Алгоритмы и программы

Программирование - знакомство со средой

Как мы уже указывали в учебнике, в подавляющем большинстве случаев разработка программ выполняется с помощью интегрированной среды: специального средства, в котором объединены (как минимум):

1. Текстовый редактор. В современных системах этот текстовый редактор не только позволяет подготовить текст программы, но и существенно помогает при подготовке: выделяя ключевые слова, выдавая подсказки, обеспечивая работу с несколькими файлами и т.д.
2. Транслятор: компилятор или интерпретатор, переводящий готовый текст в машинный код
3. Отладчик. Средства, позволяющие проверить исполнение программы по шагам, проверить значения переменных во время исполнения, выявить ошибки и неудачно реализованные части.
4. Компоновщик и библиотеки функций, позволяющие решать типовые задачи (например, взаимодействия с внешней средой - чаще всего, операционной системой)

Современная среда может предоставлять и другие инструменты - но перечисленные компоненты есть обязательно.

Существует таких сред огромное количество и выбирают из них нужное по самым разным критериям. В нашем случае нам требуется среда:

1. Использующая язык Pascal (или основанный на нем)
2. Свободно распространяемая
3. Работающая под Windows и обеспечивающая доступ к ее средствам
4. С документацией на русском языке

Какие-то серьезные средства управления крупными проектами, подготовка специфических компонентов и прочие дополнительные возможности нам не потребуются.

Воспользуемся для решения имеющихся задач средой PascalABC.NET: среда эта предназначена для обучения программированию и свободно доступна по адресу: http://pascalabc.net/

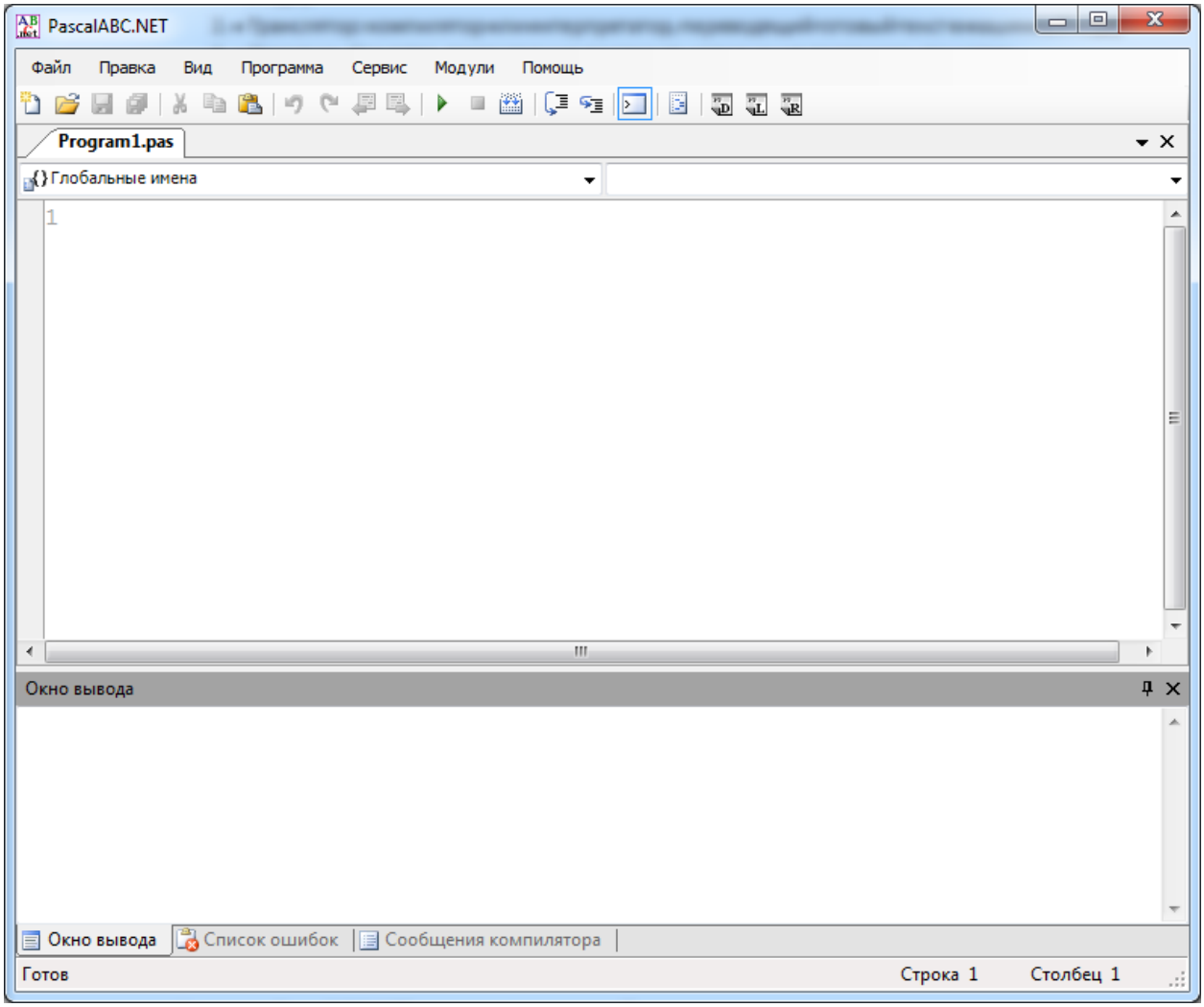
Особенно приятным для нас дополнением будет то, что он опирается и активно использует платформу .Net от Microsoft, что позволяет получить доступ к огромному количеству самых разных средств: от управления операционной системой до работы с самыми разными графическими средствами (например, 3D-графикой).

Познакомимся со средой и заодно повторим базовые приемы и средства программирования.

Интерфейс среды

Установка PascalABC.NET не представляет из себя ничего оригинального, поэтому останавливаться на ней мы не будем.

После запуска Вы увидите такое окно:



Интерфейс достаточно прост: в верхней части основное место занимает текстовый редактор. Редактор позволяет одновременно работать с несколькими файлами, выбрать файл можно наверху - с помощью закладки. Сейчас открыт один файл (пустой) под названием Program1.pas

Наберем в окне сверху первую программу. С учетом ее уровня сложности, текст подробно приводить не будем:

PascalABC.NET

Файл Правка Вид Программа Сервис Модули Помощь

Program1.pas*

Глобальные имена

```
1  program quadraticEquation;
2
3  var
4      a,b,c,discr,x1,x2 : double;
5
6  begin
7     write('Введите к-т A:');
8     readln(a);
9     write('Введите к-т B:');
10    readln(b);
11    write('Введите к-т C:');
12    readln(c);
13    discr := b*b - 4*a*c;
14    if discr < 0 then
15    begin
16       x1 := (-b-sqrt(discr))/(2*a);
17       x2 := (-b+sqrt(discr))/(2*a);
18       write ( 'Первый корень:',x1,' Второй корень:'x2 );
19    else
20       writeln ('Нет действительных корней');
21 end.
```

Окно вывода

Ошибки в 18 и 19 строке допущены намеренно.

Попробуем запустить эту программу. Для этого либо на панели нажмем кнопку "Выполнить", либо в меню "Программа" выберем команду "Выполнить". Если бы ошибок не было, программа после компиляции начала бы исполняться - но они есть:

```
16       x1 := (-b-sqrt(discr))/(2*a);
17       x2 := (-b+sqrt(discr))/(2*a);
18       write ( 'Первый корень:',x1,' Второй корень:'x2 );
19    else
20       writeln ('Нет действительных корней');
21 end.
```

Список ошибок

		Строка	Описание	Файл	Путь
	1	18	Встречено 'x2', а ожидалось ')'	Program1.pas	C:\PABCWork.NET

Из этой ошибки следует, что у нас нарушена структура строки: действительно, пропущена запятая. Исправляем, пытаемся запустить снова:

PascalABC.NET

Файл Правка Вид Программа Сервис Модули Помощь

Program1.pas*

Глобальные имена

```
8      readln(a);
9      write('Введите к-т B:');
10     readln(b);
11     write('Введите к-т C:');
12     readln(c);
13     discr := b*b - 4*a*c;
14     if discr < 0 then
15     begin
16        x1 := (-b-sqrt(discr))/(2*a);
17        x2 := (-b+sqrt(discr))/(2*a);
18        write ( 'Первый корень:',x1,' Второй корень:',x2 );
19     else
20        writeln ('Нет действительных корней');
21 end.
```

Список ошибок

		Строка	Описание	Файл	Путь
	1	19	Встречено 'else', а ожидался оператор	Program1.pas	C:\PABCWork.NET

Окно вывода | Список ошибок | Сообщения компилятора

1 ошибок Строка 19 Столбец 4

Теперь следующая ошибка: нарушена структура самой программы. Мы забыли закрыть операторные скобки: begin есть, end - нет. Поскольку программа написана "лесенкой" - мы сразу видим нарушенный блок. Исправляем, запускаем.

```
19     end
20     else
21        writeln ('Нет действительных корней');
22 end.
```

Окно вывода

```
Введите к-т A:
```

Ввод данных: Ввести Завершить

Окно вывода | Список ошибок | Сообщения компилятора | Локальные переменные | Просмотр выражений

Компиляция прошла успешно (22 строк) Строка 7 Столбец 1

Программа исполняется, и запрашивает с нас данные. Мы работаем в учебной среде, поэтому не видим классического экрана-консоли, и данные вводим в нижней часть закладки "Окно вывода":

```
20     else
21        writeln ('Нет действительных корней');
22 end.
```

Окно вывода

```
Введите к-т A:2
Введите к-т B:-3
Введите к-т C:-8
```

Ввод данных: -8 Ввести Завершить

Окно вывода | Список ошибок | Сообщения компилятора | Локальные переменные | Просмотр выражений

Компиляция прошла успешно (22 строк) Строка 7 Столбец 1

Вводим данные, получаем результат.

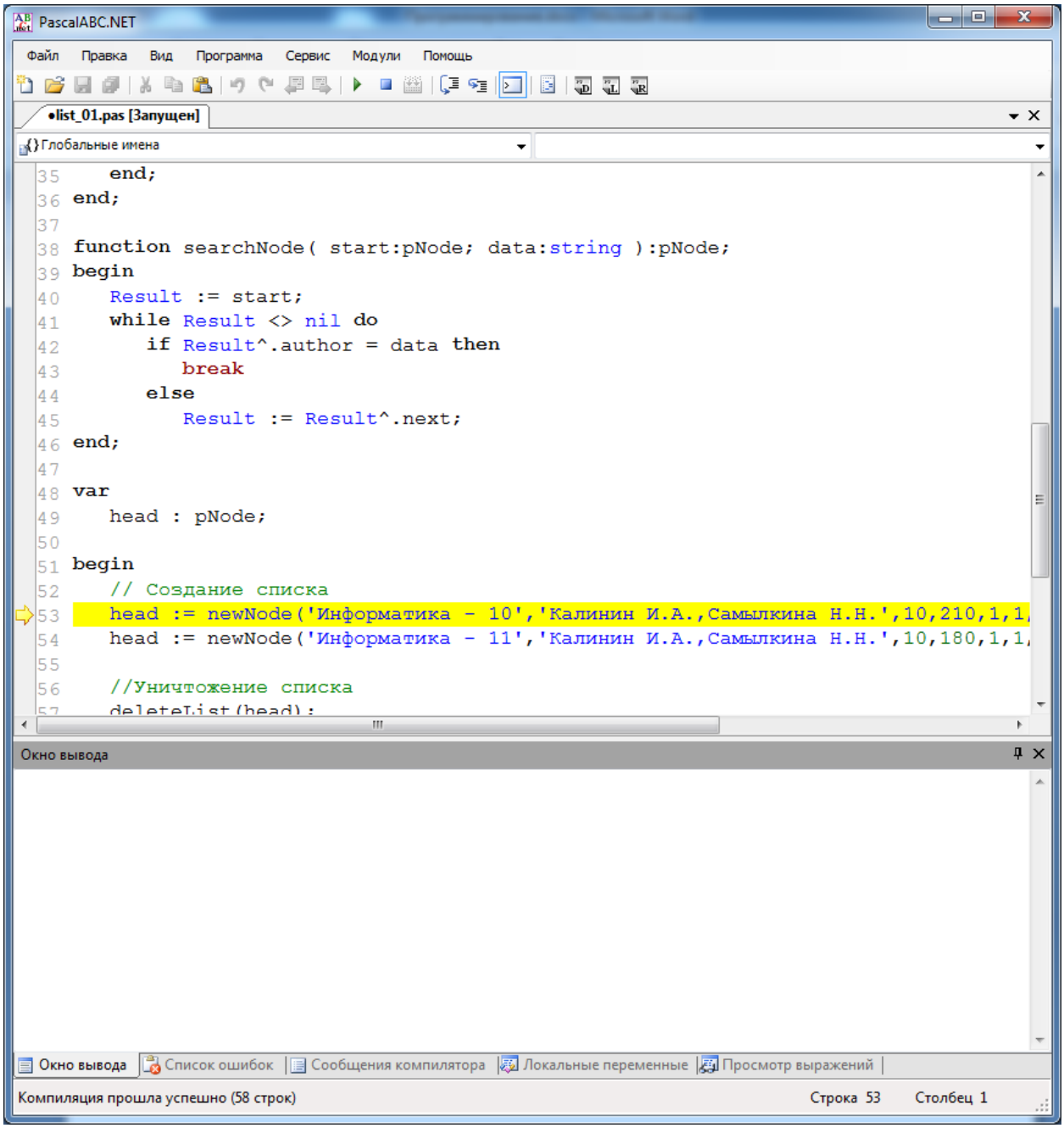
Обратите внимание, мы не могли запустить программу до исправления всех ошибок. Классические правила работы компилятора Pascal таковы, что ошибки мы должны устранять одну за другой - нельзя увидеть весь список допущенных Вами ошибок.

Разумеется, компилятор и компоновщик могут выявить только формальные, синтаксические ошибки. Логические ошибки, допущенные вами, в подавляющем большинстве случаев никто, кроме грамотного программиста, выявить и устранить не сможет.

Тем не менее, среда имеет в своем составе инструменты, предназначенные для выявления таких ошибок. Это средства отладки программ: пошаговый отладчик, точки останова и средства просмотра значений переменных. Разберем эти средства.

Начнем с основного: пошагового отладчика. Это средство, которое позволяет нам проверить исполнение программы по шагам - контролируя значения переменных и последовательность исполнения.

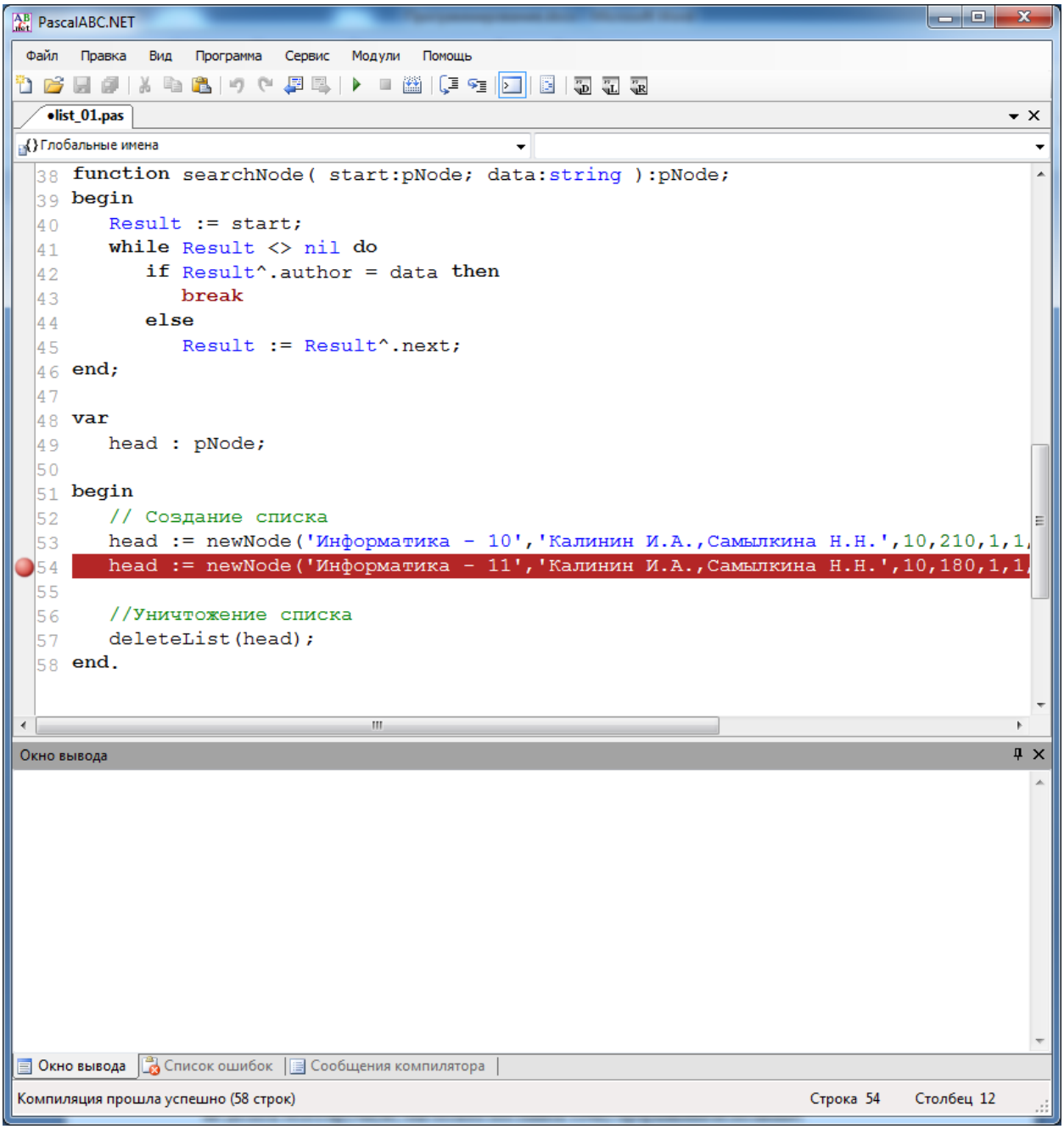
Чтобы запустить такую отладку нужно вместо запуска программы (клавишей F9) выполнить шаг (клавиша F8 или F7). Клавиша выбирается в зависимости от того, хотите ли вы попасть внутрь подпрограммы - процедуры или функции (F7) или нет (F8). Вот как выглядит рабочее окно в таком режиме:



Оператор, ожидающий исполнения, выделен желтым. При нажатии на F8 он выполнится и система перейдет к следующему шагу, при нажатии на F7 - мы попадем внутрь функции newNode и сможем проверить ее работу.

Исполняя программу по шагам, мы можем проследить как она исполняется - например, выяснить правильно ли работают условия, циклы, функции. Так можно выявить конкретные места возникновения ошибок.

Конечно, если программа велика проходить ее всю до места отладки очень затруднительно. Чтобы не делать этого вручную, мы можем поставить точку прерывания исполнения. Для этого нужно на то строке, с которой вы хотите начать пошаговую проверку при редактировании щелкнуть на полях:



Теперь можно просто запустить программу - когда дойдет до этого места, исполнение остановится и система будет ждать вашей команды.

Структуры данных.

Если ваша программа больше 10 строк - скорее всего вам пришлось задуматься о том, как хранить данные для работы. Если данных мало - достаточно объявить несколько переменных.

Если данных много, причем заранее неизвестно сколько - задача существенно усложняется. Чтобы использовать циклические структуры (не описывать же все перебор сравнением всех

переменных?), ускорить работу по поиску и добавлению данных, оптимизировать память применяют конструкции, получившие название ***структур данных***.

Структура данных - способ организации связанного набора данных в памяти, позволяющий их хранить и обрабатывать.

Из сказанного следует, что любая структура реализуется с помощью программного кода, который позволит:

1. Добавлять новые данные - т.е. записывать их в память, сохраняя общую логику.
2. Получать доступ к уже имеющимся данным - для изменения, например.
3. Искать нужные данные для каких-то операций
4. Удалять ненужные данные

Задачи эти, конечно, связаны - например, чтобы изменить данные их сначала надо найти.

Универсального способа хранить данные не существует, в каждой программе вы все равно будете выбирать оптимальный способ - исходя из задачи, имеющихся ресурсов, средств и знаний. Тем не менее, в подавляющем большинстве случаев для построения своего способа используют типовые средства и решения, которые мы и разберем.

Массив

Простейшая структура данных вам уже, возможно, известна - ***массив***, то есть (в языке Pascal или C) - множество пронумерованных однотипных элементов. У массива масса достоинств - он прост в объявлении, понятен, обеспечивает доступ к любому своему элементу, сохраняется и считывается с диска.

В языке Pascal массив объявляется с помощью ключевого слова array:

```
var

    temperature : array[1..100] of double;
```

Этот способ прост, но ограничивает наши возможности и не является рекомендованным. Лучше для всех сложных конструкций объявить специальный тип данных, а потом в секции переменных его использовать:

```
type

    tempBase : array[1..100] of double;

var

    temperature : tempBase;
```

Такой конструкцией мы объявим массив под названием temperature из 100 переменных типа double, пронумерованных от 1 до 100. К каждой отдельной переменной (элементу массива) я могу получить доступ по ее номеру-индексу:

```
    temperature[10] := 5.3;
```

Массив очень удобен для организации циклических операций - ввода, вывода, обработки.

Массив поможет вам только если вы точно знаете сколько данных у вас будет (хотя-бы для того, чтобы оценить размер массива). Вторая его неприятная особенность: он должен храниться в памяти одним большим разделом, разбить его на отдельные части нельзя и увеличение его если и возможно - будет занимать много времени. В массив не всегда просто добавить или удалить элемент.

Современные языки часто позволяют реализовать вам т.н. динамический массив - т.е. массив, размеры которого можно изменять во время работы программы. Стоит учитывать, что такая операция может занять немало времени.

Чуть более сложная конструкция, объединяющая несколько переменных - запись (она же - структура). Запись позволяет объединить разнотипные элементы, и обращаться к ним по именам.

Чтобы пользоваться записями, сначала объявляют тип:

```
Type
  Book = record
   title : string;
   author: string;
   count : word;
   price : longword;
   place1 : word;
   place2 : word;
   place3 : word;
  end;
```

А потом, в секции Var, переменную этого типа:

```
var

        myBook : Book;
```

К элементам записи мы можем обращаться таким образом:

```
        myBook.author := 'М.Дример';
```

Типы массива и записи вполне можно комбинировать, создавая, например, массив записей (таблицу). Эту возможность мы рассмотрим в рамках решения задачи-примера. Цель этого примера не столько изучить структуру данных как таковую, сколько показать какие операции нам вообще потребуются.

Постановка задачи: нам требуется написать программу хранения и поиска данных о книгах в книжном магазине. О каждой книге нам известно: название, автор, количество, цена, номер зала, номер шкафа и номер полки. Известно, что памяти достаточно, а книг не может быть больше 1000 наименований.

Решение:

Во-первых, создадим структуру для хранения данных. Элемент данных (книга) описывается не одним числом или строкой, а целым набором отдельных переменных. Мы объединяем их в запись, описав сложный тип в разделе Type:

```
Type
  book = record
   title : string;
   author: string;
   count : word;
   price : longword;
   place1 : word;
   place2 : word;
   place3 : word;
  end;
  bookStore : array[1..1000] of book;
```

После этого можем объявить массив записей:

```
Var
   store : bookStore;
   storeCount : word;
```

Обратите внимание, кроме самого массива нам требуется переменная, с помощью которой мы будем определять первую свободную запись массива. Чтобы не усложнять хранение, будем считать, что это счетчик количества книг. Это усложнит удаление записей, но упростит объяснения.

Добавление новой записи пояснений не требует: вносим данные и переходим к следующей свободной записи:

```
procedure newArrayNode (title : string;author: string; count :
word; price : longword;place1 : word; place2 : word; place3 :
word, var store : bookStore);
begin
     store[storeCount].title := title;
     store[storeCount].author:= author;
     store[storeCount].count:= count;
     store[storeCount].price:= price;
     store[storeCount].place1:= place1;
     store[storeCount].place2:= place2;
     store[storeCount].place3:= place3;
     inc(storeCount);
end;
```

Обратите внимание: массив мы передаем с ключевым словом var - чтобы изменять его, не теряя изменений при возврате.

Процедура печати записи:

```
procedure itemArrayPrint ( pos : word; var store : bookStore);
begin
     write('|',store[pos].title, '|');
     write(store[pos].author, '|');
     write(store[pos].count, '|');
     write(store[pos].price, '|');
     write(store[pos].place1, '|');
     write(store[pos].place2, '|');
     writeln(store[pos].place3, '|');
end;
```

Поиск записей, соответствующих условию, это просто циклический перебор:

```
function findRecord( author:string; startSearchPos:word; var
store : bookStore):word;
var
     current : word;
begin
     current := startSearchPos;
     while (store[current].author <> author) and ( current <
storeCount) do
          inc(current);
end;
```

Тут мы предполагаем, что фактическое количество записей в массиве хранится в глобально доступной переменной. При необходимости (и это было бы правильнее) мы можем и эту величину передавать.
Если нам понадобится найти все записи (например, для печати) мы можем несколько раз вызвать эту функцию - меняя стартовую позицию. Например:

```
procedure printAuthorBooks( author : string; var store :
bookStore);
var
     pos : word;
begin
     pos := findRecord( author, 1,store);
     while pos < storeCount do
     begin
          itemArrayPrint(pos);
          pos := findRecord( author, pos+1,store)
     end;
end;
```

Удаление записи из массива потребует от нас перестановки всех записей от удаляемой "вверх" - то есть к началу:

```
procedure deleteArrayItem( del : word; var store : bookStore);
var
     pos : word;
begin
     for pos := del to storeCount-2 do
     begin
          store[pos].title := store[pos+1].title;
          store[pos].author:= store[pos+1].author;
          store[pos].count:= store[pos+1].count;
          store[pos].price:= store[pos+1].price;
          store[pos].place1:= store[pos+1].place1;
          store[pos].place2:= store[pos+1].place2;
          store[pos].place3:= store[pos+1].place3;
     end;
     dec(storeCount);
end;
```

Обратите внимание: любая операция, требующая поиска имеет сложность как минимум O(N) - т.е. линейно растущую с длиной массива, а если эта операция еще и повторяется- зависимость становится и более "тяжелой". Пример такой операции - вставка элемента с предварительной проверкой.

Работа с файлами

Каждый раз вводить данные заново (особенно, во время отладки) - не самый лучший способ. Как вам прекрасно известно, если нужно сохранить данные на достаточно долгий срок - используются файлы.
Для реализации работы с файлами в языке Pascal предусмотрен набор функций, который мы сейчас используем для реализации функций записи и чтения данных.
В языке Pascal предусматриваются средства для работы с файлами двух основных типов: двоичных и текстовых.

В текстовых файлах данные хранятся в виде текста (именно текста - строго букв и цифр, т.е., например, число 1 будет записано байтом со значением 49), и разделяются на строки - парой из символов №10 и №13 (подробнее - в разделе текст).
Двоичные данные хранят любые данные так, как они хранятся в памяти компьютера.

В свою очередь двоичные файлы могут быть типизированными (т.е. указано, из какого типа элементов состоит файл) и бестиповыми - т.е. потока байтов, которые программа должна анализировать самостоятельно.

Мы для работы будем использовать самый удобный тип - типизированный двоичный файл, состоящий из записей объявленного нами типа.
Перед началом работы мы должны объявить переменную, которая будет указывать на файл. При ее описании мы должны указать какого типа данные хранятся в файле. Переменная описывается так:

```
Var
     bookStoreData : file of book;
```

Перед началом работы с файлом необходимо связать переменную с конкретным файлом на диске:

```
Assign(bookStoreData,'books.dat');
```

Файл будет открыт в том же каталоге, где будет запущена программа.
Далее файл необходимо открыть на запись:

```
Rewrite(bookStoreData);
```

И после этого процедурой write можно записывать туда данные:

```
          write(bookStoreData, store[pos]);
```

Процедура сохранения данных на диске очень сильно напоминает все остальные процедуры - это просто перебор записей:

```
procedure saveArray( fname : string; var store : bookStore);
var
     pos : word;
     bookStoreData : file of book;
begin
     Assign(bookStoreData,fname);
     Rewrite (bookStoreData);
     for pos := 0 to storeCount-1 do
```

```
            write(bookStoreData, store[pos]);
    Close(bookStoreData);
end;
```

Чтение данных из файла имеет ряд отличий. Во-первых, надо открыть файл не на запись, а на чтение. Это делается заменой процедуры Rewrite на Reset. Эта процедура откроет файл и поставит указатель чтения на начало файла.
Во-вторых, мы не знаем сколько записей в файле - а поэтому читаем до конца. Конец мы определим с помощью логической функции eof (Eond Of File - конец файла).
Само чтение выполняется процедурой read.

```
procedure loadArray( fname : string; var store : bookStore);
var
     pos : word;
     bookStoreData : file of book;
begin
     Assign(bookStoreData,fname);
     Reset (bookStoreData);
     pos := 0;
     while not eof(bookStoreData) do
     begin
          read(bookStoreData, store[pos]);
          inc(pos);
     end;
    Close(bookStoreData);
end;
```

Многие из этих функция требуют доработки. Например:

1. При печати никак не учитывается, что данные имеют разную длину в разных записях. В результате не получится таблица
2. При добавлении записи мы не проверяем, можно ли это сделать - ведь массив всего на 1000 единиц.
3. Нет процедуры вставки данных (а не добавления).
4. Нет ввода и вывода из файла.
5. Собственно основной программы "ведения" этой миниатюрной базы данных мы так и не написали.

Так что, как видите, пространство для доработки - предостаточное.

Сортировка и поиск.

Одна из проблем, которую обязательно приходится решать при работе с большими структурами данных - организация поиска. От его эффективности напрямую зависит выполнение любых операций, связанных с выбором элемента.
До сих пор мы использовали только один способ поиска и отбора элементов - полный перебор. Этот способ прост для понимания, но приводит к огромным затратам времени, а иногда и памяти. Мы можем его ускорить, но при одном условии: если массив будет отсортирован.

Начнем с самой операции сортировки. В учебнике мы рассмотрели несколько способов сортировки массива. Реализуем самый быстрый из них:

```
function Partition(var A:sortArray; l,r : integer):integer;
var
```

```
   i,j,x,b : integer;
begin
   x := A[ l + (r-l)div 2];
   i := l-1;
   j := r+1;
   repeat
      repeat
         dec(j);
      until (A[j] <= x);
      repeat
               inc(i);
          until (A[i] >= x);
          if i < j then
          begin
               b := a[i];
               a[i] := a[j];
               a[j] := b;
          end;
     until i >= j;
   Partition := j;
end;

procedure QuickSort(var A:sortArray; l,r : integer);
var
   q : integer;
begin
   If r > l then
   begin
          q := Partition(A,l,r);
          QuickSort(a,l,q);
          QuickSort(a,q+1,r);
   end;
end;
```

Мы подробно рассмотрели в учебнике этот алгоритм сортировки, дадим только несколько комментариев.
Во-первых, в этом примере мы использовали условный тип sortArray, на месте которого должен быть ваш тип массива.
Во-вторых, при организации сравнения и перестановки элементов нам придется учесть то, что это записи. Не во всех случаях возможна простая перестановка присваиванием, поэтому стоит проверить как это работает.

Теперь можем оптимизировать поиск. Идея бинарного поиска очень проста. Если у нас есть отсортированный (для определенности - по неубыванию) массив, то меньший элемент обязательно слева от большего.

Тогда: сравним искомый ключ с элементом в середине массива. Если ключ меньше - он обязательно слева, будем проверять левую половину. Если больше - обязательно справа, и проверять надо правую часть. Если он равен середине - мы нашли искомое.

На каждом таком шаге мы уменьшаем количество элементов для проверки в два раза. Будем повторять операцию выбора левой или правой части, пока либо найдем элемент,

либо выясним что его нет. Алгоритм уже в учебнике разобрали, остается его только записать:

```
function binarySearch (key:integer; A:sortArray) : integer;
var
   middle,left,right : integer;
begin
   left:=1;
   right:=100;
   while left < right do
   begin
      middle := left + (right-left) div 2;
      if a[middle] > key then
         left := middle +1
      else
         if a[middle] <> key
            right := middle -1
         else
            break;
   end;
   if a[middle] = key then
      result := middle
   else
      result := -1;
end;
```

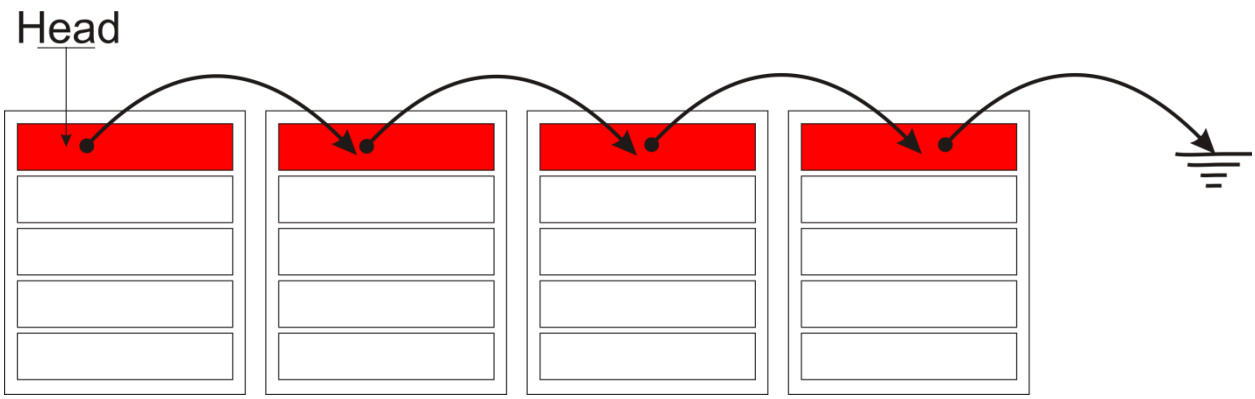
Ключевых условий у применения этого алгоритма в общем два:

1. Мы должны иметь возможность обратиться к каждому элементу последовательности по его номеру. Это не всегда возможно, как мы чуть позже увидим.
2. Последовательность должна быть отсортирована. Причем направление сортировки будет очевидно влиять и на знаки сравнения в алгоритме.

Список

Более сложная, но и более гибкая структура хранения - список. Его ключевое отличие от массива - линейная связь элементов. Т.е. прямого доступа к любому элементу у нас нет, но есть возможность перейти к следующему по порядку и так добраться до нужного.

Схематически это можно обозначить так:



Постановка задачи: все тот же склад книжного магазина, но с условиями: неизвестно количество книг, объем памяти ограничен, заказчик недоволен временем вставки и удаления записей - особенно, в самое начало массива.

Решение:

Во-первых, создадим структуру для хранения данных. Она очень похожа на предыдущую, но содержит добавление - ссылку на "следующий" элемент (специальный тип pNode).

```
type
   pNode = ^ListNode;
  ListNode = record
   next : pNode;
   bookData : book;
  end;
```

Обратите внимание, мы, как и в случае с массивом, для сохранения логики обработки используем тип book - запись о книге. Помимо прочего, это позволит нам хранить данные в файле - не меняя его формат.

Добавление записи:

```
function itemListNew (title : string;author: string; count :
word; price : longword;place1 : word; place2 : word; place3 :
word; next : PNode):pNode;
begin
   new (Result);
   Result^.title := title;
   Result^.author := author;
   Result^.count := count;
   Result^.price := price;
   Result^.place1 := place1;
   Result^.place2 := place2;
   Result^.place3 := place3;
   Result^.next := next;
end;
```

Добавление записи в список не сложнее, чем добавление записи в массив. Сначала мы отводим память на новый элемент, вносим туда данные, а потом связываем его с предыдущим элементом.

В основной программе нам потребуется хранить указатель только на "голову" списка.

Вот пример кода такой программы:

```
var
   head : pNode;

begin
   // Создание списка
   head := itemListNew ('Информатика - 10','Калинин
И.А.,Самылкина Н.Н.',10,210,1,1,1,nil);
   head := itemListNew ('Информатика - 11','Калинин
И.А.,Самылкина Н.Н.',10,180,1,1,1,head);
```

В этом примере мы создаем список, присваивая указатель на новый элемент головному указателю списка. Чтобы не терять уже добавленные элементы, передаем предыдущий указатель.

Печать записи:

```
procedure itemListPrint ( item : pNode );
begin
     write('|', item^.title, '|');
     write(item^.author, '|');
     write(item^.count, '|');
     write(item^.price, '|');
     write(item^.place1, '|');
     write(item^.place2, '|');
     writeln(item^.place3, '|');
end;
```

Поиск записи, соответствующей условию:

```
function findListItem( author:string; startSearch:pNode):pNode;
var
     current : pNode;
begin
     current := startSearch;
     while ( current <> nil)and(current^.author <> author) and
do
     current := current^.next;
end;
```

В этом коде мы начинаем с какого-то стартового узла и переходим к следующему узлу пока не найдем искомое - или не дойдем до последнего элемента. Если искомое найти не удалось, то вместо указателя на элемент функция вернет значение nil.

Если нам надо найти все вхождения (а не только одно первое) - поступим так же, как и с массивом, то есть будем перебирать, меняя начальный элемент, пока не дойдем до конца списка.

Удаление записи:

```
function itemListDelete(item : pNode) : pNode;
begin
     result := item^.next;
     Dispose ( item);
end;
```

На что стоит обратить внимание, так это на общий подход к обходу списка: мы начинаем с какого-то элемента и переходим к следующему, пока не сочтем нужным прерваться или не дойдем до конца.

Так мы можем:

- вывести список (или некоторые его элементы, отобрав их по условию)
- удалить элементы
- найти позицию и вставить элемент в список
- Сохранить элементы в файле
- Посчитать количество элементов
- и т.д.

Но каждый раз писать проход по списку для разных целей представляется не самым оптимальным решением. Современные языки программирования позволяют решить эту задачу изящнее: с помощью ***указателя на функцию***.

Сначала объявим нужный тип:

```
type
     iterate = function( item : pItem): pItem;
```

Напишем универсальную функцию перебора:

```
procedure iteratorList(start : pNode; iterateFunction :
iterate);
var
   current : pNode;
begin
   current := start;
   repeat
      current:=iterateFunction(current);
   until current = nil;
end;
```

Теперь можем написать функцию, например, печати всех элементов по условию:

```
function printListItem (item : pNode) : pNode;
begin
     if item^.author='Самылкина' then
          writeln(item^.author,',',item^.title,',',item^.price);
     result := item^.next;
end;
```

Вызов для перебора выглядит так:

```
  iteratorList(head,printListItem);
```

Самостоятельные упражнения:

1. Напишите функцию печатающую все элементы в виде таблицы
2. Перепишите функцию освобождения списка, с помощью функции-итератора
3. Напишите функцию подсчета количества элементов в списке
4. Напишите функции записи и чтения списка из файла, по аналогии с такими функциями из раздела массивы

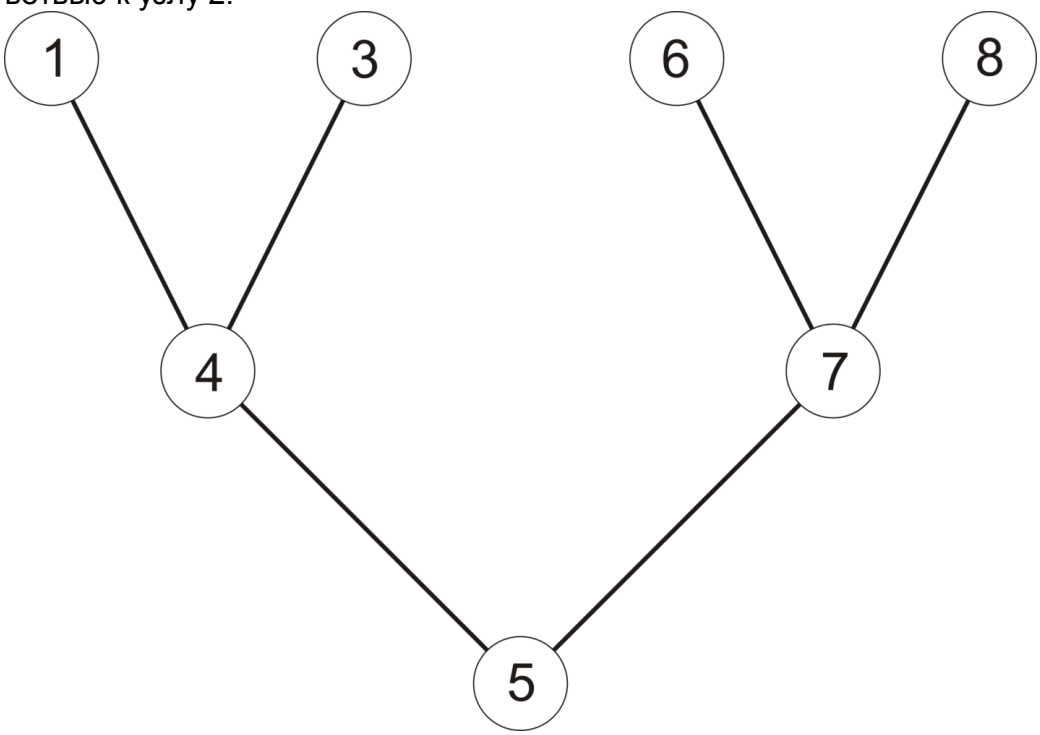
5. Предложите доработку функции-итератора, которая позволит запись и чтение выполнять с ее помощью.

Бинарное дерево поиска

Рассмотренный нами алгоритм поиска к списку применять очень невыгодно - из-за больших накладных расходов и при сортировке, и при переходах между элементами. Но существуют другие способы, организации эффективного поиска, которые нам и стоит рассмотреть. В их основе лежат деревья.

Простейший вариант дерева - бинарное дерево, разобранное нами в учебнике.
Его основное достоинство - отсутствие необходимости вычислять позицию нужного элемента. То есть, нам не нужна возможность перейти к любому элементу структуры - мы можем “дойти” до нее последовательно.

Вот как схематически выглядит структура бинарного поискового дерева для набора числе: 1,3,4,5,6,7,8
Стоит заметить, что если мы будем добавлять число 2, то оно окажется правой ветвью к узлу 2.



Структура элемента дерева похожа на структуру элемента списка:

```
type
  pNode = ^TreeNode;
  treeNode = record
    left, right: pNode;
    key: word;
 end;
```

В элементе-узле дерева, три обязательных части: ключ (собственно данные) и минимум два указателя на другие узлы. Во время поиска, в зависимости от значения поискового ключа, мы можем идти либо по “левой” либо по “правой” ветви. Чтобы не путаться, будем считать, что элементы со значениями меньшими будут находиться слева, а большими - справа.

По определению дерева, на каждый узел может сослаться как на "следующий" только один другой узел - во избежание появления циклов, то есть замкнутых путей.

Создание элемента для такого дерева выполняется функцией, очень похожей на добавление элемента в список:

```
function nodeTreeNew (key : word; left,right : PNode):pNode;
begin
   new (Result);
   Result^.key := key;
   Result^.left := left;
   Result^.right := right;
end;
```

Но добавить правильно такой элемент в дерево можно только найдя его место. Для этого нам потребуется функция поиска элемента в дереве.

```
function searchTree( root : pNode; key:word) : pNode;
var
    current : pNode;
begin
     current := root;
     while (( current <> nil )and(current^.key <> key) do
     begin
          result := current;
          if ( key > current^.key) then
               current := current^.rigth
          else
               current := current^.left;
     end;
     if current <> nil then
          result := current;
end;
```

Функция вернет нам указатель на ближайший похожий элемент. Если такой ключ уже есть в дереве - то на его узел.

Функция добавления ключа будет выглядеть так:

```
function insertTreeNode (key:word; root : pNode ):pNode;
var
     parentNode : pNode;
begin
     parentNode := searchTree(root,key);
     if ( parentNode^.key = key) then
          result := parentNode
     else
     begin
          if (parentNode^.key > key) then
               parentNode^.left := nodeTreeNew(key,nil,nil)
          else
               parentNode^.right := nodeTreeNew(key,nil,nil);
     end;
```

```
end;
```

Если ключ уже есть в дереве - мы не добавляем ничего, а возвращаем ссылку на него. Если его там еще нет - создаем его и присоединяем либо слева (если он меньше), либо справа.

Обход дерева

Печать элементов дерева, подсчет и другие операции, как и в случае со списком, удобно реализовывать с помощью указателей на функции.
Основной функцией для реализации у нас будет обход дерева. Выполнять его (в отличие от списка) можно по разному:
- сначала левое поддерево, потом правое, потом обработка самого узла
- сначала правое поддерево, потом обработка узла, потом левое.
- сначала обработка узла, потом левое дерево, потом правое.
- и т.д.

Для каких-то задач это не важно, для каких-то очень важно. Напишем для примера первый вариант:

```
type
   iterate = function( item : pItem): pItem;

function iterateTree (root : pNode; iterateFunction :
iterate):pNode;
var
   current : pNode;
begin
   current := start;
   repeat
     if current^.left <> nil then
          iterateTree(current^.left,iterateFunction);
     if current^.right <> nil then
          iterateTree(current^.right,iterateFunction);
     iterateFunction(current);
   until true;
end;
```

Обратите внимание: мы использовали одно из свойств дерева: его часть это тоже дерево (по отношению к исходному - поддерево). Отразилось это в использовании приема, который называется рекурсией: мы вызвали функцию внутри ее самой. Стоит помнить, что прием это очень мощный, но и опасный: программа может попасть в бесконечный цикл и очень быстро исчерпать ресурсы. В случае с деревом нас (при правильной реализации алгоритма) от этого защищает условие отсутствия внутренних циклов.

Задания:
1. Напишите функцию для удаления дерева, используя функцию-обходчик
2. Напишите функцию для подсчета количества элементов в дереве.
3. Напишите функцию подсчета узлов, у которых заполнен только одна ветвь.
4. Напишите итератор, с помощью которого можно будет вставлять элемент в дерево.

Хэширование

Деревья - эффективный способ организации поиска, но, как мы видели, у него есть целый ряд недостатков:

- есть вероятность, что мы не получим никакой выгоды - если дерево окажется несбалансированным.
- операция добавления с ростом дерева становится все дольше - поскольку все больше элементов надо будет проверить.
- на достаточно большом объеме данных дерево станет очень большим - и поиск все равно окажется довольно долгим

Один из вариантов решения такой задачи мы в учебнике описывали - это использование хэш-функции.
Идея проста: напишем программу, которая по значению ключа рассчитает число - хэш. Число будет рассчитываться так, чтобы значения поискового ключа распределились среди всех возможных значений хэша равномерно.
Тогда при поиске вместо сравнений мы сразу рассчитаем хэш - и будем искать только среди ключей с таким же значением хэша.
Реализуем такую задачу: у нас есть список слов русского языка. Требуется: написать программу которая быстро проверить: есть слово в этом списке или нет.

Для решения используем хэш-функцию, которую описывали в учебнике для строк. В первом варианте нашей программы мы проверим - действительно ли хэш-функций "разравнивает" ключи?
Зададим для функции количество хэш-значений: достаточно большое простое число 577.

```
program hashedSearch;

const
   NumHashLen = 577;
```

Поисковые ключи с одинаковым хэшем будем пока для простоты добавлять в список. Список реализуем точно также, как уже рассматривали - поэтому тут приведем только саму структуру:

```
type
  pHashNode = ^HashNode;
  HashNode = record
    val : string[40];
    next : ^HashNode;
  end;
```

Функцию вычисления хэш-значения по ключу (в нашем случае - строке), мы реализуем прямо по алгоритму, предложенному в учебнике:

```
function hashBKDR(S:string) : word; // Хэш-функция, предложенная
Б.Керниганом и Д.Ритчи
var
   i : byte;
   h  : uint64;
begin
  h := 0;
  for i := 1 to length(s) do
    h := h*61 + ord(s[i]); // 61 - Ближайшее простое число к
количеству возможных символов русского и английского алфавитов
  hashBKDR := h mod NumHashLen;
end;
```

В основной программе заведем два массива: массив списков и массив количества значений. Для каждого прочитанного слова мы вычислим хэш - и он станет позицией-номером в массивах.
Данные (то есть слова) мы прочитаем из текстового файла[1], в котором каждое слово записано на отдельной строке.

```
var
  SearchArray : array[0..NumHashLen] of pHashNode;
  CountArray : array[0..NumHashLen] of longword;
  txt : PABCSystem.Text;
  hs  :  word;
  n : pHashNode;
  s : string;
begin
   for hs:=0 to NumHashLen do // Инициализация пустых массивов
   begin
      CountArray[hs] := 0;
      SearchArray[hs]:= nil;
   end;

  Assign(txt,'1grams-3.txt');
  Reset(txt);
  while not Eof(txt) do
  begin
    ReadLn(txt,s);// Читаем по одному слову
    if length(s) > 1 then
    begin
      hs := hashBKDR(s); // Вычисляем хэш
      SearchArray[hs] := addNode( s, SearchArray[hs] ); //
добавляем элемент в его список
      inc(CountArray[hs]); // увеличиваем количество таких
элементов
    end;
  end;
  close(txt);

  For hs:=0 to NumHashLen do // Выведем количество элементов с
каждым значением хэш-функции
  begin
      Writeln(hs,' - ',CountArray[hs]);
      while SearchArray[hs] <> nil do
      begin
         n := SearchArray[hs]^.next;
         Dispose(SearchArray[hs]);
         SearchArray[hs] := n;
      end;
  end;
end.
```

[1] Файл получен с сайта корпуса русского языка, адрес: http://www.ruscorpora.ru/ngrams/1grams-3.zip

Запустив эту программу мы выясним, что все слова распределились по 577 значениям хэша примерно одинаково. Таким образом, мы можем за один шаг уменьшить количество элементов для поиска не в 2 раза (как в бинарном дереве), а в 577.

К сожалению, второй шаг так просто не сделаешь - такой же хэш для этих ключей рассчитывать бесполезно. Но мы можем заменить хэш-функцию - и рассчитать другие значения. Например, целый ряд таких функций приведен и проанализирован здесь: http://www.vak.ru/doku.php/proj/hash/efficiency