 01.08.96).

6) Вывести по возрастанию номера машин марки «Жигули».

7) Вывести по убыванию номера машин марки «Мерседес».

10.3 Примерный перечень вопросов

1) В чем состоит суть метода сортировки вставками?
2) Какие шаги выполняет алгоритм сортировки вставками?
3) Как программно реализуется сортировка вставками?
4) В чем достоинства и недостатки метода сортировки вставками?
5) Приведите практический пример сортировки массива методом вставок.
6) В чем состоит суть сортировки методом Шелла?
7) За счет чего метод Шелла дает лучшие показатели по сравнению с простейшими методами?

8) Приведите практический пример сортировки массива методом Шелла.

9) Какой фактор оказывает наибольшее влияние на эффективность сортировки методом Шелла?

10) Какие последовательности шагов группировки рекомендуются для практического использования в методе Шелла?

11) Как программно реализуется сортировка методом Шелла?

11 Лабораторная работа № 10. Сортировка методом прямого выбора

Лабораторная работа №10 предназначена для приобретения навыков написания программ для сортировки методом прямого выбора на языке программирования Turbo Pascal.

Разработать программы, в соответствии с заданиями в пункте 11.2 и представить отчет в соответствии с приложением В.

Последовательность выполнения лабораторной работы №10 представлена в приложении Б.

Заканчивается работа защитой с оценкой «зачтено» или «незачтено» в виде экспресс-опроса в рамках теоретического и практического материала, отнесенного к пунктам 11.1 и 11.2. Примерный перечень вопросов приведен в пункте 11.3.

На выполнение и защиту лабораторной работы отводится 4 академических часа.

11.1 Краткие теоретические сведения

В общем сортировку следует понимать как процесс перегруппировки заданного множества объектов в некотором определенном порядке. Цель сортировки облегчить последующий поиск элементов в таком отсортированном множестве. Это почти универсальная, фундаментальная деятельность. Мы встречаемся с отсортированными объектами в телефонных книгах, в списках подоходных налогов, в оглавлениях книг, в библиотеках, в словарях, на складах, почти везде, где нужно искать хранимые объекты. Даже малышей учат держать свои вещи «в порядке», и они уже сталкиваются с некоторыми видами сортировок задолго до того, как познакомятся с азами арифметики.

Таким образом, разговор о сортировке вполне уместен и важен, если речь идет об обработке данных. Что может легче сортироваться, чем данные? Тем не менее наш первоначальный интерес к сортировке основывается на том, что при

построении алгоритмов мы сталкиваемся со многими весьма фундаментальными приемами. Почти не существует методов, с которыми не приходится встречаться при обсуждении этой задачи. В частности, сортировка это идеальный объект для демонстрации огромного разнообразия алгоритмов, все они изобретены для одной и той же задачи, многие в некотором смысле оптимальны, большинство имеет свои достоинства. Поэтому это еще и идеальный объект, демонстрирующий необходимость анализа производительности алгоритмов. К тому же на примерах сортировок можно показать, как путем усложнения алгоритма, хотя под рукой и есть уже очевидные методы, можно добиться значительного выигрыша в эффективности.

Выбор алгоритма зависит от структуры обрабатываемых данных это почти закон, но в случае сортировки такая зависимость столь глубока, что соответствующие методы были даже разбиты на два класса сортировку массивов и сортировку файлов последовательностей. Иногда их называют внутренней и внешней сортировкой, поскольку массивы хранятся в быстрой оперативной, внутренней памяти машины со случайным доступом, а файлы обычно размещаются в более медленной, но и более емкой внешней памяти, на устройствах, основанных на механических перемещениях дисках или лентах . Уже на примере сортировки пронумерованных карточек становится очевидным существенное различие в этих подходах. Если карты «выстроены» в виде массива, то они как бы лежат перед сортирующим, он видит каждую из них и имеет к ней доступ. Если же карты образуют файл, то это предполагает, что видна только верхняя карта в каждой из стопок. Такое ограничение, конечно же, серьезно повлияет на метод сортировки, но ничего не поделаешь: ведь карточек может быть так много, что все они на столе не поместятся.

Прежде чем идти дальше, введем некоторые понятия и обозначения. Ими мы будем пользоваться далее. Если у нас есть элементы a1, a2,..., an, то сортировка есть перестановка этих элементов массив ak1, ak2, ..., akn, где при некоторой упорядочивающей функции f выполняются отношения f (ak1) <= f (ak2) <= ... <= f (akn).

Обычно упорядочивающая функция не вычисляется по какому-либо правилу, а хранится как явная компонента поля каждого элемента. Ее значение называется ключом кеу элемента. Поэтому для представления элементов хорошо подходят такие образования, как запись, показанная на рисунке 11.1.

Говоря об алгоритмах сортировки, мы будем обращать внимание лишь на компоненту - ключ, другие же компоненты можно даже и не определять, как представлено на рисунке 11.2. Поэтому из наших дальнейших рассмотрений вся сопутствующая информация. Чтобы уменьшить эти затраты, сортировку производят в таблице адресов ключей. После сортировки перестанавливают указатели. Это метод сортировки таблицы адресов, показанный на рисунке 11.3. Метод сортировки называется устойчивым, если в процессе сортировки относительное расположение элементов с равными ключами не изменяется. Устойчивость сортировки часто бывает желательной, если речь идет об элементах, уже упорядоченных (отсортированных) по некоторым вторичным ключам (т.е. свойствам) , не влияющим на основной ключ.

Рисунок 11.1 – Представления элементов массива

Рисунок 11.2 – Отсортированный массив

Рисунок 11.3 – Массив отсортированный другим методом

Основное условие: выбранный метод сортировки массивов должен экономно использовать доступную память. Это предполагает, что перестановки, приводящие элементы в порядок, должны выполняться на том же месте, т.е. методы, в которых элементы из массива A передаются в результирующий массив H, представляют существенно меньший интерес. Ограничив критерием экономии памяти наш выбор нужного метода среди многих возможных, мы будем сначала классифицировать методы по их экономичности, т.е. по времени их работы. Хорошей мерой эффективности может быть C - число необходимых сравнений ключей и M - число пересылок (перестановок) элементов.

Эти числа - функции от n- числа сортируемых элементов. Хотя улучшенные алгоритмы сортировки требуют порядка n*log n сравнений, мы разберем простой метод, который причисляется к, так называемым, прямым, где требуется порядка n^2 сравнений ключей. К достоинствам прямых методов относятся :

– прямые методы особенно удобны для объяснения характерных черт основных принципов большинства сортировок;

– программы этих методов легко понимать, и они коротки. Напомним, что сами программы также занимают память;

– улучшенные методы требуют небольшого числа операций, но эти операции обычно сами более сложны, и поэтому для достаточно малых n прямые методы оказываются быстрее, хотя при больших n их использовать, конечно, не следует.

Методы сортировки «на том же месте» можно разбить в соответствии с определяющими их принципами на три основные категории:

- сортировки прямым включением (By insertion);
- сортировки прямым выделением (By selection);
- сортировки прямым обменом (By exchange) .

К прямым методам относится метод прямого выбора.

Основная идея:

1) выбирается элемент с минимальным ключом;
2) он меняется местами с первым элементом;
3) этот процесс повторяется с оставшимися (n – 1) элементами, (n – 2) элементами и т. д. до тех пор, пока не останется один, самый большой элемент.

Алгоритм сортировки прямого выбора можно сформулировать так:

for i:=1 to n-1 do begin

1) присвоить к индекс наименьшего из a[i], …, a[n];
2) поменять местами a[i] и a[k];

end;

Этот алгоритм в некотором смысле противоположен прямому включению. При прямом включении на каждом шаге рассматривается только один очередной элемент исходной последовательности и все элементы готовой последовательности, среди которых отыскивается точка включения. При прямом выборе для поиска одного наименьшего элемента рассматриваются все элементы исходной последовательности, и найденный помещается как очередной элемент в готовую последовательность.

Фрагмент программы:

```
for i: =1 to n-1 do
begin
k: =i;
x: = a[i];
For j: = i+1 to n do
```

```
if a[j] < x then
begin
k :=j; x: = a[j]
end;
a[k]:= a[i];
a[i]:=x
end;
```

Эффективность алгоритма.

Количество перемещений

$$M = \frac{N}{2} \cdot (N-1) = \frac{N^2 - N}{2}$$

Количество сравнений, когда массив упорядочен

$$C_{\min} = 3 \cdot (N-1)$$

Количество сравнений, когда массив обратно отсортирован

$$C_{\max} = C_{\min} \cdot \frac{N}{2} = 3 \cdot (N-1) \cdot \frac{N}{2}$$

В худшем случае сортировка прямым выбором дает порядок n^2, как и для числа сравнений, так и для числа перемещений.

11.2 Задания на лабораторную работу

Создать группу из N студентов. Ввести их: фамилия, имя, год рождения, оценки по предметам: стр. и алг. данных, высш. математика, физика, программирование, общий балл сдачи сессии; Разработать программу с использованием метода «прямого выбора», которая бы осуществляла сортировку (согласно варианту) :

1) Фамилий студентов по алфавиту.
2) Фамилий студентов по алфавиту в обратном порядке.
3) Студентов по старшинству (начиная со старшего).
4) Студентов по старшинству (начиная с младшего).

5) Студентов по общему баллу (по возрастанию).
6) Студентов по общему баллу (по убыванию).
7) *Студентов по результатам 1-го экзамена (по возрастанию).
8) *Студентов по результатам 2-го экзамена (по убыванию).
9) *Студентов по результатам 3-го экзамена (по возрастанию).
10) *Студентов по результатам 4-го экзамена (по убыванию).
11) Имен в алфавитном порядке.
12) Имен в обратном алфавитном порядке.

11.3 Примерный перечень вопросов

1) В чем состоит суть метода сортировки выбором?
2) Какие шаги выполняет алгоритм сортировки выбором?
3) Как программно реализуется сортировка выбором?
4) В чем достоинства и недостатки метода сортировки выбором?
5) Приведите практический пример сортировки массива методом выбора.

12 Лабораторная работа № 11. Сортировка с помощью прямого обмена и ее модификации

Лабораторная работа №11 предназначена для приобретения навыков написания программ для сортировки методом прямого обмена (пузырьковая сортировка) и ее модификаций на языке программирования Turbo Pascal.

Разработать программы, в соответствии с заданиями в пункте 12.2 и представить отчет в соответствии с приложением В.

Последовательность выполнения лабораторной работы №11 представлена в приложении Б.

Заканчивается работа защитой с оценкой «зачтено» или «незачтено» в виде экспресс-опроса в рамках теоретического и практического материала, отнесенного к пунктам 12.1 и 12.2. Примерный перечень вопросов приведен в пункте 12.3.

На выполнение и защиту лабораторной работы отводится 4 академических часа.

12.1 Краткие теоретические сведения

Пузырьковая сортировка

Идея : (N-1) раз массив проходится снизу вверх (сверху вниз), при этом элементы попарно сравниваются, если нижний элемент меньше верхнего, то элементы переставляются. Каждый проход обеспечивает «всплытие» самого «лёгкого» из оставшихся элементов до тех пор, пока он не займёт своё место (т.е. выше него будут только меньшие). Поэтому этот способ сортировки называют сортировкой методом «пузырька». Идея данного метода сортировки представлена на рисунке 12.1.

Пример – массив содержит значения 4, 3, 7, 2, 1, 6.

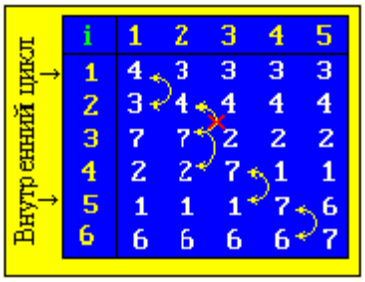


Рисунок 12.1 – Пузырьковая сортировка

Фрагмент программы:

```
for i: = 2 to n do
for j: = n downto i do
if a[j-1] > a[j] then
begin
x: =a[j-1];
a[j-1]: = a[j];
a[j]: =x
end;
```

Среднее количество перемещений:

$$M_{среднее} = \frac{3 \times (n^2 - n)}{2}$$

Количество сравнений:

$$C = \frac{n^2 - n}{2}$$

Преимущества данного метода :

- самый простой алгоритм;
- простой в реализации;
- не нужно дополнительных переменных.

Недостатки:

- долго обрабатывает большие массивы;

– в любом случае количество проходов не уменьшается.

Можно улучшить «пузырьковый» метод, если проходить массив элементов и вверх и вниз одновременно. Данное улучшение реализуется с помощью шейкерной сортировки.

Идея метода шейкерной сортировки заключается в следующем: при просмотре вверх «всплывает» «лёгкий» элемент на своё место, при просмотре вниз «тонет» на своё место «тяжёлый» элемент. Идея метода показана на рисунке 12.2.

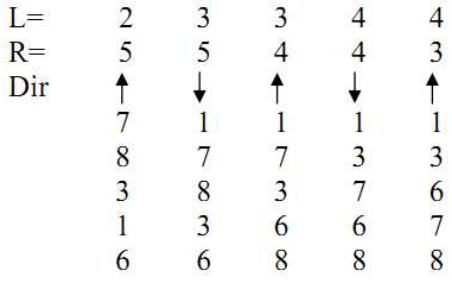


Рисунок 12.2 – Шейкерная сортировка

Процедура шейкерной сортировки:

```
Procedure Shaker_Sort(var a:mass; n:integer);
Var j, k, L, R, x : integer;
Begin
L: =2; R: = n; k: = n;
Repeat
For j:= R downto L do
If f[j – 1] > a[j] then
begin
x: = a[j – 1]; a[j – 1]: =a[j]; a[j]: =x; k: =j
end;
L:= k+1;
For j := L to R do
If a[j – 1] > a[j] then
```

```
begin
x: = a[j – 1]; a[j – 1]: = a[j]; a[j]: = x; k: = j
end;
R: = k – 1;
Until L > R;
End; {конец процедуры}
```

Среднее количество перемещений:

$$M_{среднее} = \frac{3 \times (n^2 - n)}{2}$$

Минимальное количество сравнений:

$$C_{\min} = n - 1$$

Усовершенствованием сортировки методом прямого обмена является быстрая сортировка.

Быстрая сортировка использует стратегию «разделяй и властвуй». Шаги алгоритма таковы:

1) выбираем в массиве некоторый элемент, который будем называть опорным элементом. С точки зрения корректности алгоритма выбор опорного элемента безразличен. С точки зрения повышения эффективности алгоритма выбираться должна медиана, но без дополнительных сведений о сортируемых данных её обычно невозможно получить. Известные стратегии: выбирать постоянно один и тот же элемент, например, средний или последний по положению; выбирать элемент со случайно выбранным индексом;

2) операция разделения массива: реорганизуем массив таким образом, чтобы все элементы, меньшие или равные опорному элементу, оказались слева от него, а все элементы, большие опорного — справа от него. Обычный алгоритм операции:

а) два индекса — l и r, приравниваются к минимальному и максимальному индексу разделяемого массива соответственно;

б) вычисляется индекс опорного элемента m;

в) индекс l последовательно увеличивается до тех пор, пока l-й элемент не превысит опорный;

г) индекс r последовательно уменьшается до тех пор, пока r-й элемент не окажется меньше либо равен опорному;

д) если r = l — найдена середина массива — операция разделения закончена, оба индекса указывают на опорный элемент;

е) если l < r — найденную пару элементов нужно обменять местами и продолжить операцию разделения с тех значений l и r, которые были достигнуты. Следует учесть, что если какая-либо граница (l или r) дошла до опорного элемента, то при обмене значение m изменяется на r-й или l-й элемент соответственно;

3) рекурсивно упорядочиваем подмассивы, лежащие слева и справа от опорного элемента;

4) базой рекурсии являются наборы, состоящие из одного или двух элементов. Первый возвращается в исходном виде, во втором, при необходимости, сортировка сводится к перестановке двух элементов. Все такие отрезки уже упорядочены в процессе разделения.

Пример – массив содержит следующие значения 9,4,15,10,3,18,16,12. Алгоритм сортировки показан на рисунке 12.3.

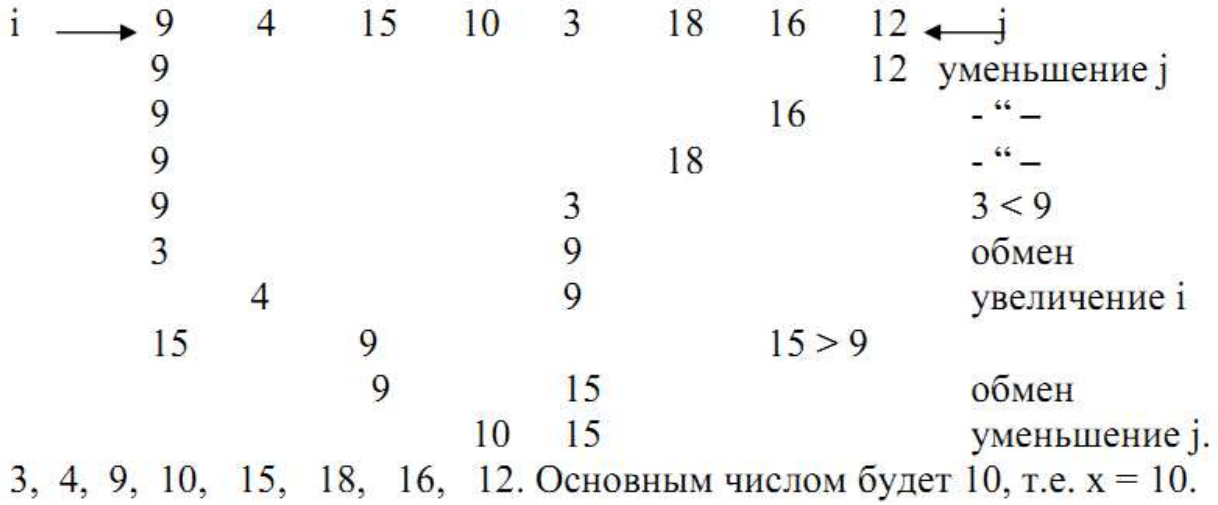


Рисунок 12.3 – Быстрая сортировка

Процедура быстрой сортировки:

```
Procedure Sort(L, r: integer; var a: mass);
Var i, j, x, y: integer;
Begin if L < r then {условие продолжения рекурсивных вызовов}
begin i: =L; j: = r; x: = a[i];
repeat
while x < a[j] do j: = j - 1; {уменьшаем j}
if i <= j then
begin y: = a[i]; a[i]: =a[j]; a[j]: =y; {обмен}
i: = i +1
end;
while x > a[i] do i: = i + 1;    {увеличиваем i}
if i <= j then
begin y: = a[i]; a[i]: = a[j]; a[j]: = y;  {обмен}
j: = j -1
end;
until i > j;
Sort(L, j, a); {разделяем левую часть}
Sort(i, r, a); {разделяем правую часть}
end
end
End;
```

Число перестановок и сравнений: (n* log(n)).

12.2 Задания на лабораторную работу

1) Составить программу вывода на экран самого большого (самого малого) элемента массива A.

2) Составить программу сортировки массива A по убыванию величин его элементов.

3) В массиве A находятся элементы. Составить программу, которая сформирует массив B, в котором расположить элементы массива B в порядке убывания.

4) *Дан упорядоченный массив A - числа, расположенные в порядке возрастания, и число a, которое необходимо вставить в массив A, так, чтобы упорядоченность массива сохранилась.

5) *Дан массив A, содержащий как отрицательные, так и положительные числа. Составить программу исключения из него всех отрицательных чисел, а оставшиеся положительные расположить в порядке их возрастания.

6) *Дан список авторов в форме массива A. Составить программу формирования указателя авторов в алфавитном порядке и вывести его на экран.

7) * Имеется n абонентов телефонной станции. Составить программу, в которой формируется список по форме: номер телефона, фамилия (номера идут в порядке возрастания).

12.3 Примерный перечень вопросов

1) В чем состоит суть метода сортировки обменом?
2) Какие шаги выполняет алгоритм сортировки обменом?
3) Как программно реализуется сортировка обменом?
4) В чем достоинства и недостатки метода сортировки обменом?
5) Приведите практический пример сортировки массива методом обмена.
6) В чем состоит суть метода быстрой сортировки?
7) За счет чего метод быстрой сортировки дает лучшие показатели по сравнению с простейшими методами?
8) Что такое опорный элемент в методе быстрой сортировки и как он используется?
9) Приведите практический пример быстрой сортировки массива.

10) Что можно сказать о применимости метода быстрой сортировки с точки зрения его эффективности?

11) Какой фактор оказывает решающее влияние на эффективность метода быстрой сортировки?

12) Почему выбор серединного элемента в качестве опорного в методе быстрой сортировки может резко ухудшать эффективность метода?

13) Какое правило выбора опорного элемента в методе быстрой сортировки является наилучшим и почему его сложно использовать?

14) Какое простое правило выбора опорного элемента в методе быстрой сортировки рекомендуется использовать на практике?

15) Какие усовершенствования имеет базовый алгоритм метода быстрой сортировки?

16) Почему быстрая сортировка проще всего программно реализуется с помощью рекурсии?

17) Как программно реализуется рекурсивный вариант метода быстрой сортировки?

18) Какие особенности имеет не рекурсивная программная реализация метода быстрой сортировки?

13 Лабораторная работа № 12. Сортировка с помощью дерева

Лабораторная работа №12 предназначена для приобретения навыков написания программ для сортировки с помощью дерева на языке программирования Turbo Pascal.

Разработать программы, в соответствии с заданиями в пункте 13.2 и представить отчет в соответствии с приложением В.

Последовательность выполнения лабораторной работы №12 представлена в приложении Б.

Заканчивается работа защитой с оценкой «зачтено» или «незачтено» в виде экспресс-опроса в рамках теоретического и практического материала, отнесенного к пунктам 13.1 и 13.2. Примерный перечень вопросов приведен в пункте 13.3.

На выполнение и защиту лабораторной работы отводится 4 академических часа.

13.1 Краткие теоретические сведения

Сортировка с помощью дерева или пирамидальная сортировка была предложена Дж. Уильямсом в 1964 году. Это алгоритм сортировки массива произвольных элементов; требуемый им дополнительный объём памяти не зависит от количества исходных данных. Время работы алгоритма — $O(n \cdot \ln n)$ в среднем, а также в лучшем и худшем случаях.

Пирамидальная сортировка является улучшенным вариантом сортировки выбором, в которой на каждом шаге должен определяться наименьший элемент в необработанном наборе данных. Поиск наименьшего элемента можно совместить с выполнением некоторых дополнительных действий, облегчающих поиск на последующих шагах.

Идея алгоритма.

Для того, чтобы прояснить всё дальнейшее изложение, в двух словах опишем идею алгоритма.

– пирамида — двоичное дерево, в котором значение каждого элемента больше (меньше) либо равно значений дочерних элементов;

– заполнив дерево элементами в произвольном порядке, можно легко его отсортировать, превратив в пирамиду;

– самый большой (маленький) элемент пирамиды находится в её вершине;

– отделяем вершинный элемент, и записываем его в конец результирующего массива;

– на место вершинного элемента записываем элемент из самого нижнего уровня дерева;

– восстанавливаем (пересортировываем) пирамиду;

– самый большой (маленький) элемент из оставшихся снова в вершине. Снова отделяем его и записываем его в качестве предпоследнего элемента результата, и так далее;

– весь фокус алгоритма в том, что пирамида без дополнительных затрат хранится прямо в исходном массиве. По мере того, как размер пирамиды уменьшается, она занимает всё меньшую часть массива, а результат сортировки записывается, начиная с конца массива на освободившиеся от пирамиды места.

Объяснение пирамидальной сортировки.

Пусть имеется исходный массив n элементов a_1 a_2, a_3, . . ., a_n. Расположим эти элементы в виде двоичного дерева следующего вида (здесь важен порядок следования индексов элементов), как показано на рисунке 13.1:

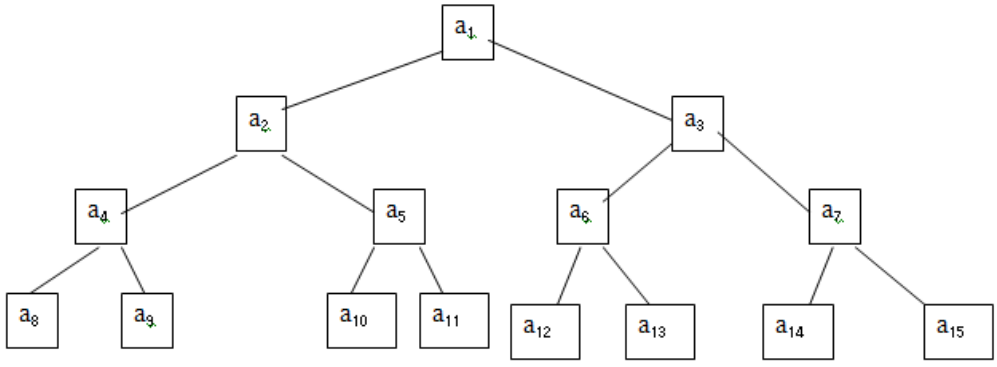


Рисунок 13.1 – Пример двоичного дерева

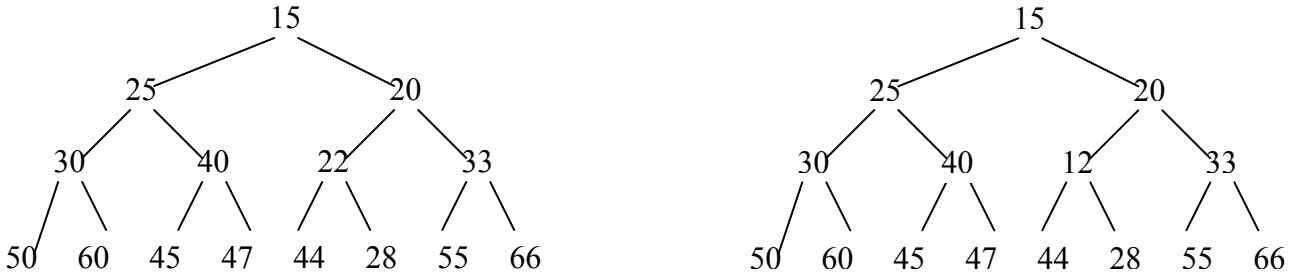
Подобное дерево называется минимальной пирамидой, если для всех элементов с индексами от 1 до n/2 выполняются следующие условия:

$$a_i <= a_{2i} \quad и \quad a_i <= a_{2i+1}$$

В частности, эти условия означают: $a_1 <= a_2$ и $a_1 <= a_3$; $a_2 <= a_4$ и $a_2 <= a_5$; $a_3 <= a_6$ и $a_3 <= a_7$; $a_4 <= a_8$ и $a_4 <= a_9$, и т.д.

Другими словами, в каждом элементарном поддереве значение в вершине этого поддерева меньше или равно значений в вершинах-потомках.

Пример двоичного дерева-пирамиды с 15-ю элементами показан на рисунке 13.2:



Это – пирамида. Соответствующий массив: 15-25-20-30-40-22-33-50-60-45-47-44-28-55-66

Это – не пирамида (вершина 12 меньше 20)

Рисунок 13.2 – Отличие двоичного дерева от дерева-пирамиды

Такие пирамиды иногда называют минимальными, в отличие от максимальных, где правило упорядоченности прямо противоположное.

Из примера видно, что пирамида не является деревом поиска, т.к. строится по другим правилам.

Можно отметить следующие полезные свойства пирамиды:

– на вершине пирамиды всегда находится наименьший элемент во всем массиве (элемент a_1), элемент a_2 является наименьшим для левого поддерева, элемент a_3 является наименьшим для правого поддерева и т.д.;

– вершины, лежащие на самом нижнем уровне пирамиды, всегда образуют из себя элементарные пирамиды, поскольку у них нет никаких потомков и их сравнивать не с кем.

Пирамидальная сортировка включает два больших этапа:

– представление исходного массива в виде пирамиды;

– последовательные удаления минимального элемента с вершины пирамиды с заменой его другим элементом.

Реализация 1 этапа включает следующие шаги:

– условно разделяем исходный массив на две половины: левую с индексами от 1 до [n/2], и правую с индексами от [n/2]+1 до n (здесь [ ] обозначает целую часть); считаем пока, что левая половина образует верхнюю часть строящейся пирамиды, а правая – нижний слой терминальных вершин;

– выбираем в левой половине последний элемент (его индекс m=[n/2]), а в правой половине – его непосредственных потомков (одного или двух, но хотя бы один будет обязательно) с индексом 2m и, возможно, 2m+1;

– если потомков два, то выбираем из них наименьшего;

– сравниваем элемент a_m с наименьшим из потомков: если он больше, то меняем эти элементы в массиве местами для получения фрагмента пирамиды; в противном случае оставляем все без изменений, поскольку эти элементы уже образуют фрагмент пирамиды;

– повторяем все описанные действия последовательно для оставшихся в левой части элементов справа налево, т.е. для a_{m-1}, a_{m-2}, a_{m-3}, . . ., a_1, при этом если происходит обмен между родительской вершиной и одним из потомков, выполняется проверка для новой вершины-потомка, т.к. она может иметь своих потомков, с которыми возможно потребуется ее обменять для выполнения условия пирамиды.

Тем самым, для каждого элемента массива находится его новое расположение в массиве таким образом, чтобы выполнялись условия пирамиды. Процесс определения нужного положения элемента в массиве-пирамиде называют просеиванием элемента через пирамиду. Построение пирамиды заканчивается после просеивания первого элемента a_1. Пример для 15 элементов приведен в таблице 13.1 (символ ~ обозначает перестановку элементов).

Таблица 13.1 – Пример просеивания элемента в массиве-пирамиде

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	сравнение и обмен
45	40	28	25	30	44	33	22	60	15	55	47	66	20	50	33> min(20, 50), 33~20
45	40	28	25	30	44	20	22	60	15	55	47	66	33	50	44< min(47, 66), нет
45	40	28	25	30	44	20	22	60	15	55	47	66	33	50	30> min(15, 55), 30~15
45	40	28	25	15	44	20	22	60	30	55	47	66	33	50	25> min(22, 60), 25~22
45	40	28	22	15	44	20	25	60	30	55	47	66	33	50	28>min(44, 20), 28~20
45	40	20	22	15	44	28	25	60	30	55	47	66	33	50	28<min(33, 50), нет
45	40	20	22	15	44	28	25	60	30	55	47	66	33	50	40>min(22, 15), 40~15
45	15	20	22	40	44	28	25	60	30	55	47	66	33	50	40>min(30, 55), 40~30
45	15	20	22	30	44	28	25	60	40	55	47	66	33	50	45>min(15, 20), 45~15
15	45	20	22	30	44	28	25	60	40	55	47	66	33	50	45>min(22, 30), 45~22
15	22	20	45	30	44	28	25	60	40	55	47	66	33	50	45>min(25, 60), 45~25
15	22	20	25	30	44	28	45	60	40	55	47	66	33	50	Пирамида построена

В худшем случае каждый шаг в просеивании очередного элемента требует двух сравнений: сначала сравниваются два потомка текущего элемента, а потом наименьший из них сравнивается с самим элементом. В примере для построения пирамиды потребовалось 11*2=22 сравнения и 9 пересылок.

Реализация второго этапа состоит в (n-1)-кратном повторении следующих действий:

– с вершины пирамиды забирается минимальный элемент a_1;

– на его место в вершину пирамиды помещается последний элемент в массиве, причем индекс этого последнего элемента на каждом шаге будет уменьшаться от a_n до a_2;

– помещенный в вершину элемент просеивается через пирамиду обычным образом, при этом он встает на свое место, а в вершину пирамиды выталкивается минимальный из оставшихся в массиве элементов;

– на последнем шаге в пирамиде останется один элемент (самый большой) и сортировка будет закончена.

При этом возникает вопрос – куда девать снимаемые с вершины пирамиды элементы? Можно просто помещать их в конец массива на место элемента, размещаемого в вершине. В результате на месте исходного массива создается упорядоченный по убыванию набор данных. При необходимости, алгоритм легко можно изменить для получения возрастающих последовательностей, если вместо минимальных использовать максимальные пирамиды.

Таблица 13.2 – Пример последовательного удаления минимального элемента с вершины пирамиды с заменой его другим элементом

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	сравнение и обмен
15	22	20	25	30	44	28	45	60	40	55	47	66	33	50	15 = min, 15~50
50	22	20	25	30	44	28	45	60	40	55	47	66	33	**15**	50>min(22, 20), 50~20
20	22	50	25	30	44	28	45	60	40	55	47	66	33	**15**	50>min(44, 28), 50~28
20	22	28	25	30	44	50	45	60	40	55	47	66	33	**15**	50>33, 50~33
20	22	28	25	30	44	33	45	60	40	55	47	66	50	**15**	20 = min, 20~50
50	22	28	25	30	44	33	45	60	40	55	47	66	**20**	**15**	50>min(22, 28), 50~22
22	50	28	25	30	44	33	45	60	40	55	47	66	**20**	**15**	50>min(25, 30), 50~25
22	25	28	50	30	44	33	45	60	40	55	47	66	**20**	**15**	50>min(45, 60), 50~45
22	25	28	45	30	44	33	50	60	40	55	47	66	**20**	**15**	22 = min, 22~66
66	25	28	45	30	44	33	50	60	40	55	47	**22**	**20**	**15**	66>min(25, 28), 66~25
25	66	28	45	30	44	33	50	60	40	55	47	**22**	**20**	**15**	66>min(45, 30), 66~30
25	30	28	45	66	44	33	50	60	40	55	47	**22**	**20**	**15**	66>min(40, 55), 66~40
25	30	28	45	40	44	33	50	60	66	55	47	**22**	**20**	**15**	25 = min, 25~47
47	30	28	45	40	44	33	50	60	66	55	**25**	**22**	**20**	**15**	47>min(30, 28), 47~28
28	30	47	45	40	44	33	50	60	66	55	**25**	**22**	**20**	**15**	47>min(44, 33), 47~33
28	30	33	45	40	44	47	50	60	66	55	**25**	**22**	**20**	**15**	28 = min, 28~55
55	30	33	45	40	44	47	50	60	66	**28**	**25**	**22**	**20**	**15**	55>min(30, 33), 55~30
30	55	33	45	40	44	47	50	60	66	**28**	**25**	**22**	**20**	**15**	55>min(45, 40), 55~40

Продолжение таблицы 13.2

30	40	33	45	55	44	47	50	60	66	**28**	**25**	**22**	**20**	**15**	55<66, 30 = min, 30~66
66	40	33	45	55	44	47	50	60	**30**	**28**	**25**	**22**	**20**	**15**	66>min(40, 33), 66~33
33	40	66	45	55	44	47	50	60	**30**	**28**	**25**	**22**	**20**	**15**	66>min(44, 47), 66~44
33	40	44	45	55	66	47	50	60	**30**	**28**	**25**	**22**	**20**	**15**	33 = min, 33~60
60	40	44	45	55	66	47	50	**33**	**30**	**28**	**25**	**22**	**20**	**15**	60>min(40, 44), 60~40
40	60	44	45	55	66	47	50	**33**	**30**	**28**	**25**	**22**	**20**	**15**	60>min(45, 55), 60~45
40	45	44	60	55	66	47	50	**33**	**30**	**28**	**25**	**22**	**20**	**15**	60>50, 60~50
40	45	44	50	55	66	47	60	**33**	**30**	**28**	**25**	**22**	**20**	**15**	40 = min, 40~60
60	45	44	50	55	66	47	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	60>min(45, 44), 60~44
44	45	60	50	55	66	47	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	60>min(66, 47), 60~47
44	45	47	50	55	66	60	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	44 = min, 44~60
60	45	47	50	55	66	**44**	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	60>min(45, 47), 60~45
45	60	47	50	55	66	**44**	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	60>min(50, 55), 60~50
45	50	47	60	55	66	**44**	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	45 = min, 45~66
66	50	47	60	55	**45**	**44**	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	66>min(50, 47), 66~47
47	50	66	60	55	**45**	**44**	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	47 = min, 47~55
55	50	66	60	**47**	**45**	**44**	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	55>min(50, 66), 55~50
50	55	66	60	**47**	**45**	**44**	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	55<60, 50 = min, 50~60
60	55	66	**50**	**47**	**45**	**44**	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	60>min(55, 66), 60~55
55	60	66	**50**	**47**	**45**	**44**	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	55 = min, 55~66
66	60	**55**	**50**	**47**	**45**	**44**	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	66>60, 66~60
60	66	**55**	**50**	**47**	**45**	**44**	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	60 = min, 60~66
66	**60**	**55**	**50**	**47**	**45**	**44**	**40**	**33**	**30**	**28**	**25**	**22**	**20**	**15**	сортировка закончена

В данном примере выполнено 51 сравнение и 40 пересылок, что вместе с этапом построения пирамиды дает 73 сравнения и 49 пересылок.

В целом, данный метод с точки зрения трудоемкости имеет типичное для улучшенных методов поведение: довольно высокая трудоемкость для небольших n, но с ростом n эффективность метода растет. При больших n метод в среднем немного уступает быстрой сортировке и имеет оценку порядка $(n*\log_2 n)/2$. Единственное, в чем он превосходит быструю сортировку, так это поведение на

аномальных входных данных, когда быстрая сортировка перестает быть «быстрой», а пирамидальная сохраняет свою трудоемкость порядка $O(n*\log_2 n)$. Пирамидальную сортировку выгодно использовать в том случае, когда требуется не провести(проводить) полную сортировку большого набора данных, а лишь найти несколько (причем – немного) первых наименьших элементов.

Необходимо отметить, что понятие пирамидального дерева имеет и самостоятельное значение, и часто используется для решения других задач, не связанных с сортировкой массивов, например – для эффективной реализации приоритетных очередей.

В заключение рассмотрим вопросы программной реализации пирамидальной сортировки. Поскольку оба этапа алгоритма основаны на повторении одинаковых действий по просеиванию элементов, удобно оформить просеивание в виде вспомогательной процедуры.

Приведем процедуру просеивания элемента в массиве пирамиде:

```
Procedure Sift (var a: mass; L,R: integer); {L- номер элемента, включаемого в пирамиду; R– номер последнего элемента массива}
Var  i, j, x: integer;
Begin i: = L; j: = 2*L; x: = a[L];
if (j<R) and (a[j+1]<a[j]) then j:= j+1; {j<R - контролирует выход за пределы массива; определяет меньший потомок}
while (j< R) and (x >a[j]) do {обмен}
begin
x: = a[i]; a[i]: = a[j]; a[j]: =x; i: = j; j: = 2*j; {перенос, определение нового узла}
if (l< R) and (a[j+1< a[j]) then j: =j+1;
end
End;   {sift}
```

Приведем пример пирамидальной сортировки:

```
Procedure  HeapSort (var a: mass; n: integer);
Var L, R, x: integer;
Begin L: = (n div 2)+1; R: = n; {определяем место элемента, у которого нет потомков и номер последнего просматриваемого элемента}
While L >1 do
begin L: =L-1; Sift (a, L, R) end; {делаем пирамиду}
While R>1 do
begin x: = a[1]; a[1]: = a[R]; a[R]: = x;
R: = R-1; Sift(a,1,R); end
End; {HeapSort}

Begin {основная программа}
Clrscr; randomize;
write('введи количество элементов массива '); readln(k);
For i: = 1 to k do
begin a[i]: = random(100); write(a[i],' ') end;
HeapSort(a,k);
writeln('отсортированный массив по возрасанию: ');
for j: = k downto 1 do write(a[j],' ');
End.
```

13.2 Задания на лабораторную работу

1) На заводе выпустили детали со следующими серийными номерами : 45, 56, 13, 75, 14, 18, 43, 11, 52, 12, 10, 36, 47, 9. Детали с четными номерами поступают на склад №1, а с нечетными на слад №2. Требуется отсортировать детали на складе №1.

2) Угнали автомобиль. Свидетель запомнил, что первой цифрой номера была 4. В базе угнанных автомобилей в этот день были следующие номера : 456, 124, 786, 435, 788, 444, 565, 127, 458, 322, 411, 531, 400, 546, 410. Нужно составить список номеров начинающихся на 4 и упорядочить его по возрастанию.

3) За неделю езды в транспорте накопились билеты с номерами 124512, 342351, 765891, 453122, 431350, 876432, 734626, 238651, 455734, 234987. Нужно отобрать «счастливые» билеты и расположить их по возрастанию.

4) Дан список людей с указанием их возраста. Для составления графика ухода сотрудников на пенсию требуется составить новый список в том порядке, в каком они будут уходить на пенсию.

5) Студенты сдали пять экзаменов. Нужно отсортировать список студентов по возрастанию общего балла по результатам сданных экзаменов.

6) В городе был один автобусный парк, куда приезжали автобусы с номерами: 11, 32, 23, 12, 6, 52, 47, 63, 69, 50, 43, 28, 35, 33, 42, 56, 55, 101. После строительства второго автопарка решили перевести туда автобусы с нечетными номерами. Для того чтобы составить расписание их движения нужно организовать список номеров автобусов второго парка, упорядочив их по убыванию.

7) Была составлена ведомость по зарплате, представленная в виде : Иванов - 166000, Сидоров - 180000, ... Требуется упорядочить этот список таким образом, чтобы размер зарплаты уменьшался.

8) На стоянке стоят автомобили со следующими номерами : 1212, 3451, 7694, 4512, 4352, 8732, 7326, 2350, 4536, 2387, 5746, 6776, 4316, 1324. Для статистики необходимо составить список автомобилей с такими номерами, сумма первых двух цифр которых равна сумме двух последних цифр, так чтобы каждый следующий номер был меньше предыдущего.

9) Выпустили лотерейные билеты с четырехзначными номерами. Выигрышными считаются те билеты, сумма цифр которых делится на 4. Составить список выигрышных билетов, упорядоченных по убыванию.

10) Молодой человек взял номер телефона у своей знакомой, но забыл его. Он смог вспомнить только первые три цифры : 469***. В его записной книжке были следующие номера телефонов : 456765, 469465, 469321, 616312, 576567, 469563, 567564, 469129, 675665, 469873, 569090, 469999, 564321, 469010. Составить список номеров начинающихся с цифр 469 и упорядочить их по убыванию.

11) Студенты сдали пять экзаменов. Нужно отсортировать список студентов по убыванию общего балла по результатам сданных экзаменов.

12) Выпустили лотерейные билеты с четырехзначными номерами. Выигрышными считаются те билеты, сумма первых трех цифр которых равна 8. Составить список выигрышных билетов, упорядоченных по возрастанию.

13.3 Примерный перечень вопросов

1) В чем состоит суть метода пирамидальной сортировки?
2) Какой набор данных имеет пирамидальную организацию?
3) Чем отличаются друг от друга дерево поиска и пирамидальное дерево?
4) Приведите пример пирамидального дерева с целочисленными ключами.
5) Какие полезные свойства имеет пирамидальное дерево?
6) Какие шаги выполняются при построении пирамидального дерева?
7) Что такое просеивание элемента через пирамиду?
8) Приведите практический пример построения пирамидального дерева.
9) Какие шаги выполняются на втором этапе пирамидальной сортировки?
10) Приведите практический пример реализации второго этапа пирамидальной сортировки.
11) Что можно сказать о трудоемкости метода пирамидальной сортировки?

14 Лабораторная работа № 13. Исследование методов линейного и бинарного поиска

Лабораторная работа №13 предназначена для приобретения навыков написания программ для линейного и бинарного поиска на языке программирования Turbo Pascal.

Разработать программы, в соответствии с заданиями в пункте 14.2 и представить отчет в соответствии с приложением В.

Последовательность выполнения лабораторной работы №13 представлена в приложении Б.

Заканчивается работа защитой с оценкой «зачтено» или «незачтено» в виде экспресс-опроса в рамках теоретического и практического материала, отнесенного к пунктам 14.1 и 14.2. Примерный перечень вопросов приведен в пункте 14.3.

На выполнение и защиту лабораторной работы отводится 4 академических часа.

14.1 Краткие теоретические сведения

Поиск – процесс отыскания информации во множестве данных (обычно представляющих собой записи) путем просмотра специального поля в каждой записи, называемого ключом. Целью поиска является отыскание записи (если она есть) с данным значением ключа.

Поиск – одно из наиболее часто встречающихся в программировании действий. Существует множество различных алгоритмов этого действия, принципиально зависящих от способа организации данных. Далее рассмотрены алгоритмы линейного и бинарного поиска.

При дальнейшем рассмотрении поиска в линейных структурах определим, что множество данных, в котором производится поиск, описывается как массив фиксированной длины:

```
A: array[1..n] of ItemType;
```

Обычно тип ItemType описывает запись с некоторым полем, играющим роль ключа, а сам массив представляет собой таблицу. Так как здесь рассматривается, прежде всего, сам процесс поиска, то будем считать, что тип ItemType включает только ключ.

Линейный (последовательный) поиск.

Последовательный поиск – самый простой из известных. Суть его заключается в следующем. Множество элементов просматривается последовательно в некотором порядке, гарантирующем, что будут просмотрены все элементы множества (например, слева направо). Если в ходе просмотра множества будет найден искомый элемент, просмотр прекращается с положительным результатом; если же будет просмотрено все множество, а элемент не будет найден, алгоритм должен выдать отрицательный результат.

Именно так поступает человек, когда ищет что-то в неупорядоченном множестве. Таким образом, данный алгоритм можно применять тогда, когда нет никакой дополнительной информации о расположении данных в рассматриваемой последовательности.

Оформим описанный алгоритм в виде функции на Паскале.

```
function LineSearch(Key: ItemType, n: integer; var A: array[1..n] of ItemType): boolean; {Функция линейного поиска, если элемент найден, то возвращает значение true, иначе – false}
  var
    i: integer;
  begin
    i := 1;
    while (i <=n ) and (A[i]<>Key) do i := i + 1;
    if A[i]=Key then LineSearch := true
                else LineSearch := false;
```

```
end;
```

В среднем этот алгоритм требует n/2 итераций цикла. Это означает временную сложность алгоритма поиска, пропорциональную O(n). Никаких ограничений на порядок элементов в массиве данный алгоритм не накладывает. При наличии в массиве нескольких элементов со значением Key алгоритм находит только первый из них (с наименьшим индексом).

Здесь можно хранить множество как в массиве (как показано выше), так и в обычном однонаправленном списке.

Существует модификация алгоритма последовательного поиска, которая ускоряет поиск. Эта модификация поиска является небольшим усовершенствованием предыдущего. В любой программе, имеющей циклы, наибольший интерес представляет оптимизация именно циклов, т. е. сокращение числа действий в них. Посмотрим на алгоритм последовательного поиска. В цикле while производятся два сравнения: (i<=n) и (A[i]<>Key). Избавимся от одного из них (от первого), введя в массив так называемый «барьер», положив A[n+1] := Key. В этом случае в цикле обязательно будет найден элемент со значением Key.

После завершения цикла требуется дополнительная проверка того, был ли найден искомый элемент, или «барьер».

Тогда функция поиска будет выглядеть так:

```
function LineSearchWithBarrier (Key: ItemType, n: integer; var A: array[1..n+1] of ItemType): boolean;
{Функция линейного поиска с барьером, если элемент найден, то возвращает значение true, иначе – false}
var
  i: integer;
begin
  I := 1;
  A[n+1] := Key;
  while A[i]<>Key do I := I + 1;
```

```
    if I <= n then LineSearchWithBarrier := true
              else LineSearchWithBarrier := false;
end;
```

Надо сказать, что хотя такая функция будет работать быстрее, но временная сложность алгоритма остается такой же – O(n). Гораздо больший интерес представляют методы, не только работающие быстро, но и реализующие алгоритмы с меньшей сложностью. Один из таких методов – бинарный поиск.

Бинарный поиск.

Этот алгоритм поиска предполагает, что множество хранится, как некоторая упорядоченная (например, по возрастанию) последовательность элементов, к которым можно получить прямой доступ посредством индекса. Фактически речь идет о том, что множество хранится в массиве и этот массив отсортирован.

Суть метода заключается в следующем. Областью поиска (L, R) назовем часть массива с индексами от L до R, в которой предположительно находится искомый элемент. Сначала областью поиска будет часть массива (L, R), где L = 1, а R = n, т. е. вся заполненная элементами множества часть массива. Теперь найдем индекс среднего элемента m := (L+R) div 2. Если Key > Am, то можно утверждать (поскольку массив отсортирован), что если Key есть в массиве, то он находится в одном из элементов с индексами от L + m до R, следовательно, можно присвоить L := m + 1, сократив область поиска. В противном случае можно положить R := m. На этом заканчивается первый шаг метода. Остальные шаги аналогичны, которые показаны на рисунке 14.1.

На каждом шаге метода область поиска будет сокращаться вдвое. Как только L станет равно R, т. е. область поиска сократится до одного элемента, можно будет проверить этот элемент на равенство искомому и сделать вывод о результате поиска.

Оформим описанный алгоритм в виде функции на Паскале.

```
function BinarySearch(Key: ItemType, n: integer; var A: array[1..n] of ItemType): boolean; {Функция двоичного поиска, если элемент найден, то возвращает значение true, иначе – false}
var
  L, m, R: integer;
begin
  L := 1; R := n;
  while (L <> R) do begin
    m := (L+R) div 2;
    if Key > A[m] then L := m+1
                  else R := m;
  end;
  if A[L]= Key then BinarySearch := true
              else BinarySearch := false;
end;
```

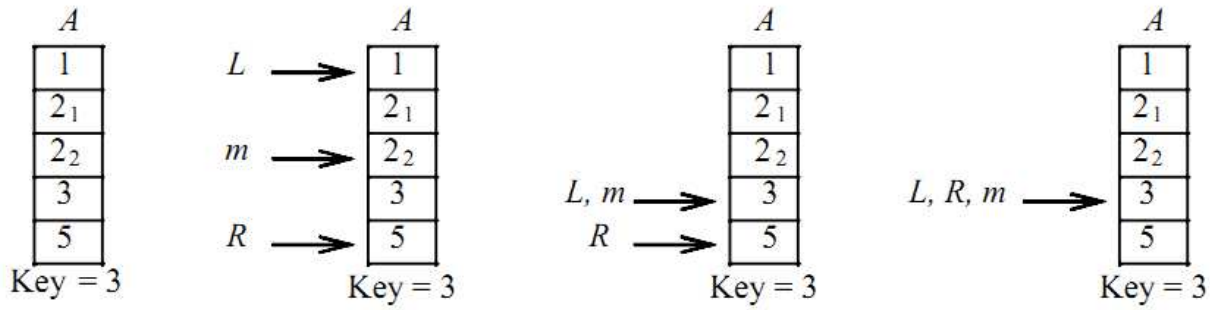


Рисунок 14.1 – Пример бинарного поиска

Как уже было сказано, область поиска на каждом шаге сокращается вдвое, а это означает сложность алгоритма пропорциональную O(log n).

14.2 Задания на лабораторную работу

1) Найти наименьший элемент в массиве A с помощью линейного поиска.
2) Поиск элементов в массиве A, которые больше 30.

3) Вывести на экран все числа массива A кратные 3 (3,6,9,...) с помощью линейного поиска.

4) Найти все элементы, модуль которых больше 20 и меньше 50, с помощью линейного поиска.

5) Вывести на экран все числа массива A кратные 4 (4,8,...) с помощью линейного поиска.

6) *Вывести на экран сообщение, каких чисел больше относительно 50, с помощью линейного поиска.

7) *Найти элемент в массиве A и найти число сравнений с помощью линейного поиска.

8) Поиск элементов случайным образом с помощью бинарного поиска.

9) *Дан список номеров машин (345, 368, 876, 945, 564, 387, 230), найти, на каком месте стоит машина с заданным номером, бинарный поиск.

10) *Поиск каждого второго элемента в списке и число сравнений.

11) *Найти элемент с заданным ключом с помощью бинарного поиска.

14.3 Примерный перечень вопросов

1) В чем состоит назначение поиска?

2) Что такое уникальный ключ?

3) Какая операция производится в случае отсутствия заданного ключа в списке?

4) В чем разница между последовательным и бинарным поиском?

5) Какой из них более эффективный и почему?

6) В чем суть последовательного поиска?

7) Приведите программную реализацию последовательного поиска.

8) Приведите практический пример последовательного поиска.

9) В чем суть бинарного поиска?

10) Приведите программную реализацию бинарного поиска.

11) Приведите практический пример бинарного поиска.

15 Примерные темы курсовых работ

1. Хеширование. Реализовать программу «Студенты и стипендия». Требования к программе: удобный интерфейс и защита от неправильных действий пользователя. Данные должны храниться в отдельных файлах.

2. Хеширование. Реализовать программу «Склад». Требования к программе: удобный интерфейс и защита от неправильных действий пользователя Данные должны храниться в отдельных файлах.

3. Хеширование. Реализовать программу «Телефонный справочник». Требования к программе: удобный интерфейс и защита от неправильных действий пользователя. Данные должны храниться в отдельных файлах.

4. Хеширование. Реализовать программу «Библиотека». Требования к программе: удобный интерфейс и защита от неправильных действий пользователя. Данные должны храниться в отдельных файлах.

5. Задача о составлении магического квадрата. То есть такого квадрата, у которого суммы столбцов равны между собой, равны суммам строк, и суммам диагоналей. При этом числа в квадрате порядка n идут от 1 до n*n. Очень интересная задача - в данном случае пользователь вводит порядок квадрата и получает этот самый квадрат. Заодно и сумма высчитывается.

6. Деревья. Реализовать операции с красно-черными деревьями. Требования к программе: удобный интерфейс и защита от неправильных действий пользователя.

7. Деревья. Реализовать операции с АВЛ – деревьями. Требования к программе: удобный интерфейс и защита от неправильных действий пользователя.

8. Деревья. Реализовать операции с B-деревьями. Требования к программе: удобный интерфейс и защита от неправильных действий пользователя.

9. Деревья. Реализовать операции с 2-3 деревьями. Требования к программе: удобный интерфейс и защита от неправильных действий пользователя.

10. Графы. Реализовать поиск точек сочленения в графе.

11. Графы. Максимальный поток минимальной стоимости. Алгоритм Клейна.

12. Графы. Максимальный поток минимальной стоимости. Алгоритм Басакера-Гоуэна.

13. Графы. Метод минимакса на двоичном дереве. Предположим, два игрока играют в игру, такую, что в каждой позиции игрок, чья очередь хода, имеет только два варианта ответа. Такую игру можно представить в виде двоичного дерева. Предположим, что игрок принимающий решение о ходе способен провести анализ на N ходов вглубь и предположим, что он в состоянии оценить конечные позиции числом тем большим, чем лучше эта конечная позиция. Вопрос: какой из двух имеющихся вариантов игры ему выбрать при условии, что противник играет наилучшим образом и имеет точно такие же критерии оценки позиций.

14. Графы. Поиск пути к бензоколонке. Некий автомобилист пытается в городе найти путь к бензоколонке. Он находится в определённой точке, и он располагает картой города, на которой отмечена единственная бензоколонка. На улицах города, как и полагается, установлены знаки, соблюдать которые водитель в принципе обязан. Но наш водитель имеет возможность нарушить некоторое количество правил. Кроме того, его автомобиль имеет некоторое количество топлива, которого возможно более чем достаточно, но может быть и не очень много. Необходимо найти путь, на котором водитель уложится в существующее количество топлива и минимальное количество раз нарушит правила дорожного движения. Экономить топливо не обязательно. Карта города представляет собой граф, для построения которого обязательно использовать указатели.

15. Графы. Задача о назначениях. Венгерский алгоритм.

16. Графы. Реализовать приближенные методы решения задачи коммивояжера. Алгоритм Кристофидеса. Алгоритм Эйлера. Метод локальной оптимизации.

17. Графы. Задача китайского почтальона.

18. Графы. Правильной вершинной раскраской неориентированного графа называется функция $C: V \to N$, удовлетворяющая условию

$$\forall (i, j) \in E: \; C(i) \neq C(j)$$

– То есть смежные вершины должны иметь разные цвета.

– Необходимо найти раскраску графа в минимальное количество цветов. Пример – составление расписания экзаменов.

19. Графы. Задача о кратчайших путях. Реализовать алгоритм Форда-Беллмана, волновой алгоритм, алгоритм Флойда, алгоритм Флойда-Уоршалла

20. Графы. Задача нахождения максимального потока. Алгоритм Форда-Фалкерсона, Алгоритм Эдмондса-Карпа и один алгоритм на выбор.

21. Пирамида. Реализовать способы реализации и основные операции. Требования к программе: удобный интерфейс и защита от неправильных действий пользователя.

Список использованных источников

1 Хусаинов, Б. С. Структуры и алгоритмы обработки данных. Примеры на языке Си: учеб. пособие / Б. С. Хусаинов. – М. : Финансы и статистика, 2004. - 464 с. : ил. + 1 электрон. диск (CD-ROM). - Библиогр.: с. 462-464. - ISBN 5-279-02775-8.

2 Кнут, Д. Э. Искусство программирования: пер. с англ. / Д. Э. Кнут . - 3-е изд. - Москва : Вильямс, 2007. - Т. 1 : Основные алгоритмы, 2007. - 720 с. : ил.. - Прил.: с. 683-691. - Предм.-имен. указ.: с. 692-712. - ISBN 5-8459-0080-8.

3 Кнут, Д. Э. Искусство программирования / Д. Э. Кнут ; под общ. ред. Ю. В. Козаченко. - 2-е изд. - М. : Вильямс, 2009. - (Классический труд : Исправленное и дополненное издание). Т. 3 : Сортировка и поиск. - , 2009. - 823 с. : ил.. - Прил.: с. 794-803. - . - Предм.-имен. указ.: с. 804-822 - ISBN 978-5-8459-0082-1.

4 Кормен, Т. Алгоритмы: Построение и анализ / Т. Кормен, Ч. Лейхерсон, Р. Риверст. – М.: МЦМНО, 2002.

5 Давыдов, В. Г. Программирование и основы алгоритмизации: учеб. пособие для вузов / В. Г. Давыдов. - М. : Высш. шк., 2003. - 447 с. : ил - ISBN 5-06-004432-7.

6 Ахо, А. Построение и анализ вычислительных алгоритмов / А. Ахо, Дж. Хопкрофт, Дж. Ульмен. – М.: Мир, 1989. – 369с.

7 Никлаус Вирт Алгоритмы и структуры данных. – Санкт-Петербург: «Невский диалект», 2001.

8 Альсведе, Р. Задачи поиска / Р. Альсведе, И. Вегенер. – М.: Мир, 1982. – 368 с.

9 Бауэр, Ф.Л. Информатика. Вводный курс: в 2-х ч. / Ф. Л. Бауэр, Г. Гооз. – М.:Мир,1981.– ч.2. – 368с.

10 Гэри, М. Вычислительные машины и труднорешаемые задачи / М. Гэри, Д. Джонсон. – М.: Мир, 1982. – 416 с.

11 Дал, У. Структурное программирование / У. Дал, Э. Дейкстра, К. Хоор. –М.: Мир, 1975.

12 Калинин, А.Г. Универсальные языки программирования. Семантический

подход / А.Г. Калинин, И.В. Мацкевич. – Радио и связь, 1991.

13 Кристофидес, Н. Теория графов. Алгоритмический подход.– М.: Мир, 1978. – 432 с.

14 Лисков, Б. Использование абстракций и спецификаций при разработке программ / Б. Лисков, Дж. Гатэг. – М.: Мир, 1989.

15 Лэнгсам, Й. Структуры данных для персональных ЭВМ / Й. Лэнгсам, М. Огенстайн, А. Тененбаум. – М.: Мир, 1989. – 588с.

16 Беннер, В.М. Методы сортировок: методическая разработка для студентов / В.М. Беннер, Г.Т. Варкентина. – Барнаул: Изд-во БГПУ, 2001. – 25с.

17 Шухман, А. Е. Практикум по решению задач на ЭВМ: в 2-х ч. / А.Е Шухман. – Оренбург: ОГПУ, 2005. – ч. 2. – 80 с.

Приложение А

(справочное)

Ошибки компиляции в среде Turbo Pascal

Таблица А.1 – Ошибки компиляции в среде Turbo Pascal

Общие ошибки:

1. Out of memory (*Выход за границы памяти*)
2. Identifier expected (*Не указан идентификатор*)
3. Unknown identifier (*Неизвестный идентификатор*)
4. Duplicate identifier (*Двойной идентификатор*)
5. Syntax error (*Синтаксическая ошибка*)
6. Error in integer constant (*Ошибка в целой константе*)
7. String constant exceeds fine (*Строковая константа превышает допустимые размеры*)
8. Error in real constant (*Ошибка в вещественной константе*)
9. Unexpected end of file (*Не найден конец файла*)
10. Line too tons (*Слишком длинная строка*)
11. Type identifier expected (*Здесь нужен идентификатор типа*)
12. Too many open files (*Слишком много открытых файлов*)
13. File not found (*Файл не найден*)
14. Invalid file name (*Неверное имя файла*)
15. Disk full (*Диск заполнен*)
16. Undefiner typein pointer definition (*Неопределенный тип в объявлении указателя*)
17. Variable identifier expected (*Отсутствует идентификатор переменной*)
18. Error in type (*Ошибка в объявлении типа*)
19. Structure too large (*Слишком большая структура*)
20. Set-base type of range (*Базовый тип множества нарушает границы*)
21. File components may not be files (*Компонентами файла не могут быть файлы*)
22. Invalid string length (*Неверная длина строки*)
23. Type mismatch (*Несоответствие типов*)
24. Invalid subrange base type (*Неправильный базовый тип для типа диапазона*)
25. Lower bound greater than upper bound (*Нижняя граница больше верхней*)
26. Ordinal type expected (*Нужен порядковый тип*)
27. Integer constant expected (Нужна целая константа)
28. Consant expected (*Нужна константа*)

29. Integer or real constant expected (*Нужна целая или вещественная константа*)
30. Pointer type identifier expected (*Нужен идентификатор типа*)
31. Invalid function result type (*Неправильный тип результата функции*)
32. Label identifier expected (*Нужен идентификатор метки*)
33. BEGIN expected (*Нужен BEGIN*)
34. END expected (*Нужен END*)
35. Integer expression expected (*Нужно выражение типа INTEGER*)
36. Ordinal expresiion expected (*Нужно выражение перечисляемого типа*)
37. Boolean expression expected (*Нужно выражение типа BOOLEAN*)
38. Operand types do not match operator (*Типы операндов не соответствуют операции*)
39. Error in expression (*Ошибка в выражении*)
40. Illegal assigment (*Неверное присваивание*)
41. Field identifier expected (*Нужен идентификатор поля*)
42. Code segment too large (*Сегмент кода слишком большой*)
43. Data segment too large (*Сегмент данных слишком велик*)
44. DO expected (*Нужен оператор DO*)
45. OF expected (*Требуется OF*)
46. INTERFACE expected (*Требуется интерфейсная секция*)
47. Invalid relocatable refence (*Неправильная перемещаемая ссылка*)
48. THEN expected (*Требуется THEN*)
49. TO or DOWNTO expected (*Требуется TO или DOWNTO*)
50. Undefiner forward (*Неопределенное опережающее описание*)
51. Invalid typecast (*Неверное преобразование типа*)
52. Division by zero (*Деление на нуль*)
53. Invalid file type (*Неверный файловый тип*)
54. Cannot Read or Write variables of this type (*Нет возможности считать или записать переменые данного типа*)
55. Pointer variable expected (*Нужно использовать переменную-указатель*)
56. String variable expected (*Нужна строковая переменная*)
57. String expression expected (*Нужно выражение строкового типа*)
58. Circular unit refence (*Перекрестная ссылка модулей*)
59. Unit name mismatch (*Несоответствие имен программных модулей*)
60. Unit version masmatch (*Несоответствие версий модулей*)
61. Internal stack overflow (*Переполнение внутреннего стока*)
62. Unit file format error (*Ошибка формата файла модуля*)

63. Implementation expected (*Отсутствует исполняемая часть модуля*)

64. Constant and case types do not match (*Типы констант и тип выражений оператора CASE не соответствуют друг другу*)

65. Record or object variable expected (*Нужна переменная типа запись или объект*)

66. Constant out of range (*Константа нарушает границы*)

67. File variable expected (*Нужна файловая переменная*)

68. Pointer expression expected (*Нужно выражение типа указатель*)

69. Integer or real expression expected (*Нужно выражение вещественного или целого типа*)

70. Label not within current block (*Метка не находится внутри текущего блока*)

71. Label already defined (*Метка уже определена*)

72. Undefined label in processing statement part (*Неопределенная метка в предшествующем разделе операторов*)

73. Invalid @ argement (*Неправильный аргумент операции @*)

74. Unit expected (*Нужно кодовое слово Unit*)

75. <;> expected (*Нужно указать <;>*)

76. <:> expected (*Нужно указать <:>*)

77. <,> expected (*Нужно указать <,>*)

78. <(> expected (*Нужно указать <(>*)

79. <)> expected (*Нужно указать <)>*)

80. <-> expected (*Нужно указать <->*)

81. <:=> expected (*Нужно указать <:=>*)

82. <[> or <(.>expected (*Нужно указать <[> или <(.>*)

83. <]> or <.)> expected (*Нужно указать <]> или <.)>*)

84. <.> expected (*Нужно указать <.>*)

85. <..> expected (*Нужно указать <..>*)

86. Too many variables (*Слишком много переменных*)

87. Invalid FOR control variable (*Неправильный параметр цикла оператора FOR*)

88. Integer variable expected (*Нужна переменная целого типа*)

89. Files types are not allowed here (*Здесь не могут использоваться файлы*)

90. String length mismatch (*Несоответствие длины строки*)

91. Invalid orering of fields (*Неверный порядок полей*)

92. String constant expected (*Нужна константа строкового типа*)

93. Integer or real variable expected (*Нужна переменная типа INTEGER или REAL*)

94. Ordinal vairale expected (*Нужна переменная порядкового типа*)

95. Character expression expected (*Предшествующее выражение должно символьный тип*)

96. Overflow in arithmetic operation (*Переполнение в арифметической операции*)

97. No enclosing For, While or Repeat statement (*Операторы For, While или Repeat без окончания*)

98. Case constant out of range (*Константа Case нарушает допустимые границы*)

99. Error in statement (*Ошибка в операторе*)

100. Must be in 8087 mode to compile this (*Для компиляции необходим режим 8087*)

101. Target address not found (*Указанный адрес не найден*)

102. Include files are not allowed here (*Здесь не допускаются включаемые файлы*)

103. Invalid qualifier (*Неверный квалификатор*)

104. Invalid variable refence (*Недействительная ссылка на переменную*)

105. Too many symbols (*Слишком много обозначений*)

106. Statement part too large (*Слишком большой раздел операторов*)

107. Files must be var parameters (*Файлы должны передаваться по имени*)

108. Header does not match previous definition (*Заголовок не соответствует предыдущему определению*)

109. Cannot evaluate this expression (*Некорректное вычисление выражения*)

110. Invalid format specifier (*Неверный спецификатор формата*)

111. Invalid indirect refence (*Недопустимая косвенная ссылка*)

112. Structured variable are not allowed here (*Здесь нельзя использовать переменную структурного типа*)

113. Cannot evaluate without System unit (*Нельзя вычислить выражение без модуля SYSTEM*)

114. Cannot access this symbol (*Нет доступа к данному символу*)

115. Invalid floating-point operation (*Недопустимая операция с плавающей запятой*)

116. Procedure or function variable expected (*Должна использоваться переменная процедурного типа*)

117. Invalid procedure or function refence (*Недопустимая ссылка на процедуру или функцию*)

118. File access denied (*Отказ в доступе к файлу*)

119. Object type expected (*Здесь должен быть тип OBJECT*)

120. Local object types are not allowed (*Нельзя объявлять локальные объекты*)

121. VIRTUAL expected (*Пропущено слово VIRTUAL*)

122. Method identifier expected (*Пропущен идентификатор инкапсулированного правила*)

123. Virtual constructor are not allowed (*Конструктор не может быть виртуальным*)

124. Destructor identifier expected (*Пропущен идентификатор деструктора*)

125. Fail only allowed within constructor (*Неизвестный модуль*)

126. Invalid combination of opcode and operends (*Недопустимая комбинация кода команды и операндов*)

124. Memory refence expected (*Нужна ссылка на память*)

125. Invalid symbol refence (*Неверное обозначение ссылки*)

126. Code generation error (*Ошибка при генерации программы*)

127. Duplicate dynamic method index (*Повторяется индекс динамического правила*)

128. Procedure or function identifier expected (*Нужен идентификатор процедуры или функции*)

Ошибки, возникающие во время выполнения программы

Некоторые ошибки, обнаруженные во время выполнения программы, приводят к появлению на экране сообщения вида:

Runtime error nnn at xxxx:yyyy

(ошибка периода исполнения nnn по адресу xxxx:yyyy), где nnn- номер ошибки, xxxx:yyyy- адрес (сегмент и смещение). После этого сообщения программа завершает свою работу.

Ошибки периода исполнения делятся на четыре категории:

1) Ошибки, обнаруживаемые ДОС (*коды ошибок 1-99*);

2) Ошибки ввода\вывода (*100-149*);

3) Критические ошибки (*150-199*);

4) Фатальные ошибки (*200-255*);

Ошибки, обнаруживаемые ДОС

1. Invalid function number (*Неверный номер функции*)

2. File not found (*Не найден файл*)

3. Path not found (*Путь не найден*)

4. Too many open files (*Слишком много открытых файлов*)

5. File access defined (*Отказано в доступе к файлу*)

6. Invalid file handle (*Недопустимый файловый канал*)

7. Invalid file access code (*Недействительный код доступа к файлам*)

8. Invalid drive number (*Недопустимый номер дисковода*)

9. Cannot remove current directory (*Нельзя удалить текущий каталог*)

10. Cannot rename across drives (*Нельзя при переименовании указывать разные дисководы*)

Ошибки ввода\вывода

1. Disk read error (*Ошибка чтения с диска*)

2. Disk write error (*Ошибка записи на диск*)

3. File not assigned (*Файлу не присвоено имя*)

4. File not open (*Файл не открыт*)

5. File not open\or output (*Файл не открыт для вывода*)

6. Invalid numeric format (*Неверный числовой формат*)

Критические ошибки

1. Disk is write protected (*Диск защищен от записи*)
2. Unknown unit (*Неизвестный модуль*)
3. Drive not ready (*Дисковод находится в состоянии "Не готов "*)
4. Unknown command (*Неопознанная команда*)
5. CRC error in data (*Ошибка в исходных данных*)
6. Bad drive request strukture length (*При обращении к диску указана неверная длина структуры*)
7. Disk seek error (*Ошибка при операции установки головок на диск*)
8. Unknown media type (*Неизвестный тип носителя*)
9. Sector not found (*Сектор не найден*)
10. Printer out of paper (*Кончилась бумага на принтере*)
11. Device write\emit (*Ошибка при записи на устройство*)
12. Device read fault (*Ошибка при чтении с устройства*)
13. Hardware failure (*Сбой аппаратуры*)

Фатальные ошибки

Эти ошибки всегда приводят к немедленной остановке программы.

1. Division by zero (*Деление на нуль*)
2. Range check error (*Ошибка при проверке границ*)
3. Stack overflow error (*Переполнение стека*)
4. Hearp overflow error (*Переполнение кучи*)
5. Invalid pointer operation (*Недействительная операция с указателем*)
6. Floating point overflow (*Переполнение при операции с плавающей запятой*)
7. Invalid floating point operation (*Недопустимая операция с плавающей запятой*)
8. Floating point underflow (*Исчезновение порядка при операции с плавающей запятой*)
9. Object not initialized (*Не инициирован объект*)
10. Call to abstract method (*Вызов абстрактного правила*)

Приложение Б

(обязательное)

Ход выполнения лабораторных работ

Для успешной защиты лабораторных работ, студентам нужно выполнить следующие этапы:

1) изучить теоретический материал по теме лабораторной работы (лекции, учебники);
2) написать программу для каждого задания;
3) распечатать текст и результат программы для отчета;
4) оформить отчет по лабораторной работе;
5) защитить лабораторную работу.

Приложение В

(обязательное)

Содержание отчета по лабораторной работе

Отчет по лабораторным работам № 1–4 предоставляется следующим образом:

1) титульный лист;

2) цель лабораторной работы;

3) словесная постановка задачи. В этом подразделе проводится полное описание задачи. Описывается суть задачи, анализ входящих в нее физических величин, область их допустимых значений, единицы их измерения, возможные ограничения, анализ условий при которых задача имеет решение (не имеет решения), анализ ожидаемых результатов;

4) листинг программы. Подраздел должен содержать текст программы на языке программирования Turbo Pascal с комментариями;

5) результат выполнения программы;

6) выводы по лабораторной работе.

Отчет по лабораторным работам № 5–13 предоставляется следующим образом:

1) титульный лист;

2) словесная постановка задачи. В этом подразделе проводится полное описание задачи. Описывается суть задачи, анализ входящих в нее физических величин, область их допустимых значений, единицы их измерения, возможные ограничения, анализ условий при которых задача имеет решение (не имеет решения), анализ ожидаемых результатов;

3) математическая модель. В этом подразделе вводятся математические описания физических величин и математическое описание их взаимодействий. Цель подраздела – представить решаемую задачу в математической формулировке;

4) алгоритм решения задачи. В подразделе описывается разработка структуры алгоритма, обосновывается абстракция данных, задача разбивается на подзадачи. Схема алгоритма выполняется по ЕСПД (ГОСТ 19.701-90);

5) листинг программы. Подраздел должен содержать текст программы на языке программирования Turbo Pascal с комментариями;

6) результат выполнения программы;

7) выводы по лабораторной работе.