Лабораторная работа № 6

ДЕРЕВЬЯ В ПРИКЛАДНЫХ АЛГОРИТМАХ

Цель работы: изучение и анализ структур данных, применяемых для описания деревьев в прикладных программах; анализ и программная реализация операторов для работы с бинарными деревьями; разработка и исследование прикладных программ, в которых применяются бинарные деревья.

1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1. Машинное представление деревьев

Для машинного представления деревьев обычно используются массивы или связанные списки, которые могут быть реализованы в динамической памяти.

Для работы с деревьями применяются следующие операторы:

1) `PARENT(i,T)` – функция для определения предшественника (предка) вершины `i` в дереве `T`. Для корня дерева эта функция возвращает нулевое значение или пустой указатель в зависимости от реализации;

2) `LEFT(i,T)` - функция для определения левого последователя (потомка) вершины `i` в дереве `T`. Возвращает номер вершины (указатель) или нулевое значение (пустой указатель) при отсутствии левого потомка;

3) `RIGHT(i,T)` - функция для определения правого последователя (потомка) вершины `i` в дереве `T`. Возвращает номер вершины (указатель) или нулевое значение (пустой указатель) при отсутствии правого потомка;

4) `INFO(i,T)` – функция, которая возвращает информацию (метку), приписанную вершине `i` дерева `T`;

5) `ROOT(T)` – функция для определения корня дерева `T`.

Представление структуры дерева с помощью **массива предков** является самым простым способом, требующим наименьшего объема памяти.

Если дерево имеет `n` вершин, то требуется одномерный массив `TREE[1..n]`. Элементы этого массива кодируются следующим образом:

`TREE[i]=j`, если вершина `j` является предком вершины `i`;

`TREE[i]=0`, если вершина `i` является корнем.

Пример описания структуры дерева массивом предков показан на рис. 1. Такое представление поддерживает оператор определения предка

```
j:=PARENT(i,TREE),
```

поскольку `TREE[i]=j`.

Индекс	1	2	3	4	5	6	7	8	9	10
TREE	0	1	1	2	2	5	5	5	3	3

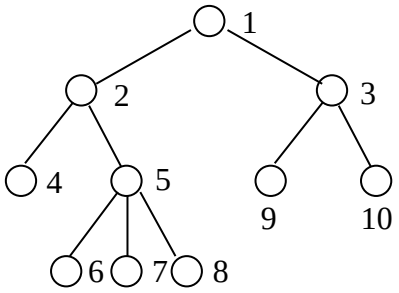


Рис. 1

При реализации оператора `INFO(i,T)` требуется еще один массив `DATA[1..n]`, элемент которого `DATA[i]` будет хранить информацию, соответствующую вершине `i`. Заметим, что на рис. 1 такой массив не показан.

Для выполнения операторов `ROOT`, `LEFT`, `RIGHT` требуется просмотреть массив `TREE` один раз. Предполагается, что потомки каждой вершины нумеруются строго слева направо по возрастанию индексов.

Выполнение оператора вставки вершин в качестве новых листьев выполняется достаточно просто, если в массиве есть зарезервированные заранее элементы. Удаление вершин весьма затруднительно, поскольку связано с изменением структуры связей, которая в свою очередь, описывается не только значениями элементов `j=TREE[i]`, но и их индексами `i`.

Узловое представление бинарных деревьев также использует несколько одномерных массивов, определяющих для каждой вершины данные, левого и правого потомков, а также предка. Кроме того, для удобства работы может быть введено непосредственное определение его корня.

```
const
  maxnodes=100 {макс. число вершин дерева}
type
  TREE=record
    data: array[1..maxnodes] of my_type;
    left: array[1..maxnodes] of integer;
    right: array[1..maxnodes] of integer;
    parent: array[1..maxnodes] of integer;
    root: integer; {индекс корня}
    node: integer {число вершин дерева}
  end;
```

Значениями элементов массивов `left`, `right` и `parent` могут быть индексы соответствующих вершин дерева или нули. Пример описания структуры бинарного дерева с произвольной нумерацией вершин приведен на рис. 2.

Индекс	1	2	3	4	5	6	7	8	9
parent	0	1	2	1	4	5	4	2	5
left	4	8	0	7	6	0	0	0	0
right	2	3	0	5	9	0	0	0	0

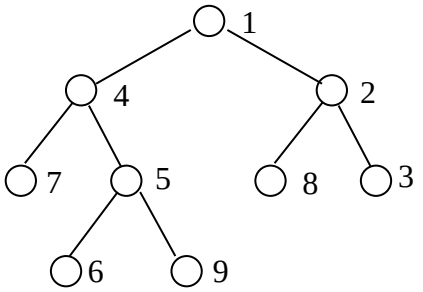


Рис. 2

Такое представление поддерживает все операторы для работы с деревьями, которые не изменяют своей структуры в процессе решения задачи, а также позволяет легко добавлять новые вершины.

Следует заметить, что узловое представление дерева можно упростить в зависимости от используемого алгоритма. Если реализуется просмотр дерево от корня к листьям, то можно исключить массив `parent`, если от листьев к корню, то можно исключить массивы `left` и `right`.

Переменная `root` также может не использоваться. В этом случае корень можно найти за один просмотр массива `parent`, где требуется найти элемент с нулевым значением.

Представление деревьев в динамической памяти предполагает использование указателей и создание связанных структур данных, подобных однонаправленным или двунаправленным спискам.

Для каждой вершины дерева, кроме поля данных, должны быть заданы указатели на другие вершины в зависимости от решаемой задачи:

вариант 1 - указатель на предка;

вариант 2 - указатели на потомков;

вариант 3 - указатели на предка и потомков.

Первый вариант позволяет легко описывать деревья любой конфигурации (рис. 3), однако ориентирован только на их просмотр от листьев к корню, что встречается достаточно редко.

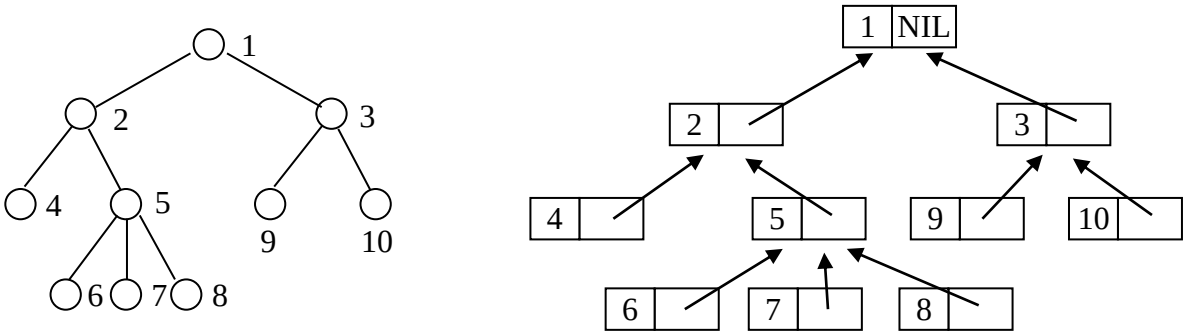


Рис. 3

Второй вариант является наиболее распространенным и применяется только для бинарных деревьев, где число потомков не более двух (рис. 4).

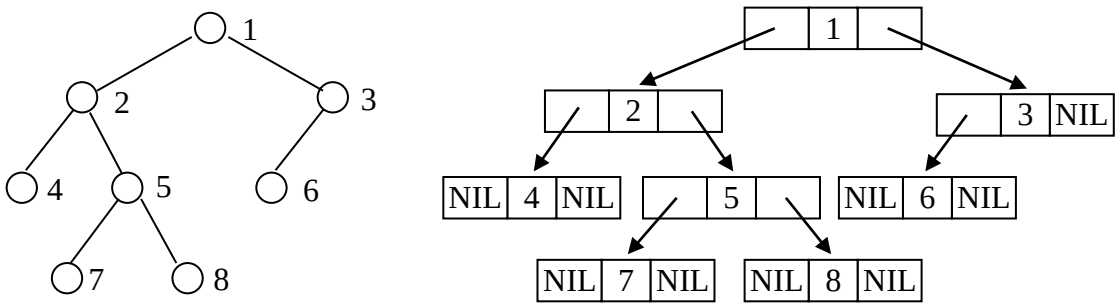


Рис. 4

Третий вариант также используется только для бинарных деревьев, но более редко, поскольку ориентирован на просмотр деревьев как от листьев к корню, так и от корня к листьям.

Рассмотрим реализацию второго варианта более подробно. Определим тип данных «дерево», как указатель на ячейку, задающую одну вершину дерева:

```
type
  TREE = ^node;
  node = record
    data: {тип данных для вершины}
    left, right: TREE;
  end;
  position = TREE; {позиция вершины в дереве}
```

Конкретное дерево описывается указателем на корень дерева, например, `root1`, который размещается в статической памяти:

```
var
  root1: TREE;
```

Позиция вершины определяется как указатель на эту вершину. Для корня позиция определяется соответствующим указателем.

Реализация некоторых операторов для работы с деревом может иметь следующий вид.

```
procedure MAKENULL(var T:TREE);
begin
  T:=NIL;
end;

function LEFT(p:position; T:TREE):position;
begin
  if p=NIL then ERROR("Вершина p не существует")
  else LEFT:=p^.left;
end;
```

Реализация оператора PARENT(p,T) определения предка вершины p для второго варианта описания дерева затруднительна и связана с систематическим просмотром дерева T, начиная от корня, до выявления такой пары вершин p и q, для которой выполняется условие LEFT(q,T)=p или RIGHT(q,T)=p. Тогда q – это предок p.

1.2. Поиск по бинарному дереву

Использование дерева для организации записей позволяет сохранить эффективность бинарного поиска при достаточно простой реализации операций вставки и удаления записей.

Для выполнения поиска применяется двоичное (бинарное) дерево, в котором любая вершина (узел) может иметь не более двух последователей (потомков).

Построение дерева производится следующим образом. При этом предполагается, что никакие две записи не имеют одинаковых ключей.

1. Первую запись входной последовательности сопоставить с корнем дерева.

2. Сравнить ключ очередной записи с ключом корня. Если он меньше, сравнить с ключом левого потомка, если больше - с ключом правого потомка и т.д. Если потомок отсутствует, то образовать новую вершину дерева и сопоставить с ней соответствующую запись.

3. Повторять п. 2 до тех пор, пока не будет просмотрена вся входная последовательность записей.

Пример 1. Пусть входная последовательность записей имеет следующие значения ключей:

8, 11, 3, 9, 14, 1, 10, 5, 6, 12, 15, 7, 13.

Полученное дерево показано на рис. 5, где рядом с вершинами указаны их порядковые номера, определяемые исходной последовательностью ключей.

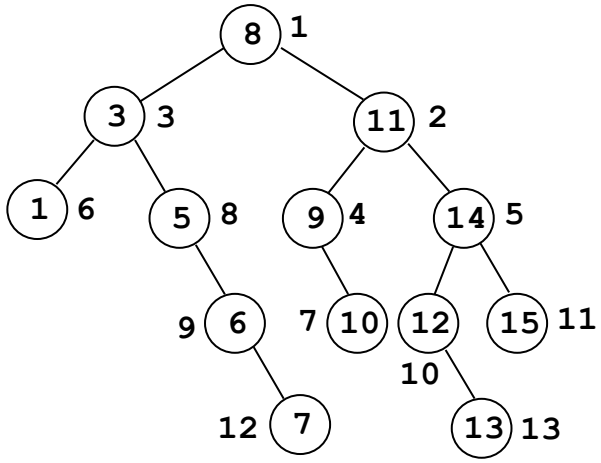


Рис. 5

В процессе построения дерева удобно использовать его узловое представление, в котором массив предков не используется:

```
const
  maxn = 100;  {макс. число вершин дерева}
type
  TREE = record
   a: array[1..maxn] of integer;  {массив ключей}
   L: array[1..maxn] of integer;  {левые потомки}
   R: array[1..maxn] of integer; {правые потомки}
   n: integer       {текущее число вершин дерева}
  end;
```

Учитывая, что вершины построенного дерева (рис. 5) пронумерованы в порядке следования ключей исходной последовательности, получим следующее узловое представление, где `n = 13`.

Узел	1	2	3	4	5	6	7	8	9	10	11	12	13
a	8	11	3	9	14	1	10	5	6	12	15	7	13
L	3	4	6	0	10	0	0	0	0	0	0	0	0
R	2	5	8	7	11	0	0	9	12	13	0	0	0

При поиске начиная от корня производится просмотр вершин дерева аналогично его построению. Отсутствие левого или правого потомка, к которому должен быть выполнен переход, означает неуспешное завершение поиска. Схема алгоритма поиска по бинарному дереву приведена на рис. 6.

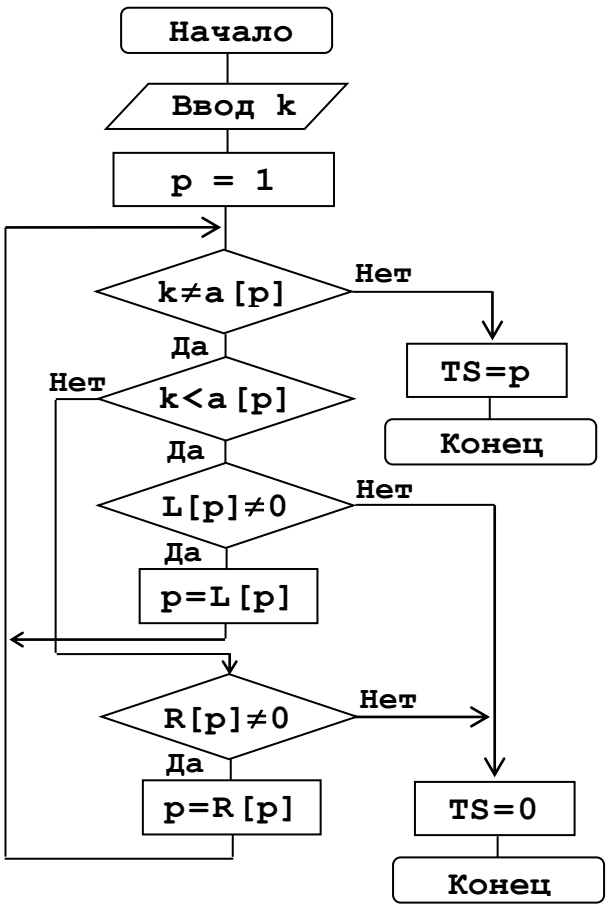


Рис. 6

Представленный алгоритм можно реализовать как функцию TS, в которой используются следующие данные:

k – аргумент поиска (целое число);

p – номер некоторой вершины дерева;

a[1..n] – целочисленный массив ключей;

L[p], R[p] – номера левого и правого потомков вершины p;

TS – результат поиска (индекс TS∈{1,2,...,n} искомого элемента, где a[BS]=k, или признак неуспешного завершения поиска TS=0).

Также предполагается, что корень всегда имеет первый номер, а отсутствие потомков кодируется нулями в массивах L и R.

Замечание. Этот же алгоритм можно использовать для построения дерева поиска. Добавление вершины в дерево (за исключением корня) производится при выполнении условия L[p]=0 или R[p]=0. В первом случае требуется определить L[p]=q, во втором R[p]=q, где q - номер очередной вершины. Далее выполняется присваивание:

a[q]=k; L[q]=0; R[q]=0.

1.3. Обходы бинарного дерева

Многие прикладные алгоритмы, использующие бинарные деревья, включают два этапа. На первом этапе строится бинарное дерево, а на втором выполняется систематический обход его вершин.

Наиболее часто используемыми способами обхода вершин являются:

1) обход в прямом порядке (просмотр “в глубину”);

2) обход в обратном порядке;

3) симметричный обход (обход во внутреннем порядке).

Указанные способы обхода вершин бинарного дерева описываются рекурсивными алгоритмами TLR, LRT и LTR соответственно, где T – корень дерева, L – левое поддерево и R – правое поддерево (рис. 5)

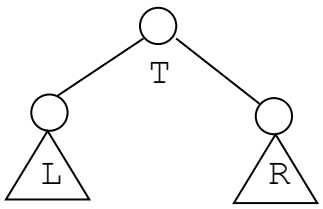


Рис. 5

Алгоритм TLR, реализующий *обход в прямом порядке*, включает следующие действия:

1) просмотреть корень T;

2) выполнить обход в прямом порядке левого поддерева L;

3) выполнить обход в прямом порядке левого поддерева R.

Обход в обратном порядке выполняется по алгоритму LRT:

1) выполнить обход в обратном порядке левого поддерева L;

2) выполнить обход в обратном порядке левого поддерева R;

3) просмотреть корень T.

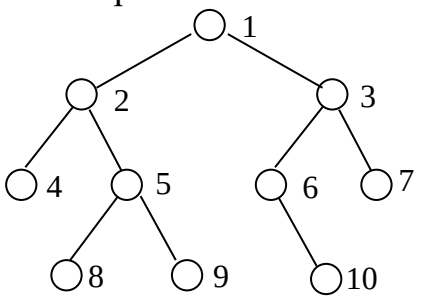
Симметричный обход по алгоритму LTR:

1) выполнить симметричный обход левого поддерева L;

2) просмотреть корень T;

3) выполнить симметричный обход левого поддерева R.

Пример 2. На рис. 6 приведено бинарное дерево и результаты работы трех алгоритмов обхода, представленные в виде последовательностей индексов вершин.



Обход в прямом порядке (TLR)
1,2,4,5,8,9,3,6,10,7

Обход в обратном порядке (LRT)
4,8,9,5,2,10,6,7,3,1

Симметричный обход (LTR)
4,2,8,5,9,1,6,10,3,7

Рис. 6

Рассмотрим реализацию алгоритма TLR для некоторого дерева T. Рекурсивная процедура обхода в прямом порядке вершин поддерева, корнем которого является вершина p, выводит метки (индексы) вершин:

```
procedure TLR(p: position);
var q: position;
begin
  if p <> NIL then
    begin
      write(INFO(p,T));
      q:= LEFT(p,T); TLR(q);
      q:= RIGHT(p,T); TLR(q);
    end;
end;
```

Аналогично строятся другие процедуры обхода дерева.

```
procedure LRT(p: position);
var q: position;
begin
  if p <> NIL then
    begin
      q:= LEFT(p,T); LRT(q);
      q:= RIGHT(p,T); LRT(q);
      write(INFO(p,T));
    end;
end;
```

```
procedure LTR(p: position);
var q: position;
begin
  if p <> NIL then
    begin
      q:= LEFT(p,T); LTR(q);
      write(INFO(p,T));
      q:= RIGHT(p,T); LTR(q);
    end;
end;
```

1.4. Сортировка по бинарному дереву

В качестве практического применения обхода дерева рассмотрим алгоритм сортировки. Он включает два этапа.

На первом этапе строится сортирующее дерево (подобно дереву для реализации поиска, но с одним небольшим отличием): ключ корня левого поддерева всегда должен быть меньше, чем ключ корня дерева, а ключ корня правого поддерева – *больше или равен*, чем ключ корня дерева.

На втором этапе выполняется симметричный обход вершин по алгоритму `LTR`. Получаемая в процессе обхода последовательность номеров вершин определяет отсортированную по возрастанию последовательность ключей.

Пример 3. Пусть имеется входная последовательность ключей:

Индекс	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Ключ	14	15	4	9	7	18	3	5	16	4	20	17	9	14	5

Ей соответствует бинарное дерево, показанное на рис. 7, где значения ключей указаны рядом с вершинами, а номера вершин однозначно соответствует индексам в исходной последовательности.

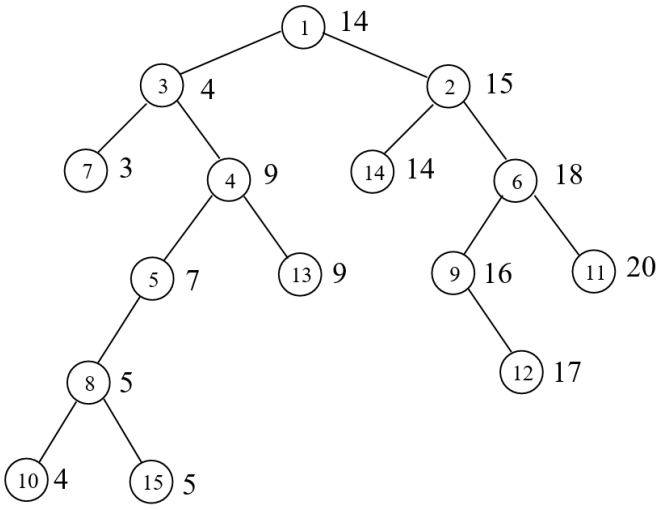


Рис. 7

Выполнение симметричного обхода этого дерева по алгоритму `LTR` дает следующий порядок просмотра вершин, которая определяет отсортированную последовательность ключей.

Узел	7	3	10	8	15	5	4	13	1	14	2	9	12	6	11
Ключ	3	4	4	5	5	7	9	9	14	14	15	16	17	18	20

Алгоритм сортировки по бинарному дереву имеет следующие особенности: сложность $O(n\log n)$; требуется $O(n)$ дополнительной памяти; удобен при получении данных путем чтения из потока.

В приложении приведены два варианта реализации рассмотренного алгоритма сортировки, реализованных с представлением дерева в динамической памяти. Во втором варианте применяется рекурсивная процедура `ADD` добавления вершин, которая использует рекурсивное определение бинарного дерева.

2. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

2.1. Реализовать на языке программирования набор операторов для работы с бинарными деревьями. Для описания дерева использовать узловое представление, вариант на основе указателей или другое машинное представление в соответствии с полученным заданием.

2.2. Выполнить первое задание (решение задачи поиска).

2.2.1. Разработать алгоритм построения бинарного дерева для решения задачи поиска в массиве неповторяющихся ключей.

2.2.2. Разработать алгоритм поиска по бинарному дереву в массиве неповторяющихся ключей (за основу принять алгоритм построения соответствующего бинарного дерева).

2.2.3. Написать программу для решения задачи поиска по бинарному дереву для входной последовательности неповторяющихся ключей. В программе предусмотреть возможность анализа структур данных, описывающих сформированное бинарное дерево (например, путем их вывода на экран дисплея или на печать).

2.2.4. Решить контрольный пример «вручную» (включая построение дерева) и сравнить его с машинным решением.

2.3. Разработать и реализовать на языке программирования алгоритмы обхода бинарного дерева (для заданного способа машинного представления дерева).

2.4. Исследовать разработанные программы на нескольких контрольных примерах.

2.4. Выполнить второе задание (решение задачи сортировки).

2.4.1. Разработать алгоритм построения бинарного дерева для решения задачи сортировки произвольного массива ключей (можно модифици-

ровать алгоритм построения бинарного дерева для решения задачи поиска из п. 2.2.1).

2.4.2. Написать программу для решения задачи сортировки путем симметричного обхода бинарного дерева для произвольного массива ключей. В программе предусмотреть возможность анализа структур данных, описывающих сформированное бинарное дерево (например, путем их вывода на экран дисплея или на печать).

2.4.3. Решить контрольный пример «вручную» (включая построение дерева) и сравнить его с машинным решением.

3. СОДЕРЖАНИЕ ОТЧЕТА

1. Цель работы.

2. Вариант задания и исходные данные.

3. Программная реализация операторов для работы с бинарным деревом (для заданного варианта машинного представления дерева).

4. Отчетные материалы по первому заданию:

- схема алгоритма построения бинарного дерева для задачи поиска;
- схема алгоритма поиска по бинарному дереву;
- исходные данные и результаты решения контрольного примера для задачи поиска.

5. Схемы алгоритмов обхода бинарного дерева (для заданного способа машинного представления дерева).

6. Отчетные материалы по второму заданию:

- схема алгоритма построения бинарного дерева для задачи сортировки;
- исходные данные и результаты решения контрольного примера для задачи сортировки.

7. Выводы.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как реализовать алгоритм поиска по бинарному дереву для произвольного массива ключей?

2. Будет ли работать корректно алгоритм поиска по бинарному дереву для произвольного массива ключей, если использовать дерево, которое строится для сортировки?

3. Можно ли определить первое (последнее) вхождение ключа в произвольном массиве при поиске по бинарному дереву?

4. Как реализовать поиск всех вхождений ключа в произвольном массиве при поиске по бинарному дереву?

Приложение

Сортировка по дереву (вариант 1)

```
program TREE_S;
uses crt;
type
  TREE = ^node;
    node = record
    data: integer;
    left, right: TREE
  end;
var
   root, last, next, pass: TREE;
   i,n: integer;
   m: array[1..100] of integer;
procedure LTR(r: TREE);
begin
 if r <> nil then
   begin
     LTR(r^.left);
     write(r^.data, ' ');
     LTR(r^.right);
   end;
end;

BEGIN
clrscr;
n:=10; for i:= 1 to n do read(m[i]);
for i:= 1 to n do write(m[i],' '); writeln;
{создание пустого дерева}
root:=nil;
{создание корня}
new(root);
root^.data:=m[1]; root^.left:=nil; root^.right:=nil;
for i:=2 to n do
  begin
    {создание следующей вершины}
    new(next);
    next^.data:=m[i]; next^.left:=nil; next^.right:=nil;
    {просмотр дерева}
    pass:=root;
    while pass<>nil do
      begin
        last:= pass;
        if next^.data < pass^.data
           then pass:=pass^.left
           else pass:=pass^.right;
      end;
    {добавление вершины к дереву}
    if next^.data < last^.data
       then last^.left:=next
       else last^.right:=next;
  end;
writeln;
LTR(root);
END.
```

Сортировка по дереву (вариант 2)

```
program TREE_S_R;
uses crt;
type
  TREE = ^node;
    node = record
    data: integer;
    left, right: TREE
  end;
var
  root, last, next, pass: TREE;
  i,n: integer;
  m: array[1..100] of integer;
procedure LTR(r: TREE);
begin
 if r <> nil then
   begin
     LTR(r^.left);
     write(r^.data, ' ');
     LTR(r^.right);
   end;
end;
procedure ADD(var r:TREE; p: TREE);
begin
  if r = nil then r:=p
  else
   if p^.data<r^.data then ADD(r^.left, p)
                      else ADD(r^.right, p);
end;

BEGIN
clrscr;
n:=10; for i:= 1 to n do read(m[i]);
for i:= 1 to n do write(m[i],' '); writeln;
{создание пустого дерева}
root:=nil;
for i:=1 to n do
  begin
    {создание следующей вершины}
    new(next);
    next^.data:=m[i]; next^.left:=nil; next^.right:=nil;
    {добавление вершины к дереву}
    ADD(root, next);
  end;

writeln;
LTR(root);
END.
```