Дмитриева Марина Валерьевна

ЗАНЯТИЕ 11. ДЕРЕВЬЯ В ПРОГРАММИРОВАНИИ: ПОСТРОЕНИЕ И ИСПОЛЬЗОВАНИЕ

Многие реальные объекты имеют иерархическую структуру, например, схема предприятия или структура власти в государстве, генеалогическое дерево семьи или родословная некоторого человека. Для представления таких объектов и обработки связанной с ними информации удобна организация данных, отражающая структуру объектов. Если абстрагироваться от конкретного содержания элементов, то получится математический объект, называемый деревом. Рассмотрим некоторые способы представления и обработки деревьев.

ПРЕДСТАВЛЕНИЕ ДАННЫХ С ПОМОЩЬЮ ДЕРЕВА

Данные, имеющие иерархическую структуру, удобно представлять с помощью деревьев. Дерево представляет собой конечное множество элементов, называемых узлами, или вершинами. Вершины расположены на разных уровнях иерархии. На самом высоком уровне иерархии (пусть номер этого уровня 1) располагается единственный узел, называемый корнем. Каждый узел, расположенный на i-ом уровне, связан на более высоком $(i-1)$-ом уровне с единственным узлом, который является исходным, или предком для данного узла. Каждый узел i-ого уровня может быть связан с одним или несколькими узлами на более низком $(i+1)$-ом уровне. Такие узлы $(i+1)$-ого уровня называются порожденными узлами, или потомками. Узлы, не имеющие порожденных, называются листьями. На плоскости узлы удобно изображать точками, узлы i-ого уровня связывать дугами с порожденными узлами $(i+1)$-ого уровня.

Примером дерева может служить генеалогическое дерево, в котором порожденными узлами являются сыновья. На рисунке 1 изображено лишь два поколения потомков Александра Невского: сыновья и внуки.

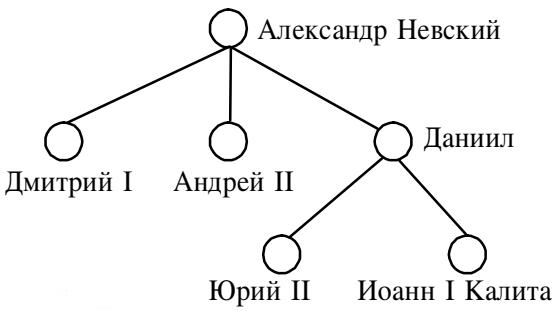
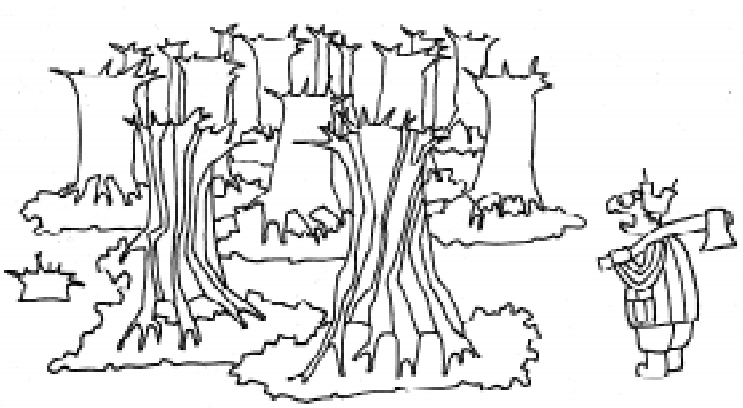


Рисунок 1



На самом высоком уровне иерархии ... располагается единственный узел, называемый корнем.

Можно построить генеалогическое дерево своей семьи, выбрав в качестве корня дерева, например, прадедушку. С помощью дерева можно представить родословную некоторого человека. В таком дереве каждый узел имеет два порожденных, например, левый узел хранит данные о матери, правый – об

Можно построить генеалогическое дерево своей семьи, выбрав в качестве корня дерева ... прадедушку.

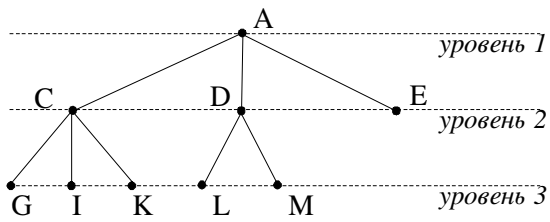


Рисунок 2

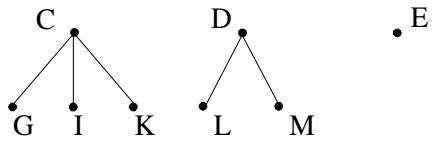


Рисунок 3

отце. На каком-то уровне информация о родственниках безвозвратно утеряна.

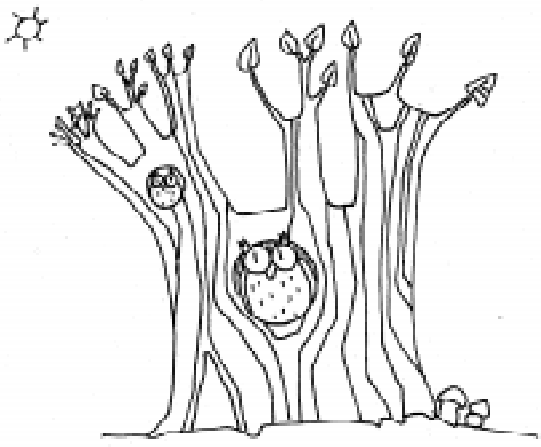
На рисунке 2 изображено дерево, узлы которого помечены буквами. Корнем этого дерева является узел *A*. Это единственный узел, расположенный на самом высоком уровне иерархии. Порожденные корнем узлы: *C*, *D*, *E*. Узел *A* для узлов *C*, *D*, *E* является исходным. Узлы *G*, *I*, *K*, *L*, *M*, *E* не имеют порожденных узлов и являются листьями. На рисунке 3 изображены три дерева. Последнее дерево состоит лишь из одного узла – корня *E*. Эти три дерева являются поддеревьями дерева, расположенного на рисунке 2. Корни этих поддеревьев связаны с единственным узлом *A*. Узел *A* расположен на более высоком уровне иерархии. Таким образом, порожденные узлы являются корнями поддеревьев данного дерева. При таком способе представления дерева достаточно иметь указатель на корень дерева для того, чтобы получить доступ к любой вершине дерева.

Дерево можно определить как частный случай графа, а именно, дерево – это связный граф без циклов. В [6] приведены примеры различных деревьев и способы их представления.

Приведем рекурсивное определение дерева. Деревом называется конечное множество элементов *D*, состоящее из одного или нескольких элементов, удовлетворяющее условиям:

1. Есть один выделенный элемент, называемый корнем дерева.
2. Остальные элементы множества *D* содержатся в *m* попарно непересекающихся множествах D_1, D_2,..., D_m, каждое из которых является деревом. Деревья D_1, D_2,..., D_m являются поддеревьями выделенного корня.

Дерево, изображенное на рисунке 2, можно представить в виде совокупности множеств (рисунок 4). Элемент *A* – корень дерева, остальные элементы содержатся в трех попарно непересекающихся множествах {*C*, *G*, *I*, *K*}, {*D*, *L*, *M*}, {*E*}. Каждое из множеств, в свою очередь, является деревом. Рассмотрим множество {*C*, *G*, *I*, *K*}. Элемент *C* – корень дерева, остальные элементы содержатся в трех по-



Приведем рекурсивное определение дерева...

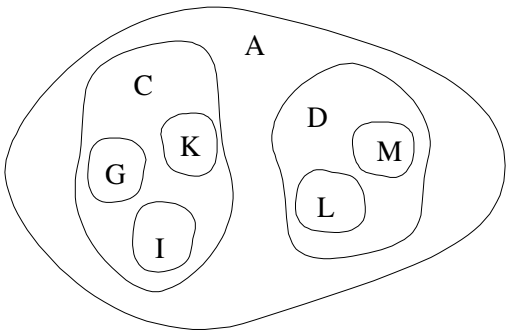


Рисунок 4

парно непересекающихся множествах $\{G\},\{I\},\{K\}$, каждое из которых – дерево, состоящее лишь из одного элемента – корня. Если узлы изобразить точками на плоскости, а корни дерева связать с корнями поддеревьев, то получим изображение, как на рисунке 2.

При работе с деревьями удобно определить пустое дерево как дерево, не содержащее узлов.

БИНАРНЫЕ ДЕРЕВЬЯ И ИХ ПРЕДСТАВЛЕНИЕ

Структуры данных и алгоритмы их обработки будут наиболее простыми в случае так называемых бинарных деревьев, то есть таких деревьев, каждый узел которых имеет не более двух порожденных узлов (левого и правого), и, соответственно, не более двух поддеревьев (левого и правого). На рисунке 5 представлены бинарные деревья. Деревья на рисунках 5в и 5г различны, так как в первом случае узел *D* является левым порожденным для *C*, а во втором – правым порожденным для *C*.

Один из способов представления деревьев применяется в случае, если известно, что каждый элемент дерева имеет не более k поддеревьев при небольшом значении k. Тогда можно включить в каждый элемент k указателей на поддеревья. Наиболее частый случай $k = 2$, при котором дерево называют бинарным, а два поддерева – левым и правым, соответственно. При таком способе представления достаточно иметь лишь указатель на корень дерева, чтобы получить доступ к любому элементу, спускаясь вниз по указателям. Отсутствующее дерево представляется пустым указателем.

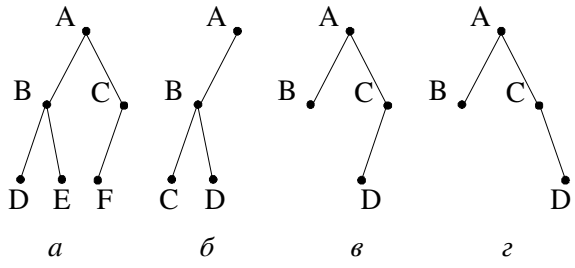


Рисунок 5

Узел дерева представляет собой запись, состоящую из трех полей, первое поле – информационное, второе и третье – указатели на левый и правый порожденные узлы (на корни левого и правого поддеревьев). Если элемент имеет тип **node**, то тип значения, называемого указателем на элемент **node**, записывается так: **^node**. В разделе определения типов можно ввести синоним для типа **^node**: **tree=^node.**

Описание типов данных для бинарного дерева может выглядеть так:

```
type elem_tree=char; tree=^node;
     node= record
         info:elem_tree; left,right:tree
     end
```

Заметим, что в описании сначала идет определение типа **tree**, так как тип tree используется при описании полей **left** и **right**. Заметим, что конструкция **^z** – единственный описатель типа, который может содержать еще не определенный в программе идентификатор типа **Z**. Если в программе переменная **t** описана следующим образом: **var t: tree**, то говорят, что **t** является указателем на элемент типа **node**. Переменная **t** может иметь значение **nil**. Это означает, что указатель не установлен ни на какой элемент. Переменная, на которую установлен указатель, обозначается **t^**, в нашем случае **t^** имеет тип **node**. Получить доступ к полям переменной **t^** можно следующим образом: **t^.info**, **t^.left**, **t^.right**.

Если в программе описаны две переменные **p** и **q** (**var p,q: tree**), то после выполнения оператора присваивания **p:=q** оба указателя **p** и **q** установлены на одну и ту же переменную, а именно на ту, на которую первоначально был установлен указатель **q**. Значение переменной типа указателя может измениться в результате выполнения оператора присваивания **p:=nil**. Значение переменной типа указателя

может измениться в результате выполнения стандартной процедуры **new(p)**, при выполнении которой выделяется память под значение типа **node**, и указатель p устанавливается на новый, созданный элемент с неопределенными значениями полей. Если память не может быть выделена, то значение p становиться равным **nil**.

Выполнение оператора присваивания **p^:= q^** приведет к тому, что, хотя указатели установлены на разные переменные, значения этих переменных одинаковы.

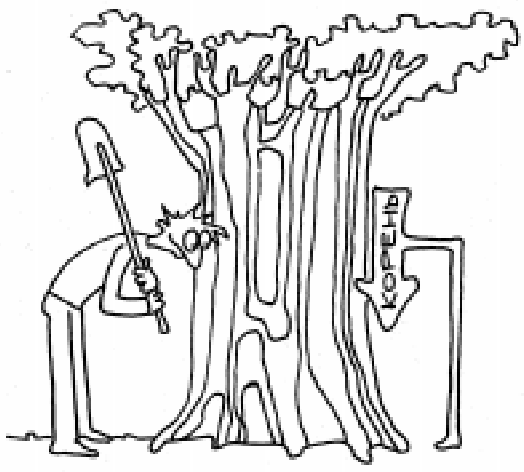
ОСНОВНЫЕ АЛГОРИТМЫ ОБРАБОТКИ БИНАРНЫХ ДЕРЕВЬЕВ

При обработке деревьев удобно использовать рекурсивные методы. Напомним, что при разработке рекурсивного алгоритма требуется обратить внимание на следующие моменты:

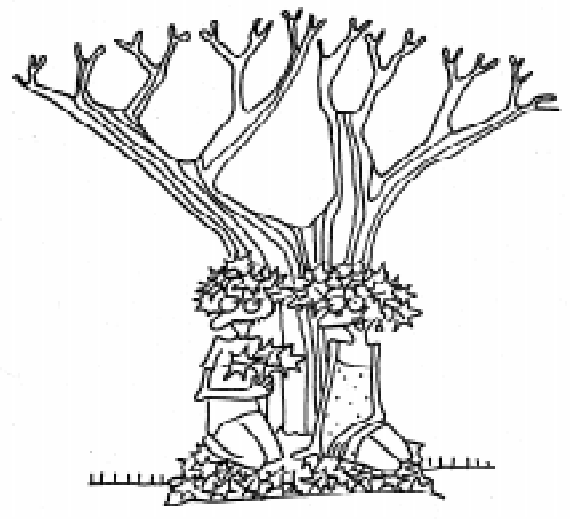
1. Определить параметры, от которых зависит решение задачи.
2. Решить задачу в тривиальном случае.
3. Свести задачу к ней самой, но с другими, более «простыми», параметрами.

АЛГОРИТМ ВЫЧИСЛЕНИЯ ЧИСЛА УЗЛОВ БИНАРНОГО ДЕРЕВА

При выбранном способе представления дерева для его задания достаточно иметь указатель на корень дерева, который и является единственным параметром. Если дерево пусто, то число его узлов равно 0. Это и есть решение задачи в тривиальном случае. Если же дерево не пусто, то число его узлов определяется как сумма числа узлов в левом поддереве и числа узлов в правом поддереве, увеличенная на единицу. Число узлов дерева можно определить рекурсивно следующим образом:



При выбранном способе представления дерева для его задания достаточно иметь указатель на корень дерева...



Опишите рекурсивный алгоритм вычисления количества листьев бинарного дерева.

nnode(t)=1+nnode(tl)+ nnode(tr),

где t – исходное дерево, tl, tr – левое и правое поддеревья. Таким образом, решение задачи о числе узлов дерева сведено к решению той же самой задачи, но уже применительно к поддеревьям данного дерева. В каждом из поддеревьев число узлов меньше, чем в дереве. Именно в этом смысле понимаются в данном случае более «простые» параметры.

Опишем функцию, которая для бинарного дерева, заданного указателем на корень, определяет количество узлов в дереве.

```
function nnode (t: tree): integer;
begin
  if t=nil
  then nnode=0
  else  nnode=1+nnode(t^.left)+
        nnode(t^.right)
end
```

Задача 1.

Уровень 1.

1) Опишите рекурсивный алгоритм вычисления количества листьев бинарного дерева.

В терминах генеалогического дерева – это количество членов рода, не имеющих (или не имевших) сыновей. В терминах дерева родословной – это те предки, про обоих родителей которых нам ничего не известно.

2) Опишите рекурсивный алгоритм вычисления количества ребер бинарного дерева. В терминах генеалогического дерева – это число сыновей в роду, в терминах дерева родословной – число известных предков.

ОПРЕДЕЛЕНИЕ ВЫСОТЫ ДЕРЕВА

Высотой дерева называется максимальный из уровней всех узлов дерева. Высота пустого дерева равна 0, высота дерева, состоящего лишь из корня, равна 1, высота дерева, изображенного на рисунке 6, равна 5. На рисунке 7 изображено дерево, где через *A* и *B* обозначены поддеревья, которые в общем случае могут быть и пустыми. При построении алгоритма можно рассуждать и так: как найти высоту дерева *T*, если известны высоты поддеревьев *A* и *B*? Для того чтобы найти высоту дерева *T*, надо взять максимальную из высот поддеревьев *A* и *B* и увеличить на 1.

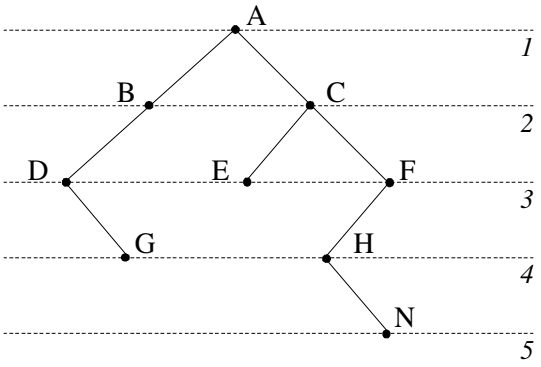


Рисунок 6

Рекурсивно высоту бинарного дерева *t* опишем так:

$h(t)=\max(h(tl),h(tr))+1,\ h(\text{null})=0,$

где *t* – исходное дерево, *tl*, *tr* – левые и правые поддеревья, null – пустое дерево.

ПОИСК ЭЛЕМЕНТА В ДЕРЕВЕ

Рассмотрим решение задачи поиска элемента в дереве. При решении этой задачи используются два параметра, один параметр задает дерево, второй – тот элемент, который требуется найти в дереве. Как и ранее, тривиальное решение будет в случае, когда дерево пусто. В таком дереве нет узлов, следовательно, нет интересующего нас узла. Если дерево не пусто, то исследуется корень дерева. Если элемент, расположенный в корне дерева, совпадает с заданным элементом, то задача решена. В противном случае поиск элементов следует производить в поддеревьях *A* и *B* (рисунок 7). Порядок просмотра поддеревьев произволен, если это не оговорено особо. Просматриваем, например, поддерево *A*. Если требуемый элемент найден, то задача решена. Если же элемент в поддереве *A* не обнаружен, то поиск следует продолжать в поддереве *B*. В этом случае результат поиска во всем дереве зависит от того, найден или нет элемент в поддереве *B*.

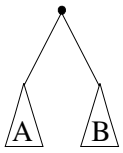
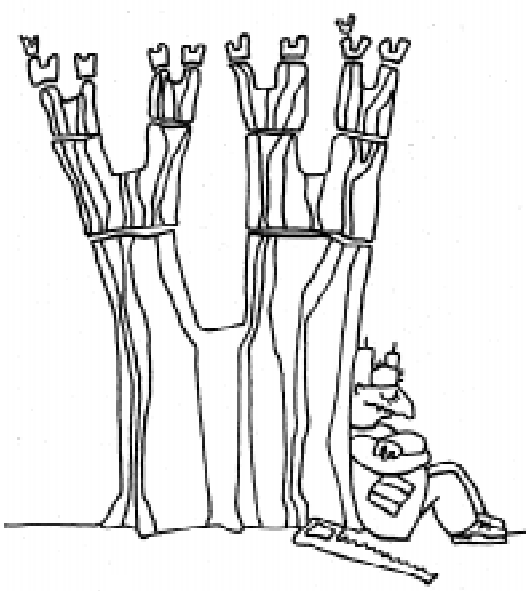


Рисунок 7

Задача 2.

Уровень 1. Опишите рекурсивный алгоритм, определяющий число элементов в бинарном дереве, равных заданному.

Уровень 2. Напишите рекурсивную процедуру определения числа узлов, находящихся на заданном уровне.



Опишите рекурсивный алгоритм, определяющий число элементов в бинарном дереве, равных заданному...

Формат ввода:

Заданный уровень k

Количество узлов в дереве n

1 ЛП1 ПП1

2 ЛП2 ПП2

...

n ЛПn ППn

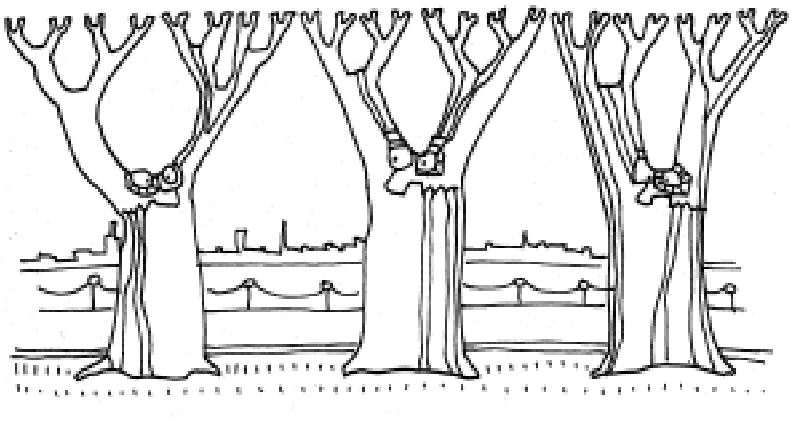
Примечание. ЛП – ссылка на левое поддерево, ПП – ссылка на правое поддерево, пустые поддеревья обозначаются 0.

Формат вывода:

Число узлов на уровне k

Пример.

Дерево изображено на рисунке 8.



Напишите рекурсивную процедуру определения числа узлов, находящихся на заданном уровне...

Ввод:

```
2
7
1 2 3
2 4 5
3 6 7
4 0 0
5 0 0
6 0 0
7 0 0
```

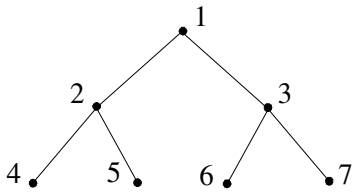


Рисунок 8

Вывод:

```
2
```

ОБХОД ДЕРЕВА

Одна из распространенных задач при работе с деревьями – выполнение некоторого действия p с каждым элементом дерева. Часто такую задачу называют задачей обхода дерева. При работе с деревьями операций [7] приводились алгоритмы обхода, позволяющие по дереву операций получить префиксную и постфиксную нотацию формул.

В качестве примера приведем левосторонний обход дерева. При левостороннем обходе сначала полностью обходится левое поддерево корня (если поддерево существует), затем действие p применяется к корню дерева, а затем обходится правое поддерево. При этом поддерево обходится левосторонним способом, то есть описанный алгоритм является рекурсивным.

Опишем алгоритм нахождения всех узлов дерева, расположенных на i-ом уровне ($i > 0$). В этом случае задача имеет два параметра, один задает бинарное дерево, второй – номер уровня, на котором ищем вершины. Если дерево пусто, то вершин i-ого уровня в нем нет. Если же дерево не пусто и значение i равно 1, то элемент i-ого уровня – это корень дерева. Сведем решение задачи к ней самой, но с другими параметрами. Заметим, что все узлы i-ого уровня дерева t являются узлами

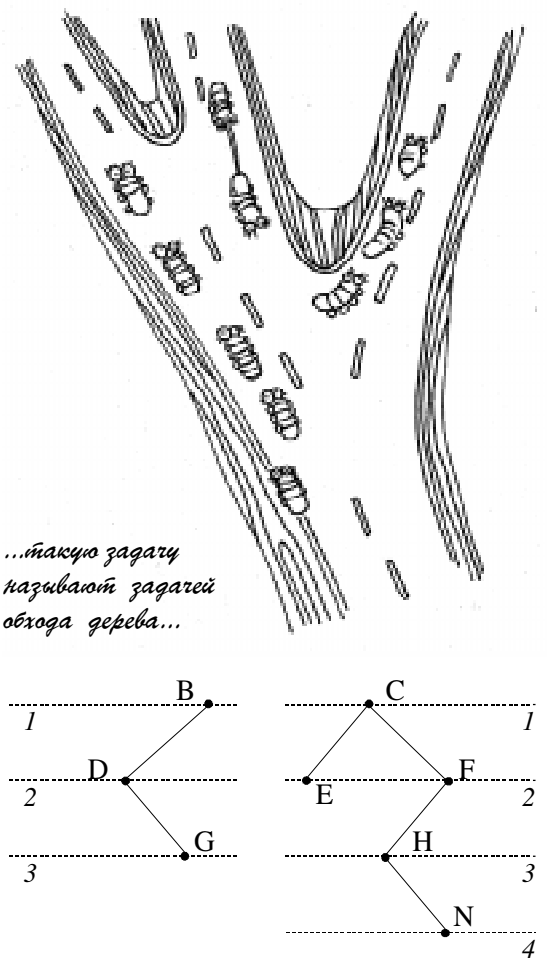


Рисунок 9

(i –1)-ого уровня для левого и правого поддеревьев дерева t. Например, узлы D, E, F являются узлами третьего уровня для дерева на рисунке 6 и узлами второго уровня для левого и правого поддеревьев, изображенных на рисунке 9.

Задача 3.

Уровень 2. После многих часов, проведенных в библиотеке, Виви составила свою родословную. Она получилась настолько обширной, что Виви запуталась в отношениях родства между своими предками. Помогите Виви разобраться в ее родословной. Для этого напишите программу, которая выписывает для двух предков Виви их отношение родства. Родословная задана бинарным деревом, причем для каждого человека в корне левого поддерева находится его мать, а в корне правого – отец.

Формат ввода:

n (количество узлов дерева без учета корня)
1 имя 1 ЛП1 ПП1
2 имя 2 ЛП2 ПП2
...
n имя n ЛПn ППn
Имя предка 1
Имя предка 2

Формат вывода:

Отношение родства между предком 1 и предком 2.

Пример.

Примечание. Для описания следует использовать только слова «мать, матери, сын, сына, отец, отца, дочь, дочери». (Если имен, приведенных в примере окажется недостаточно, можно использовать следующие: Васио, Петио, Толио, Борио, Мими, Диди, Лулу, Зузу и т.д.) Дерево изображено на рисунке 10.

Ввод:

6
1 Люси 3 4
2 Педро 5 6
3 Мари 0 0
4 Антонио 0 0
5 Лили 0 0
6 Пабло 0 0
Люси
Лили

Вывод:

Люси = мать дочери сына Лили

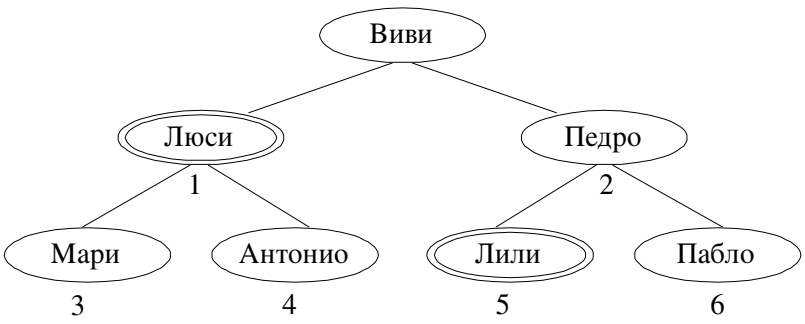
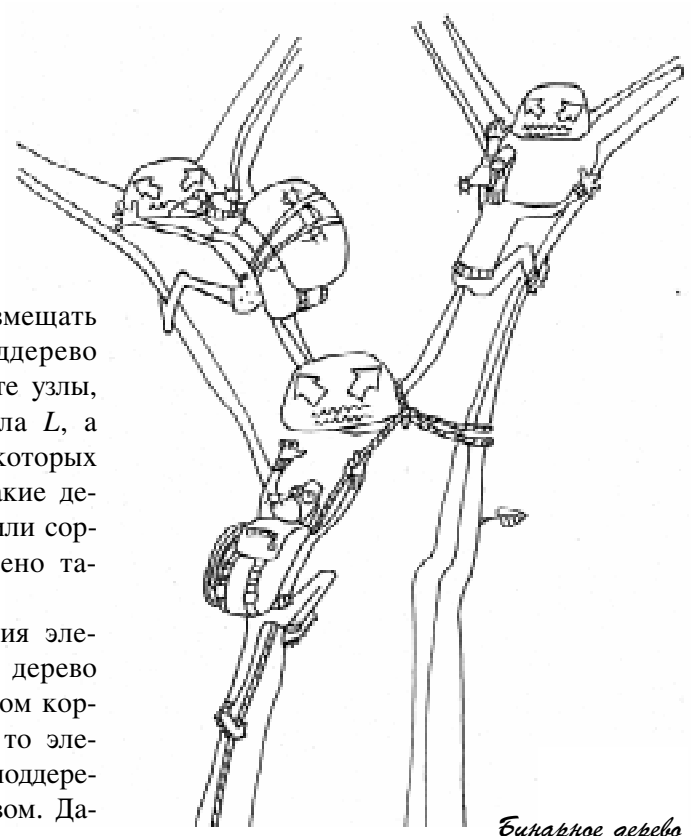


Рисунок 10

БИНАРНОЕ ДЕРЕВО ПОИСКА

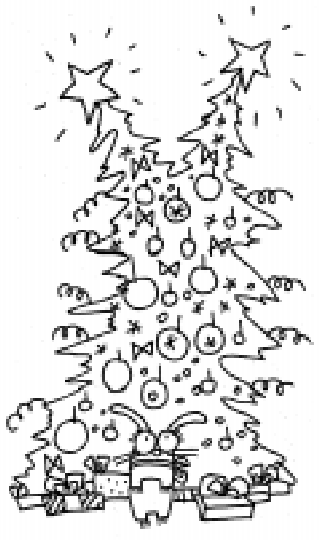
Бинарное дерево удобно использовать для быстрого поиска данных. Будем считать, что элементы, которые будут организованы в бинарное дерево, снабжены числовым признаком, который назовем весом элемента. Элементы в дереве будем размещать таким образом, чтобы левое поддерево любого узла *L* содержало только те узлы, вес которых меньше, чем вес узла *L*, а правое поддерево – те узлы, вес которых больше или равен весу узла *L*. Такие деревья называют деревьями поиска или сортировки. На рисунке 11 изображено такое дерево.



Бинарное дерево удобно использовать для быстрого поиска данных...

Опишем алгоритм добавления элемента с заданным значением *z* в дерево поиска *T*. Сравнивается вес *z* с весом корня дерева *T*. Если вес *z* меньше, то элемент следует разместить в левом поддереве *T*, в противном случае – в правом. Далее поиск места для размещения элемента повторяется рекурсивно. Добавляемый элемент образует новый лист дерева.

При добавлении элементов со значениями 3 и 5 в дерево поиска, изображенное на рисунке 11, дерево примет вид, показанный на рисунке 12.



При добавлении элементов ... дерево примет вид, как показано на рисунке...

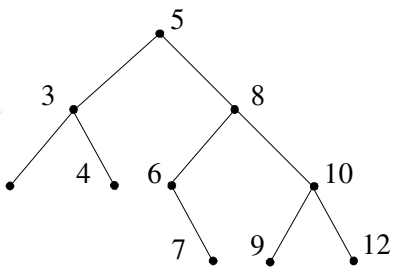


Рисунок 11

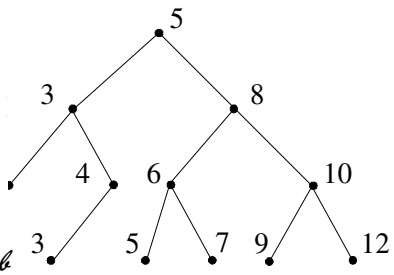


Рисунок 12

Опишем процедуру добавления в бинарное дерево поиска *T* элемента *z*. Считаем, что тип информационного поля элемента дерева – целый

```
(elem_tree=integer)
procedure add_tree
  (var t: tree; z: elem_tree);
  var p:tree;
begin if t=nil
   then begin new(t);
        t^.info := z;
        t^.left := nil;
        t^.right := nil
        end
   else if z < t^.info
    then add_tree (t^.left,z)
    else add_tree (t^.right,z)
end
```

При левостороннем обходе бинарного дерева поиска элементы дерева образуют возрастающую последовательность, левосторонний обход дерева на рисунке 12 обеспечит обработку вершин в порядке: 2 3 3 4 5 5 6

7 8 9 10. При правостороннем обходе значения будут расположены в порядке убывания.

Для деревьев поиска легче, чем для обычных бинарных деревьев, решается задача поиска. На каждом шаге известно поддерево, в котором следует продолжать поиск.

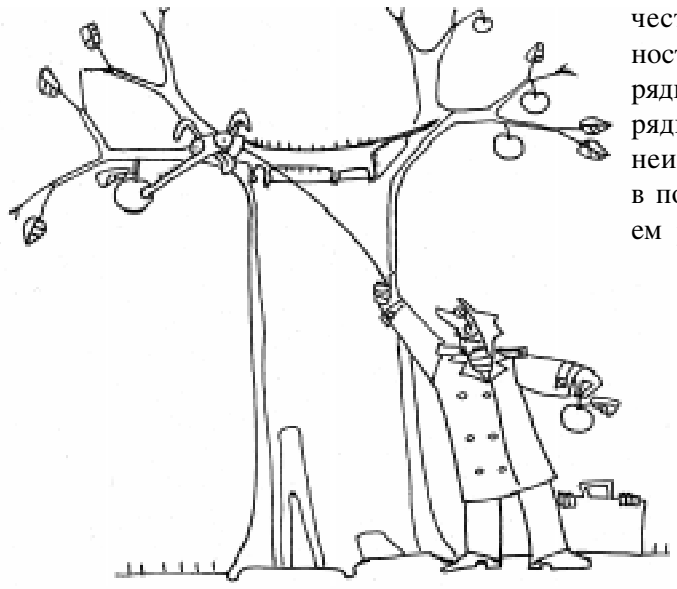
ПОИСК С ВКЛЮЧЕНИЕМ

При решении многих задач используется алгоритм, который получил название *Поиск с включением*. Согласно этому алгоритму элемент добавляется в дерево поиска лишь в тех случаях, если такого элемента в дереве не было. Таким образом, все элементы дерева различны.

Задача 4.

Уровень 1. Опишите два алгоритма (рекурсивный и нерекурсивный) поиска элемента с заданным весом в дереве поиска.

Уровень 2. Задана последовательность ненулевых целых чисел, число 0 – признак конца. Напишите программу, которая формирует последовательность чисел в порядке убывания, причем каждое число встречается в этой последовательности лишь один раз.



Опишите два алгоритма ... поиска элемента с заданным весом в дереве поиска...

Указание. По заданной последовательности постройте дерево поиска, используя процедуру, реализующую алгоритм поиска с включением. Затем осуществите правосторонний обход построенного дерева.

Формат ввода:

Последовательность натуральных чисел, завершаемая нулем

Формат вывода:

Отсортированная в порядке убывания последовательность без повторений

Пример.

Ввод:

2 9 8 6 2 8 5 0

Вывод:

9 8 6 5 2

Задача 5.

Рассмотрим задачу, которая является упрощенным вариантом задачи о частотном словаре. Вместо анализа слов некоторого текста, будут анализироваться числа некоторой последовательности.

Во входном потоке задана последовательность ненулевых целых чисел, число 0 считается признаком конца последовательности. Требуется написать программу, которая формирует две таблицы. В каждой из таблиц хранится число и количество его вхождений в последовательность. Первая таблица упорядочена в порядке возрастания чисел, вторая – в порядке убывания частот вхождения. Нам неизвестно количество различных чисел в последовательности, поэтому мы не знаем размера таблиц.

Можно поступить следующим образом: все числа последовательности организовать в бинарное дерево поиска. Для каждого числа проверять, встречалось ли оно ранее в последовательности. Если число встретилось впервые, то в дерево включить узел, одно из полей которого совпадает с числом, второе поле характеризует частоту вхождения и равно в данном случае 1. Если же рассматриваемое число уже в последовательности

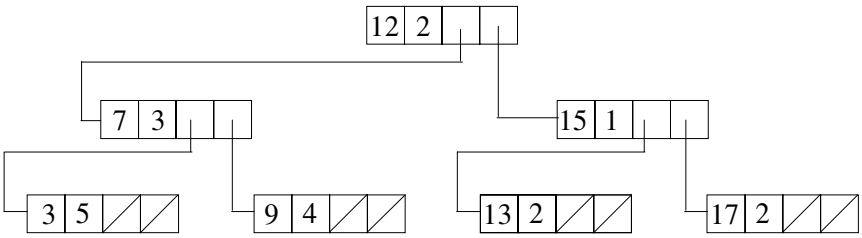


Рисунок 13

встречалось, то в дереве есть соответствующий этому числу узел. В этом случае узел надо найти и частоту вхождения увеличить на 1.

После просмотра всей последовательности будет построено дерево поиска, количество узлов которого равно количеству различных чисел в последовательности. С каждым числом хранится количество его вхождений в последовательность. Осуществляя левосторонний обход дерева, получим таблицу, числа в которой упорядочены по возрастанию. Рассмотрим последовательность чисел: 12 7 3 15 3 9 7 12 13 7 3 9 17 9 13 3 17 3 9 0. После просмотра всей последовательности будет построено дерево T, изображенное на рисунке 13. В каждом узле дерева первое число – элемент последовательности, второе число – частота вхождения в последовательность. При применении левостороннего обхода к построенному дереву T будут выведены значения, представленные в таблице 1.

Таблица 1

Элемент последовательности	Частота вхождения
3	5
7	3
9	4
12	2
13	2
15	1
17	2

Для того чтобы построить таблицу в порядке убывания частот, можно по дереву поиска построить дерево, элементы которого упорядочены по частоте вхождения, а затем применить к нему процедуру правостороннего обхода.

При левостороннем обходе дерева T в качестве процедуры обработки узла возьмем процедуру добавления узла в дерево поиска L, так чтобы дерево L было упорядочено по частоте вхождения элементов в последовательность. Дерево изображено на рисунке 14.

Применяя правосторонний обход к дереву L, получим следующую таблицу 2, элементы которой упорядочены в порядке убывания частот.

Таблица 2

Элемент последовательности	Частота вхождения
3	5
9	4
7	3
17	2
13	2
12	2
15	1

Применим правосторонний обход дерева T и построим дерево поиска L_1, элементы которого упорядочены по частоте вхождения. Дерево L_1 представлено на рисунке 15. Применяя к дереву L_1 пра-

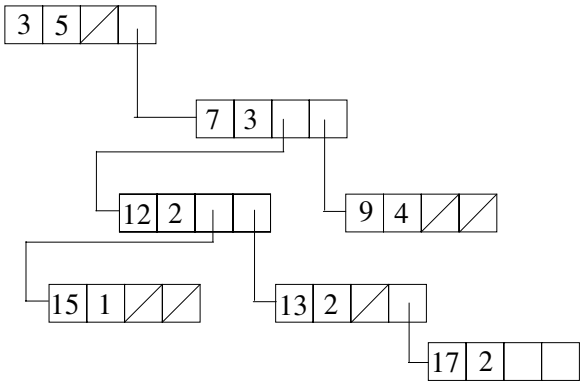


Рисунок 14

восторонний обход, получим таблицу 3, которая упорядочена в порядке убывания частот, но отлична от таблицы 2.

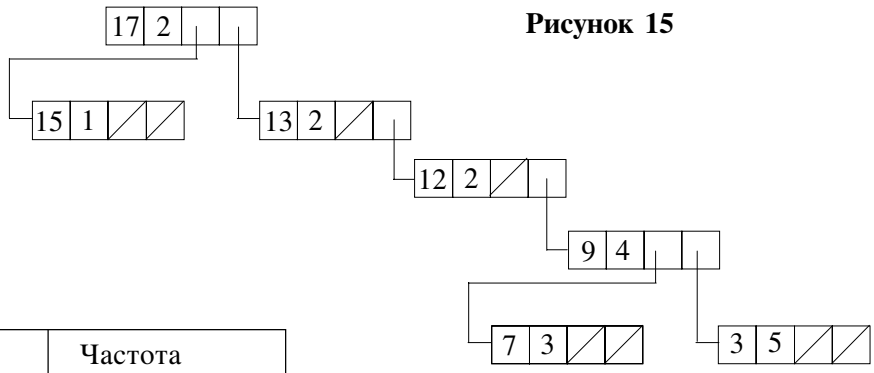


Рисунок 15

Таблица 3

Элемент последовательности	Частота вхождения
3	5
9	4
7	3
12	2
13	2
17	2
15	1

Уровень 2. Напишите программу, которая по последовательности чисел формирует две таблицы и реализует описанный алгоритм.

Формат ввода:

Последовательность натуральных чисел, завершаемая нулем

Формат вывода:

Две таблицы построчно и поочередно

Пример.

Ввод:

12 7 3 15 3 9 7 12 13 7 3 9 17 9 13 3 17 3 9 0

Вывод:

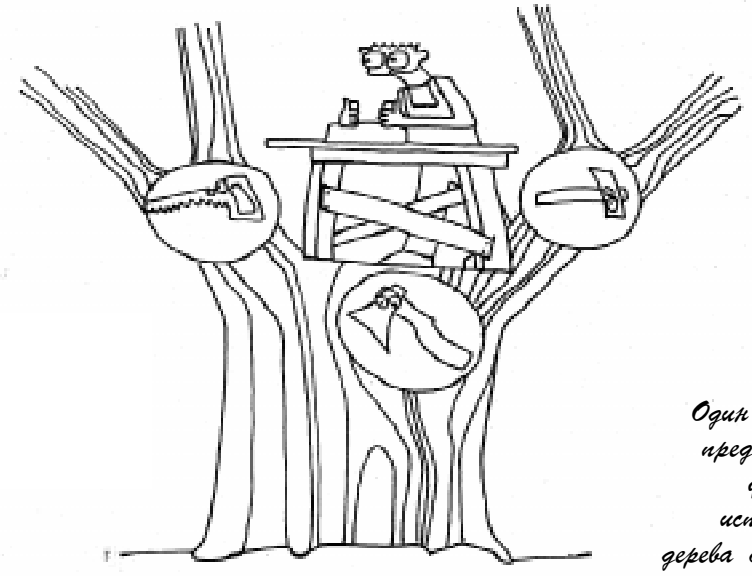
3	5
7	3
9	4
12	2
13	2
15	1
17	2
3	5
9	4
7	3
17	2
13	2
12	2
15	1

ДЕРЕВО ОПЕРАЦИЙ

С помощью бинарного дерева удобно представлять различные формулы. В [7] рассматривались различные способы представления формул и некоторые алгоритмы их обработки. Один из способов представления формул – использование дерева операций. Формула просматривалась слева направо, и с помощью стека строилось дерево операций. Напомним, что в дереве операций каждый узел (кроме листьев) соответствует операции, а левое и правое поддеревья – операндам.

Опишем алгоритм построения дерева операций по формуле, представленной строкой. Формула просматривается слева направо один раз.

Будем считать, что формула состоит из однобуквенных идентификаторов (букв латинского алфавита), круглых скобок и



Один из способов представления формул — использование дерева операций...

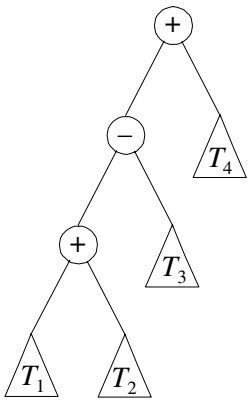


Рисунок 16

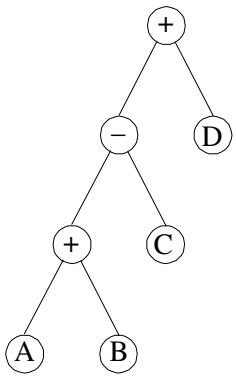


Рисунок 17

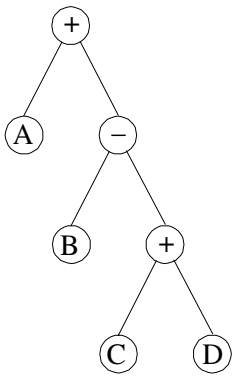


Рисунок 18

операций (+, –, *, /). Для анализа формулы нам удобно ввести следующие понятия, характеризующие отдельные части формулы и формулу в целом. Будем называть *множителем* либо идентификатор переменной либо формулу, заключенную в круглые скобки. Назовем *термом* либо один множитель либо последовательность множителей, соединенных знаками «*» и «/», *формулой* – либо один терм либо последовательность термов, соединенных знаками «+» и «–».

Опишем процедуру Form с одним параметром *t*, действие которой состоит в том, что по просмотренной формуле процедура строит соответствующее ей дерево операций со ссылкой на корень *t*. По данному нами определению, формула всегда начинается с терма. Будем считать, что процедура Term(t_1) строит дерево для части формулы, являющейся термом, и t_1 – ссылка на корень построенного дерева. При анализе формулы в результате исполнения вызова Term(t_1) будет построено дерево, соответствующее первому терму формулы со ссылкой на корень t_1. Если в формуле следующий за первым термом символ отличен от знака операции «+» или «–», то это означает, что формула состоит из одного терма, и анализ формулы завершен. Если же вслед за первым термом идет знак операции «+» или «–», то требуется продолжать анализ формулы. Далее формируется узел дерева для знака операции. В качестве левого поддерева надо взять дерево, соответствующее первому терму, с помощью процедуры Term построить дерево для следующего терма – формулы и взять его в качестве правого поддерева узла, соответствующего знаку операции, и т.д. Если формула имеет вид $T_1+T_2–T_3+T_4$, то в результате работы процедуры Form должно быть построено дерево, показанное на рисунке 16. Процедура Form по формуле $A+B–C+D$ построит дерево, изображенное на рисунке 17, а по формуле $A+(B–(C+D))$ – дерево, изображенное на рисунке 18.

При решении этой задачи тип элемента дерева символьный (**elem_tree=char**). Опишем процедуру, которая строит дерево с указателем **t** на его вершину для формулы. Первый элемент уже выбран. Текущий символ хранится в переменной **c**.

```
procedure Form (var t: tree);
 var p,t1,t2: tree;
begin
 Term(t1);
 {построено дерево для первого терма}
 while (c='+') or (c='-') do
   begin new(p);
      p^.info := c;
      {строим узел для операции}
      p^.left := t1;
      c := cursym();
      {выбрали следующий символ}
      Term(t2);
     {построено дерево для второго терма}
      p^.right :=t2;
      {подцепили к узлу операции}
      t1 := p
   end;
   t := t1
end
```

Теперь опишем процедуру Term. Нам требуется, чтобы процедура Term умела по соответствующему терму строить дерево с указателем на корень *t*. Мы определили терм так, что он всегда начинается с множителя. Будем считать, что процедура Mn(*t*) строит дерево операций для части формулы, определенной как мно-

житель, и формирует указатель *t* на корень построенного дерева. Процедура Term похожа на процедуру Form.

Процедура Mn анализирует часть формулы, определенной как множитель, строит дерево, соответствующее проанализированной части формулы, и выбирает следующий за множителем символ. Считаем, что первый символ множителя уже выбран. Множителем является либо идентификатор, либо формула, заключенная в скобки. Если очередной символ – буква (множитель в данном случае – это идентификатор), то строится дерево *t* с одним узлом, и для анализа выбирается следующий символ формулы.

Если же очередной символ – открывающая круглая скобка, то требуется проанализировать случай, когда множитель представляет собой формулу, заключенную в скобки. Следует воспользоваться процедурой Form, которая умеет строить дерево с указателем на корень *t* для любой формулы. Итак, в этом случае множитель имеет вид (*f*), и мы предположили, что в результате работы процедуры Form построено дерево со ссылкой на корень *t*, соответствующее формуле *f*. После работы процедуры Form выбран следующий за формулой символ. Если этот символ – закрывающая скобка, то анализ множителя завершен, осталось выбрать только следующий за скобкой символ для продолжения анализа формулы. Если же проанализирована формула *f*, а за ней символ, отличный от закрывающей скобки, то нарушен баланс скобок, возникла ошибочная ситуация. Если же окажется, что очередной символ не буква и не открывающая скобка, то должна быть зафиксирована ошибка.

Задача 7.

Уровень 2. Напишите программу, которая по формуле, представленной строкой, строит дерево операций. По дереву операций сформируйте формулу в постфиксной записи.

Формат ввода:

Формула со знаками операций «+», «–», «*», «/» и операндами, представленными буквами латинского алфавита.

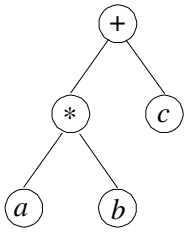
Формат вывода:

Формула в постфиксной записи.

Задача 8.

Одну и ту же формулу можно записать разными способами, расставляя в ней скобки или опуская их согласно приоритетам операций. Например, формула *a*b+c* может быть записана одним из способов: (*a*b*)+*c*, ((*a*b*)+*c*), ((*a*)*(*b*)+(*c*)).

Можно считать, что все формулы различны, но эквиваленты между собой. Пару скобок назовем избыточной, если после ее удаления формула остается эквивалентной исходной. Три приведенные формулы содержат избыточные скобки. Каждой из четырех формул соответствует дерево операций, приведенное на рисунке 19.



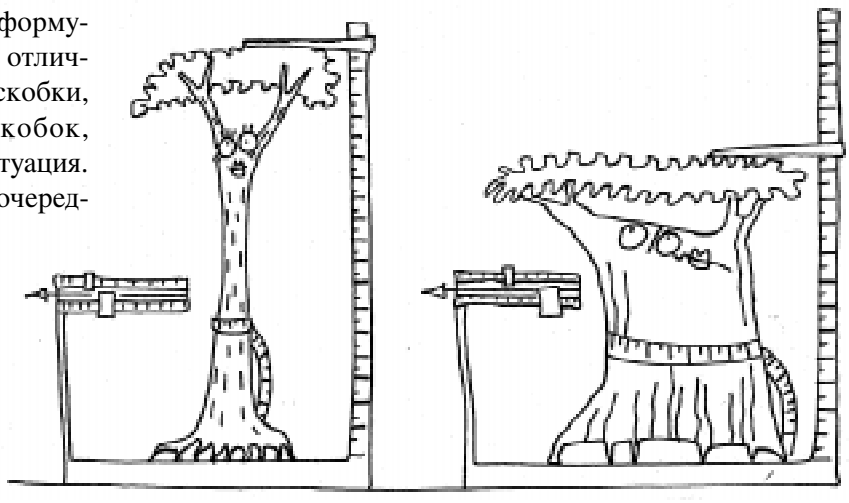


Рисунок 19

Можно считать, что все формулы различны, но эквиваленты между собой...

Уровень 2. Пусть заданы две формулы f_1 и f_2. Напишите программу, проверяющую их эквивалентность.

Указание. Можно поступить следующим образом: по каждой из формул построить дерево операций, затем сравнить построенные деревья. Если формулы отличаются лишь избыточными скобками, то деревья операций у них должны совпадать.

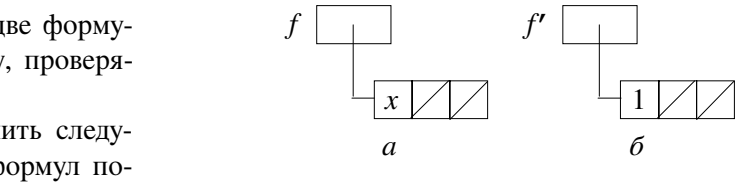
Формат ввода:
Формула 1
Формула 2

Формат вывода:
Да/нет

Пример.

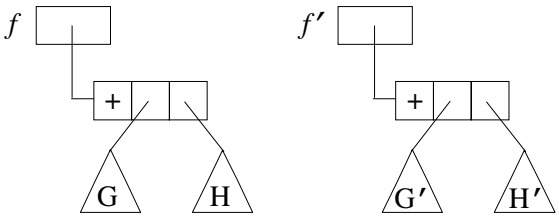
Ввод:
(*a***b*)+*c*
((*a***b*)+*c*)

Вывод:
Да



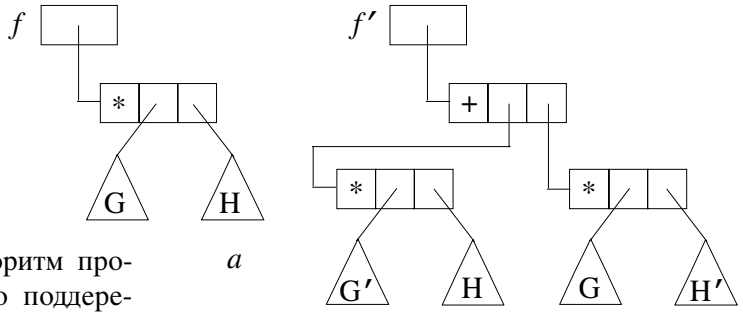
а *б*

Рисунок 20



a *b*

Рисунок 21



a *b*

Рисунок 22

Задача 9.

Уровень 1. Опишите алгоритм проверки, является ли одно дерево поддеревом другого.

Задача 10.

Немного усложним задачу сравнения формул. Будем рассматривать формулы, состоящие из малых латинских букв, круглых скобок, знаков операций «+» и «*», обладающих свойством коммутативности (перестановочности). Формулы будем считать эквивалентными, если на каждом уровне скобок они отличаются лишь порядком своих операндов. Следующие формулы эквивалентны: (*a*+*b*) и (*b*+*a*), ((*a***b*)+*c*) и (*c*+(*b***a*)).

Уровень 2. Напишите программу, проверяющую эквивалентность формул с учетом коммутативности операций.

Указание. Для того чтобы проверить, эквивалентны ли такого вида формулы, можно построить по ним деревья операций и далее сравнивать полученные деревья.

Формат ввода-вывода такой же, как в задаче 8.

АЛГОРИТМ СИМВОЛЬНОГО ДИФФЕРЕНЦИРОВАНИЯ*

Рассмотрим формулу, в которой встречаются целые константы (для простоты возьмем от 0 до 9), переменные, круглые скобки и знаки операций «+» и «*». Требуется по заданной формуле построить формулу, являющуюся ее производной по некоторой переменной. Для решения этой задачи представим формулу деревом операций. В результате действий над деревом операций, представляющим формулу, построим дерево операций для производной заданной формулы.

Если формула *f* представляет собой константу, то ее производная равна нулю. Если формула состоит из одной переменной *x*, по которой находится производная, то по дереву операндов (рису-

* Для учащихся 10–11 классов.

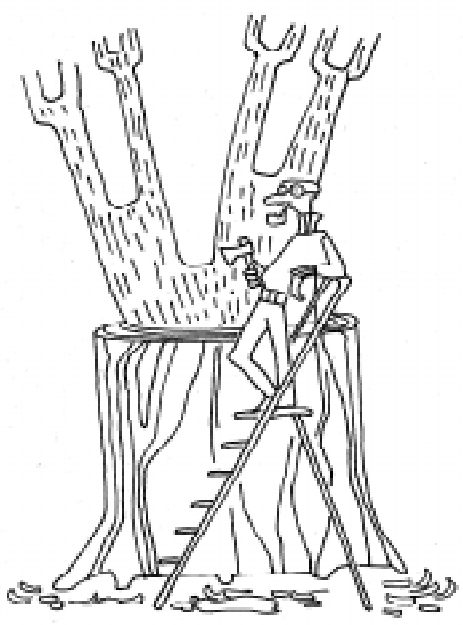
нок 20a) должно быть построено дерево (рисунок 20b).

Пусть теперь формула имеет вид: $g+h$. Тогда соответствующее формуле дерево операций изображено на рисунке 21a, где G, H – деревья операций для формул g и h соответственно. Так как производная в этом случае равна $g'+h'$, то деревья операций для производных должны быть такими, как на рисунке 21b, где G', H' – деревья операций для формул g', h'. Для того, чтобы найти производную формулы f, требуется найти производные для составляющих ее формул g и h. Задача свелась к ней самой, но с более простыми параметрами.

Нам осталось рассмотреть случай, когда формула f имеет вид $g*h$. Дерево для такой формулы представлено на рисунке 22a. Производной является формула вида $g'*h + g*h'$. Ей соответствует дерево операций на рисунке 22б, где G, H, G', H' – деревья операций для формул g, h, g', h', соответственно. Для того, чтобы строить производную в этом случае, надо уметь строить копию дерева операций.

Задача 11.*

Уровень 2. Напишите программу, которая по формуле строит ее производную.



Напишите программу, которая по формуле строит ее производную...

Указание. По формуле построить дерево операций, по нему построить дерево операций для производной, далее построить формулу-производную.

Формат ввода:

Формула, содержащая переменные, представленные строчными буквами латинского алфавита, константы от 0 до 9, знаки операций +, *, круглые скобки.

Формат вывода:

Формула-производная.

Пример.

Ввод:

$x+x*x$

Вывод:

$1+2*x$

УПРОЩЕНИЕ ФОРМУЛ

Пусть задана формула $f=(a*x+b)*(x-2)$, где x – переменная, a, b – константы. Этой формуле будет соответствовать дерево операций, изображенное на рисунке 23.

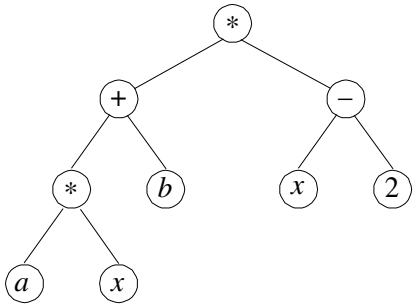


Рисунок 23

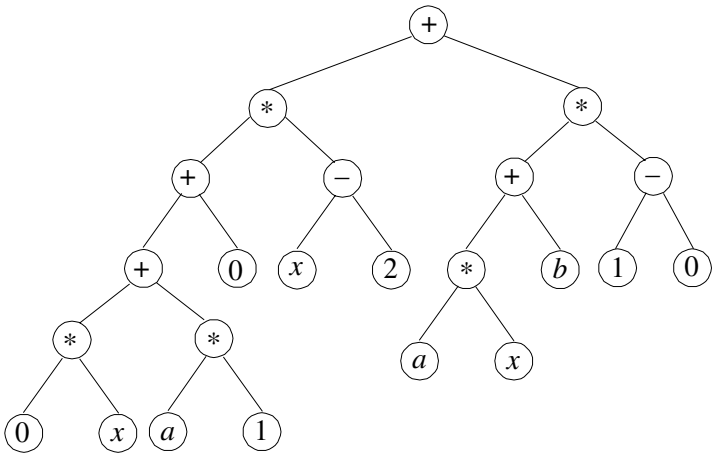


Рисунок 24

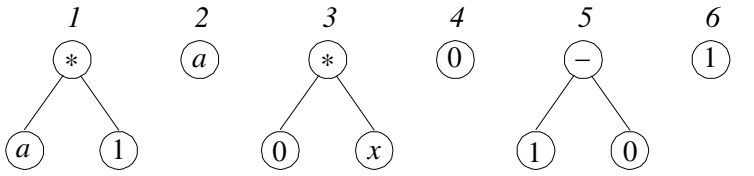


Рисунок 25

Дерево, соответствующее производной по *x*, представлено на рисунке 24. Уже на этом примере видно, что построенная формула может быть значительно упрощена, если выполнить некоторые очевидные действия. Рассмотрим, поддеревья, показанные на рисунке 25. Поддерево 1 может быть преобразовано в поддерево 2, поддерево 3 – в поддерево 4, поддерево 5 – в поддерево 6.

Следующая наша цель – выполнить описанные преобразования и, тем самым, сократить полученное дерево операций.

Задача 12.

Уровень 1. Опишите алгоритм, позволяющий выполнять частичные вычисления для формул, представленных деревом операций и содержащих константы.

Уровень 2. На основе предложенного алгоритма напишите программу частичных упрощений формулы.

Формат ввода:

Формула, содержащая переменные, представленные строчными буквами латинского алфавита, константы от 0 до 9, знаки операций +, *, –, круглые скобки.

Формат вывода:

Упрощенная формула.

Пример.

Ввод:

*(a+x)**1+*b**0–0

Вывод:

a+x

Например, после выполнения действий с константами формула, о которой говорилось выше, будет упрощена, дерево операций, представляющее упрощенную формулу, изображено на рисунке 26.

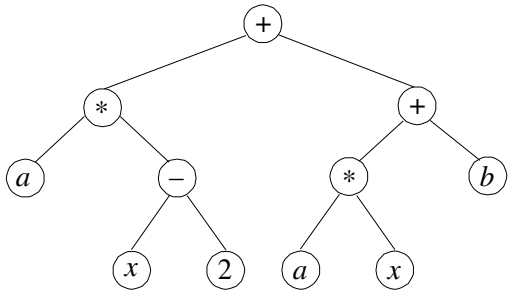
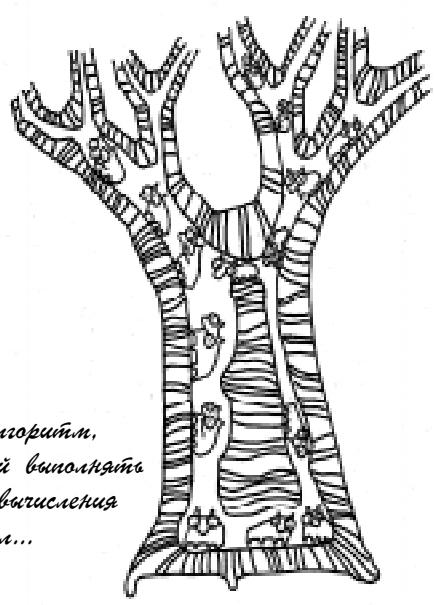


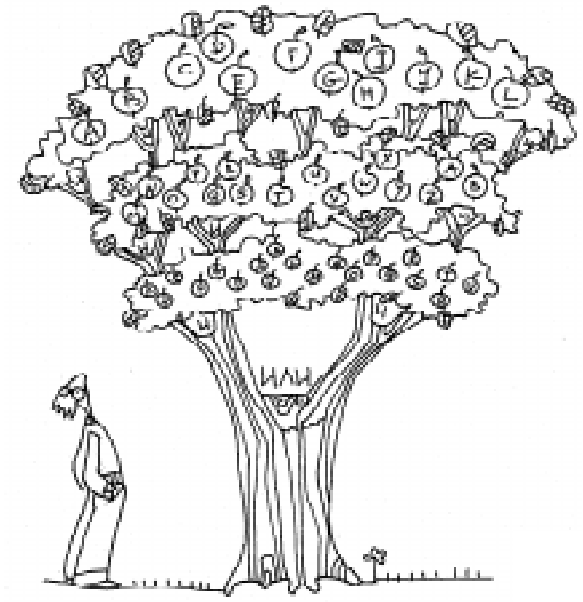
Рисунок 26



Задача 13.

Уровень 2. Задано логическое выражение, состоящее из переменных, круглых скобок, знаков логических операций "|" (или), "&" (и), "~" (отрицание), логических констант И (истина) и Л (ложь). Имя любой переменной задается одной латинской буквой. Вычисление выражения производится согласно приоритетам операций. Операция отрицания имеет наивысший приоритет, приоритет операции "&" выше приоритета операции "|". Требуется упростить выражение, произведя константные вычисления и удалив лишние скобки. Упрощения должны производиться согласно правилам, приведенным в таблице, через *a* обозначено произвольное выражение.

.



Задано логическое выражение...

Исходное выражение	После замены
a | Л	*a*
a | И	И
a & Л	Л
a & И	*a*
~~*a*	*a*
a & ~*a*	Л
a | ~*a*	И
~И	Л
~Л	И

Программа должна применять указанные правила к заданному выражению до тех пор, пока это возможно. Порядок применения правил может быть произвольным.

Формат ввода:

Выражение, состоящее из строчных букв латинского алфавита, круглых скобок, знаков логических операций «|» (или), «&» (и), «~» (отрицание), логических констант И (истина) и Л (ложь).

Формат вывода:

Упрощенное выражение.

Пример.

Ввод:

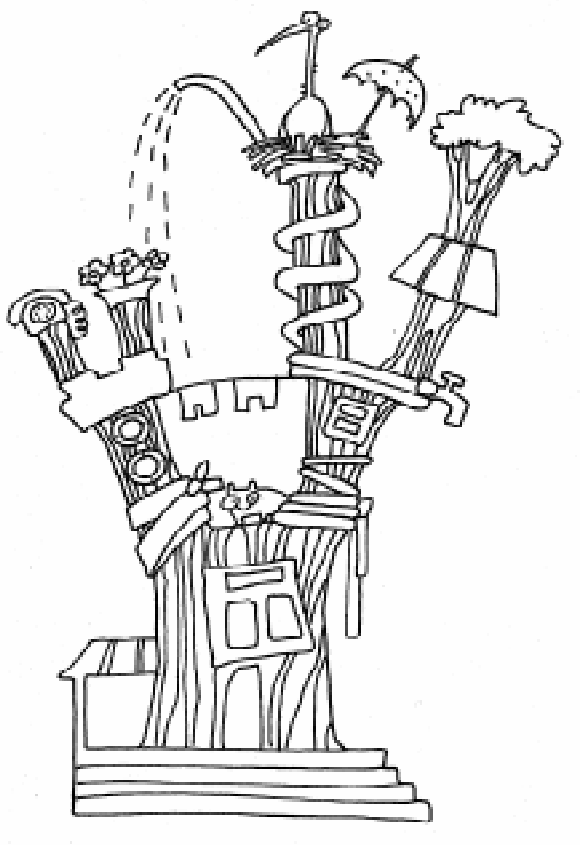
a & И & И | *b*

Вывод:

a | *b*

Мы рассмотрели бинарное дерево и алгоритмы работы с ним. Представленные нами алгоритмы анализировали, в основном, структуру дерева и, в меньшей мере, элементы, из которых дерево строилось.

Дерево поиска использовали для работы с данными, на которых задан порядок. Многие алгоритмы, описанные для бинарных деревьев, применимы и в случае деревьев поиска. Некоторые алгоритмы для дерева поиска были изменены. Поиск элемента в дереве поиска можно делать эффективнее, чем просто в бинарном дереве. Были добавлены новые алгоритмы, присущие только деревьям поиска: добавление элемента в дерево поиска, построение дерева поиска, поиск с включением и др.



Мы рассмотрели бинарное дерево и алгоритмы работы с ним. Представленные нами алгоритмы анализировали, в основном, структуру дерева и, в меньшей мере, элементы, из которых дерево строилось...

Дерево операций использовалось для представления формул. Многие алгоритмы, описанные для бинарных деревьев, применимы и для деревьев операций, причем, для деревьев операций алгоритмы имеют свою интерпретацию. Например, поиск числа листьев в бинарном дереве можно трактовать как поиск операндов в дереве операций. Соответствующие обходы бинарных деревьев позволяют получать формулы в бесскобочной записи. Специфичными для деревьев операций являются алгоритмы: построение по формуле дерева операций, вывод формулы в инфиксной записи с необходимыми скобками, вычисление значения формулы, дифференцирование формулы, выполнение частичных преобразований и др.

Следуя технологии объектно-ориентированного программирования, можно создать объект «бинарное дерево», который является родителем двух других объектов: «дерево поиска» и «дерево операций». Потомки наследуют методы родителя. При необходимости методы родителя могут быть переопределены. Каждый потомок может иметь свои собственные методы. Объектно-ориентированное программирование – отдельная интересная тема, а мы заметим лишь, что и отношения между объектами отображаются с помощью дерева, например, изображенного на рисунке 27.

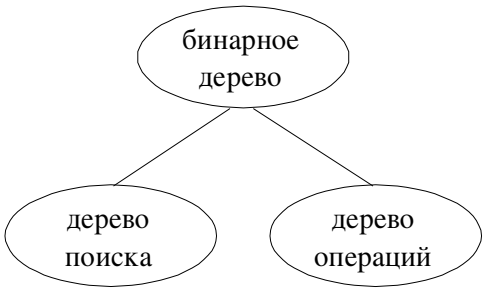


Рисунок 26

Литература.

1. Вирт Н. Алгоритмы+структуры данных=программы. М.: Мир, 1985, 392 с.
2. Дмитриева М.В., Кубенский А.А. Элементы современного программирования. СПб., 1991.
3. Епанешников А., Епанешников В. Программирование в среде Turbo Pascal 7.0. М., 1993.
4. Дмитриева М.В., Кубенский А.А. Турбо Паскаль и Турбо Си: построение и обработка структур данных. СПб., 1996.
5. Касьянов В.Н., Сабельфельд В.К. Сборник задач по практикуму на ЭВМ. М., 1986.
6. Черкасова П.Г. Компьютер и графы. Компьютерные инструменты в образовании, № 6, 1999 г.
7. Дмитриева М.В., Поздняков С.Н. Формулы, формулы, формулы... Компьютерные инструменты в образовании, № 2, 2000 г.

НАШИ АВТОРЫ

Дмитриева Марина Валерьевна, доцент кафедры информатики Санкт-Петербургского государственного университета.