3. ДЕРЕВЬЯ

Полезной нелинейной структурой данных является дерево (или лес).

3.1. Определения дерева, леса, бинарного дерева. Скобочное представление

Дадим формальное определение *дерева*, следуя [10].

Дерево – конечное множество T, состоящее из одного или более узлов, таких, что

а) имеется один специально обозначенный узел, называемый *корнем* данного дерева;

б) остальные узлы (исключая корень) содержатся в $m \geq 0$ попарно не пересекающихся множествах $T_1, T_2, ..., T_m$, каждое из которых, в свою очередь, является деревом. Деревья $T_1, T_2, ..., T_m$ называются *поддеревьями* данного дерева.

При программировании и разработке вычислительных алгоритмов удобно использовать именно такое *рекурсивное* определение, поскольку рекурсивность является естественной характеристикой этой структуры данных.

Каждый узел дерева является корнем некоторого поддерева. В том случае, когда множество поддеревьев такого корня пусто, этот узел называется *концевым узлом*, или *листом*. *Уровень* узла определяется рекурсивно следующим образом: 1) корень имеет уровень 1; 2) другие узлы имеют уровень, на единицу больший их уровня в содержащем их поддереве этого корня. Используя для уровня узла a дерева T обозначение *уровень* (a, T), можно записать это определение в виде

$$\textit{уровень}(a,T) = \begin{cases} 1, & \text{если } a - \text{корень дерева } T \\ \textit{уровень}(a,T_i)+1, & \text{если } a - \text{не корень дерева } T \end{cases}$$

где T_i – поддерево корня дерева T, такое, что $a \in T_i$.

Говорят, что каждый корень является *отцом* корней своих поддеревьев и что последние являются *сыновьями* своего отца и *братьями* между собой. Говорят также, что узел n – *предок* узла m (а узел m – *потомок* узла n), если n – либо отец m, либо отец некоторого предка m.

Если в определении дерева существен порядок перечисления поддеревьев T_1, T_2, ..., T_m, то дерево называют *упорядоченным* и говорят о «первом» (T_1), «втором» (T_2) и т. д. поддеревьях данного корня. Далее будем считать, что все рассматриваемые деревья являются упорядоченными, если явно не оговорено противное. Отметим также, что в терминологии теории графов определенное ранее упорядоченное дерево более полно называлось бы «конечным ориентированным (корневым) упорядоченным деревом».

Лес – это множество (обычно упорядоченное), состоящее из некоторого (быть может, равного нулю) числа непересекающихся деревьев. Используя понятие леса, пункт *б* в определении дерева можно было бы сформулировать так: *узлы дерева, за исключением корня, образуют лес*.

Традиционно дерево изображают графически, например так, как на рис. 3.1.

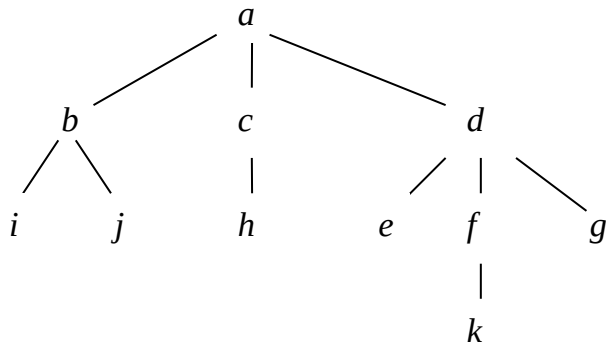


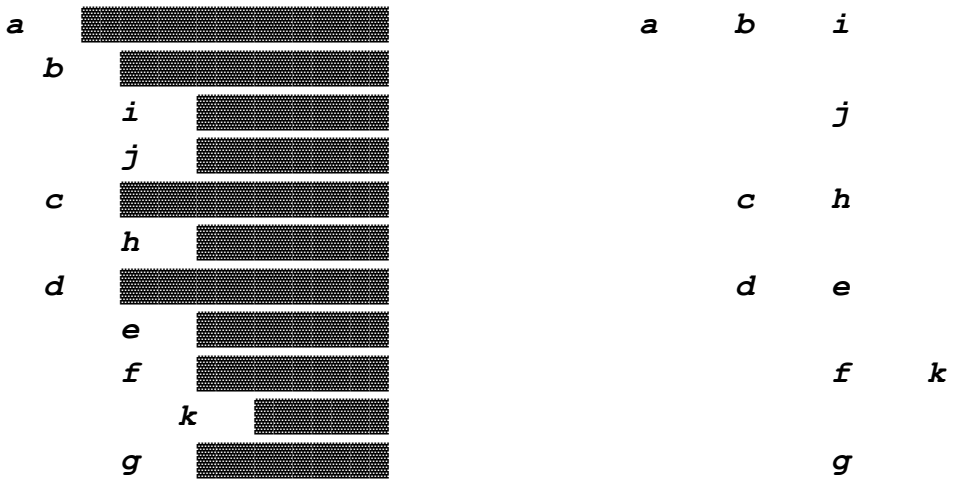
Рис. 3.1. Графическое изображение дерева

В графическом представлении для изображения структуры дерева используется двухмерность рисунка. При машинной обработке часто удобнее использовать текстовое представление дерева. Например, таким представлением может быть так называемый уступчатый список. Здесь «двухмерность» проявляется за счет того, что текст разбит на строки и фиксируется позиция символа узла в строке. На рис. 3.2, *а*, *б* представлено в виде уступчатого списка дерево, изображенное на рис. 3.1.

Другой вид текстового (и принципиально одномерного) представления дерева – это так называемая скобочная запись (ср. с записью иерархических списков в 1.7). Определим скобочное представление дерева и леса:

< лес > ::= пусто | < дерево > < лес >,

< дерево > ::= (< корень > < лес >).



а *б*

Рис. 3.2. Представление дерева: *а* – в виде уступчатого списка;
б – в виде «упрощенного» уступчатого списка

В скобочном представлении дерево, изображенное на рис. 3.1, запишется как

$$(a\ (b\ (i)\ (j))\ (c\ (h))\ (d\ (e)\ (f\ (k))\ (g))).$$

Наиболее важным типом деревьев являются *бинарные деревья*. Удобно дать следующее формальное определение. *Бинарное дерево* – конечное множество узлов, которое либо пусто, либо состоит из корня и двух непересекающихся бинарных деревьев, называемых правым поддеревом и левым поддеревом. Так определенное бинарное дерево *не* является частным случаем дерева. Например, бинарные деревья, изображенные на рис. 3.3, различны между собой, так как в одном случае корень имеет пустое правое поддерево, а в другом случае правое поддерево непусто. Если же их рассматривать как деревья, то они идентичны.

Рис. 3.3. Бинарные деревья из двух узлов

Определим скобочное представление бинарного дерева (БД):

< БД > ::= < пусто > | < непустое БД >,

< пусто > ::= Λ,

< непустое БД > ::= (< корень > < БД > < БД >).

Здесь пустое дерево имеет специальное обозначение – Λ.

Например, бинарное дерево, изображенное на рис. 3.4, имеет скобочное представление

(*a* (*b* (*d* Λ (*h* Λ Λ)) (*e* Λ Λ)) (*c* (*f* (*i* Λ Λ) (*j* Λ Λ)) (*g* Λ (*k* (*l* Λ Λ) Λ)))).

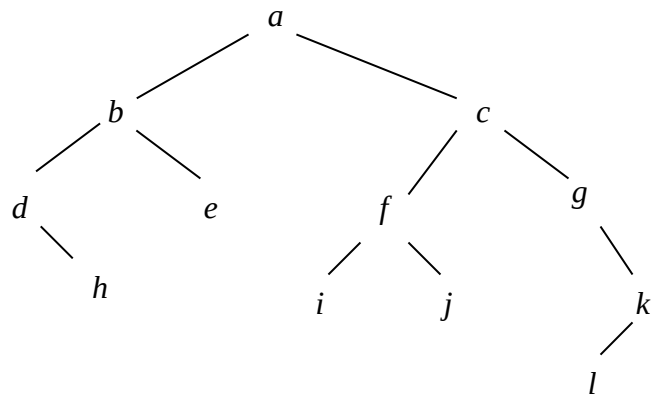


Рис. 3.4. Бинарное дерево

Можно упростить скобочную запись бинарного дерева, исключив «лишние» знаки Λ по правилам:

1) (< корень > < непустое БД > Λ) ≡ (< корень > < непустое БД >),

2) (< корень > Λ Λ) ≡ (< корень >).

Тогда, например, скобочная запись бинарного дерева, изображенного на рис. 3.4, будет иметь вид

(*a* (*b* (*d* Λ (*h*) (*e*)) (*c* (*f* (*i*) (*j*)) (*g* Λ (*k* (*l*))))).

3.2. Спецификация дерева, леса, бинарного дерева

Рассмотрим функциональную спецификацию структуры данных дерева с узлами типа α: *Tree of* α = *Tree* (α). При этом лес деревьев *Forest* (α) определим как *L_list* (*Tree* (α)) через уже известную структуру линейного списка *L_list* с базовыми функциями *Cons*, *Head*, *Tail*, *Null* (см. 1.6). Базовые операции с деревом задаются набором функций:

1) *Root*: *Tree* → α;

2) *Listing*: *Tree* → *Forest*;

3) *ConsTree*: α ⊗ *Forest* → *Tree*

и аксиомами (∀*u:* α; ∀*f: Forest* (α); ∀*t*: *Tree* (α)):

A1) *Root* (*ConsTree* (*u*, *f*)) = *u*;

A2) *Listing* (*ConsTree* (*u*, *f*)) = *f*;

A3) *ConsTree* (*Root* (*t*), *Listing* (*t*)) = *t*.

Здесь функции *Root* и *Listing* – селекторы: *Root* выделяет корень дерева, а *Listing* выделяет лес поддеревьев корня данного дерева. Конструктор *ConsTree* порождает дерево из заданных узла и леса деревьев.

Тот факт, что структура данных *Forest* явно определена через *L_list* (*Tree*), позволяет реализовать структуру дерева (леса) на базе другой структуры данных, а именно на базе иерархических списков. Достаточно рассматривать при этом описанное в 3.1 скобочное представление дерева как *S*-выражение специальной структуры. Возможно и другое удобное представление дерева (леса), основанное на некотором соответствии леса и бинарного дерева, описанном далее в 3.3.

Рассмотрим функциональную спецификацию структуры данных бинарного дерева с узлами типа α: *BinaryTree* (α) ≡ *BT* (α). Здесь важно различать ситуации обработки пустого и непустого бинарных деревьев, поскольку некоторые операции определяются только на непустых бинарных деревьях. Далее считаем, что значение типа *BT* есть либо Λ (пустое бинарное дерево), либо значение типа *NonNullBT*. Тогда базовые операции типа *BT* (α) задаются набором функций:

1) *Root*: *NonNullBT* → α;

2) *Left*: *NonNullBT* → *BT*;

3) *Right*: *NonNullBT* → *BT*;

4) *ConsBT*: α ⊗ *BT* ⊗ *BT* → *NonNullBT*;

5) *Null*: *BT* → *Boolean*;

6) Λ: → *BT*

и набором аксиом (∀*u*: α, ∀*b*: *NonNullBT* (α), ∀*b*1, *b*2: *BT* (α)):

A1) *Null* (Λ) = *true*;

A1') *Null* (*b*) = *false*;

A2) *Null* (*ConsBT* (*u*, *b*1, *b*2)) = *false*;

A3) *Root* (*ConsBT* (*u*, *b*1, *b*2)) = *u*;

A4) *Left* (*ConsBT* (*u*, *b*1, *b*2)) = *b*1;

A5) *Right* (*ConsBT* (*u*, *b*1, *b*2)) = *b*2;

A6) *ConsBT* (*Root* (*b*), *Left* (*b*), *Right* (*b*)) = *b*.

Здесь функции *Root*, *Left* и *Right* – селекторы: *Root* выделяет корень бинарного дерева, а *Left* и *Right* – его левое и правое поддеревья соответственно. Конструктор *ConsBT* порождает бинарное дерево из

заданных узла и двух бинарных деревьев. Предикат *Null* – индикатор, различающий пустое и непустое бинарные деревья.

3.3. Естественное соответствие бинарного дерева и леса

Бинарные деревья особенно полезны, в том числе потому, что существует естественное взаимно-однозначное соответствие между лесами и бинарными деревьями, и многие операции над лесом (деревом) могут быть реализованы как соответствующие операции над бинарным деревом, представляющим этот лес (дерево).

Опишем такое представление леса бинарным деревом (и далее будем называть его *естественным*). Пусть лес F типа *Forest* задан как список деревьев T_i типа *Tree* (для $\forall i \in 1..m$):

$$F = (T_1\ T_2\ ...\ T_m).$$

Тогда *Head* (*F*) – первое дерево T_1 леса *F*, а *Tail* (*F*) – лес остальных деревьев (T_2 ... T_m). Если далее в дереве *Head* (*F*) выделить корень *Root* (*Head* (*F*)) и лес поддеревьев *Listing* (*Head* (*F*)),то исходный лес *F* представляется как совокупность трех частей:

1) корня первого дерева – *Root* (*Head* (*F*)),

2) леса поддеревьев первого дерева – *Listing* (*Head* (*F*)),

3) леса остальных деревьев – *Tail* (*F*).

Из этих трех частей рекурсивно порождается бинарное дерево *B* (*F*), представляющее лес *F*:

B (*F*) ≡ **if** *Null* (*F*) **then** Λ
else *ConsBT* (*Root* (*Head* (*F*)),
B (*Listing* (*Head* (*F*))),
B (*Tail* (*F*))).

Согласно этому определению корнем бинарного дерева *B* (*F*) является корень первого дерева T_1 в лесу *F*, левым поддеревом бинарного дерева *B* (*F*) является бинарное дерево, представляющее лес поддеревьев первого дерева T_1, а правым поддеревом бинарного дерева *B* (*F*) является бинарное дерево, представляющее лес остальных (кроме первого) деревьев исходного леса *F*.

Это формальное определение имеет следующую наглядную

интерпретацию. Рассмотрим, для примера, лес из двух деревьев, представленный на рис. 3.5.

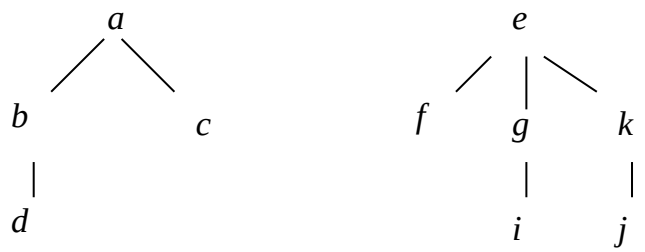


Рис. 3.5. Лес из двух деревьев

Представляющее этот лес бинарное дерево можно получить, если соединить последовательно сыновей каждой семьи и убрать все вертикальные связи, кроме связей, идущих от отцов к их первым сыновьям, как это показано на рис. 3.6.

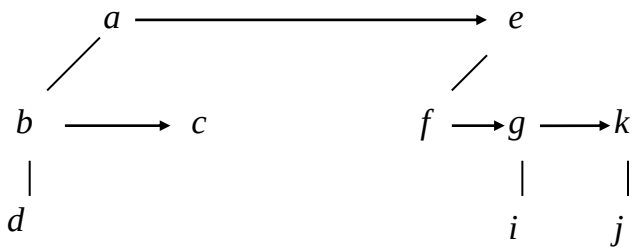


Рис. 3.6. Преобразованный лес

Повернув затем изображение леса (рис. 3.6) на 45°, получаем бинарное дерево, изображенное на рис. 3.7.

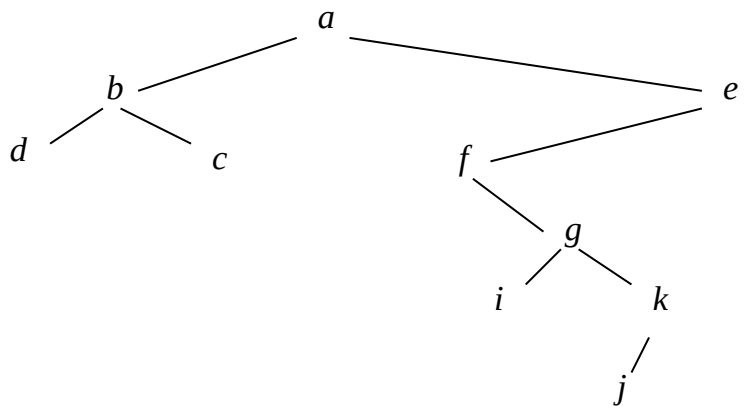


Рис. 3.7. Бинарное дерево, соответствующее лесу на рис. 3.5

Легко видеть, что, выполняя эти действия в обратном порядке, получим единственный лес, соответствующий данному бинарному дереву.

Это обратное преобразование формально описывается следующим образом (здесь *F* – лес типа *Forest*, а *B* – бинарное дерево типа *BT*):

F (*B*) ≡ **if** *Null* (*B*) **then** *Nil*
else *Cons* (*ConsTree* (*Root* (*B*), *F* (*Left* (*B*))),
F (*Right* (*B*))).

Согласно этому определению первое дерево T_1 леса *F* образуется из корня бинарного дерева *B* и леса поддеревьев, полученного из левого поддерева бинарного дерева *B*. Остальные деревья (T_2 ... T_m) леса $F = (T_1\ T_2\ ...\ T_m)$ составляют лес, полученный из правого поддерева бинарного дерева *B*.

3.4. Обходы бинарных деревьев и леса

Многие алгоритмы работы с бинарными деревьями основаны на последовательной (в определенном порядке) обработке узлов дерева. В этом случае говорят об обходе (прохождении) бинарного дерева. Такой обход порождает определенный порядок перечисления узлов бинарного дерева. Выделяют несколько стандартных вариантов обхода. Будем именовать их в зависимости от того порядка, в котором при этом посещаются корень дерева и узлы левого и правого поддеревьев. Например, при КЛП-обходе сначала посещается корень, затем обходятся в КЛП-порядке последовательно левое и правое поддеревья. Приведем рекурсивные процедуры КЛП-, ЛКП- и ЛПК-обходов, прямо соответствующие их рекурсивным определениям (операция обработки узла обозначена как «посетить (узел)»):

```
procedure обходКЛП (b: BTree);
{прямой}
begin
  if not Null (b) then
  begin
    посетить (Root (b));
    обходКЛП (Left (b));
    обходКЛП (Right (b));
  end
end{обходКЛП};
```

```
procedure обходЛКП (b: BTree);
{обратный}
 begin
   if not Null (b) then
   begin
     обходЛКП (Left (b));
     посетить (Root (b));
     обходЛКП (Right (b));
   end
 end{обходЛКП};
```

```
procedure обходЛПК (b: BTree);
{концевой}
begin
  if not Null (b) then
  begin
     обходЛПК (Left (b));
     обходЛПК (Right (b));
     посетить ( Root (b));
  end
end{обходЛПК}
```

Следует обратить внимание на то, что в литературе используется различная терминология при именовании видов обхода бинарного дерева. Перечислим наиболее популярные варианты терминологии (виды обходов перечислены во всех вариантах в одной и той же последовательности):

1) КЛП, ЛКП, ЛПК [14];

2) прямой, обратный, концевой [10];

3) прямой, симметричный, обратный [13];

4) сверху вниз, слева направо, снизу вверх [5], [6];

5) префиксный (*PreOrder*), инфиксный (*InOrder*), постфиксный (*PreOrder*) [5], [6].

Иногда обход КЛП называют обходом в глубину. В некоторых случаях используется смешанная терминология [16]. Будем придерживаться далее вариантов терминологии 1 и 2. В варианте 2 названия обходам даны соответственно тому, что в КЛП-порядке корень посещается перед посещением узлов левого поддерева (прямой порядок), а в ЛКП-порядке корень посещается после обхода узлов левого поддерева (обратный порядок). В ЛПК-порядке корень посещается после обхода узлов левого и правого поддеревьев (концевой порядок). Такая терминология основана на важной роли левого поддерева в естественном соответствии бинарного дерева и леса (см. 3.3).

Терминология варианта 5 явно связана с обходом бинарного дерева, представляющего арифметическое выражение с бинарными операциями. Пусть, например, дано арифметическое выражение

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

На рис. 3.8 представлено соответствующее ему бинарное дерево.

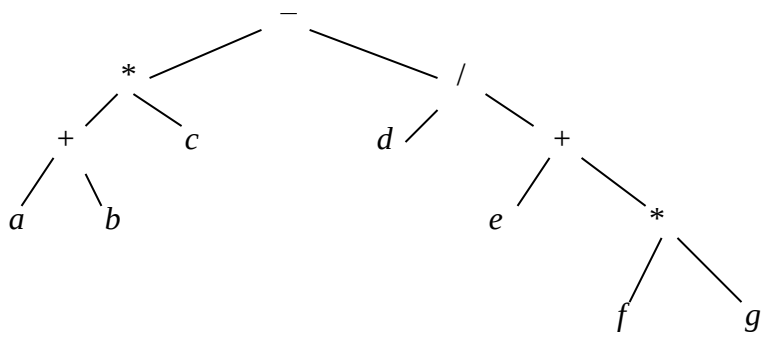


Рис. 3.8. Бинарное дерево, представляющее
арифметическое выражение $(a + b) * c – d / (e + f * g)$

Тогда три варианта обхода этого дерева порождают три известные формы записи арифметического выражения:

1) КЛП – префиксную запись

$$– * + a\ b\ c / d + e * f\ g\ ;$$

2) ЛКП – инфиксную запись (без скобок, необходимых для задания последовательности выполнения операций)

$$a + b * c – d / e + f * g\ ;$$

3) ЛПК – постфиксную запись

$$a\ b + c * d\ e\ f\ g * + / – .$$

В качестве упражнения полезно дать интерпретацию и остальным вариантам названий обходов.

Нерекурсивные процедуры обхода бинарных деревьев

Учитывая важность эффективной реализации обходов бинарных деревьев, рассмотрим *нерекурсивные* процедуры обходов. Общий нерекурсивный алгоритм для всех трех порядков обхода использует стек S для хранения упорядоченных пар (p, n), где p – узел бинарного дерева, а n – номер операции, которую надо применить к p, когда пара (p, n) будет выбрана из стека. Операции «посетить корень», «пройти левое поддерево», «пройти правое поддерево» нумеруются числами 1, 2, 3 в зависимости от порядка их выполнения в данном варианте обхода. В алгоритме эти операции будут реализованы следующим образом:

посетить корень – посетить (*p*);
пройти левое поддерево – **if not** *Null* (*Left* (*p*)) **then** *S* ← (*Left* (*p*), 1);
пройти правое поддерево – **if not** *Null* (*Right* (*p*)) **then** *S* ← (*Right* (*p*), 1).

Здесь и далее для краткости и наглядности использованы следующие обозначения операций со стеком: *S* ← *e* вместо *S* := *Push* (*e*, *S*) и *e* ← *S* вместо *Pop*2 (*e*, *S*).

Тогда общий алгоритм имеет вид:

```
procedure обход (b: BTree);
   var S: Stack of (BTree, operation);
        p: BTree;       op: operation {=1..3};
begin
   S := Create;  S ← (b, 1);
   while not Null (S) do
   begin (p, op) ← S;
      if op = 1 then begin S ← (p, 2); операция 1 end
      else if op = 2 then begin S ← (p, 3); операция 2 end
      else {op = 3} операция 3
   end{while}
end{обход}
```

В случае КЛП-обхода можно существенно упростить алгоритм, исключая лишние для этого варианта манипуляции со стеком. Здесь нет необходимости хранить в стеке номер операции. Итак, конкретизация (с упрощением) общего алгоритма для КЛП-обхода имеет вид:

```
procedure обход_КЛП (b: BTree); {прямой}
   var S: Stack of BTree;  p: BTree;
begin
    S:= Create; S ← b;
    while not Null (S) do
    begin
       p ← S; посетить (p);
       if not Null (Right (p)) then S ← Right (p);
       if not Null (Left (p)) then S ← Left (p)
    end{while}
end{обход_КЛП}
```

В случае ЛКП-обхода также возможно некоторое упрощение – в стеке сохраняются указания лишь на операции 1 или 2:

```
procedure обход_ЛКП (b: BTree); {обратный}
  var S: Stack of (BTree, operation);
      p: BTree; op: operation {=1..2};
begin  S:= Create; S ← (b , 1);
  while not Null (S) do
  begin (p, op) ← S;
    if op = 1 then
    begin S ← (p, 2);  if  not Null (Left (p)) then S ← (Left (p), 1)
    end
    else {op=2}
    begin
      посетить (p);  if not Null (Right (p)) then S ← (Right (p), 1)
    end
end{while}
end{обход_ЛКП}
```

Конкретизация общего алгоритма для ЛПК-обхода (здесь нет упрощений) имеет вид:

```
procedure обход_ЛПК (b: BTree); {концевой}
  var S: Stack of (BTree, operation);
      p: BTree;       op: operation {=1..3};
begin  S:= Create; S ← (b, 1);
  while not Null (S) do
  begin
    (p, op) ← S;
    if op = 1 then
    begin S ← (p, 2);  if not Null (Left (p)) then S ← (Left (p), 1)
    end
    else if op = 2 then
    begin
      S ← (p, 3); if not Null (Right (p)) then S ← (Right (p), 1)
    end else {op=3 } посетить (p)
  end{while}
end{обход_ЛПК}
```

Еще один полезный способ обхода бинарного дерева – обход в *горизонтальном* порядке (в ширину). При таком способе узлы бинарного дерева проходятся слева направо, уровень за уровнем от корня вниз (поколение за поколением от старших к младшим). Легко указать нерекурсивную процедуру горизонтального обхода, аналогичную процедуре КЛП-обхода (в глубину), но использующую *очередь* вместо стека:

```
procedure обход_горизонтальный (b: BTree);
   var Q: queue of BTree;
       p: BTree;
begin
    Q := Create;
    Q ← b;
    while not Null (Q) do
    begin
      p ← Q;
      посетить (p);
      if not Null (Left (p)) then Q ← Left (p);
      if not Null (Right (p)) then Q ← Right (p)
    end{while}
end{обход_горизонтальный}
```

Обходы леса

Следуя естественному соответствию между бинарными деревьями и лесами, можно на основе КЛП-, ЛКП- и ЛПК-обходов бинарного дерева получить три соответствующих порядка прохождения леса (и, следовательно, произвольного дерева).

Прямой порядок:

а) посетить корень первого дерева;

б) пройти поддеревья первого дерева (в прямом порядке);

в) пройти оставшиеся деревья (в прямом порядке).

Обратный порядок:

а) пройти поддеревья первого дерева (в обратном порядке);

б) посетить корень первого дерева;

в) пройти оставшиеся деревья (в обратном порядке).

Концевой порядок:

а) пройти поддеревья первого дерева (в концевом порядке);

б) пройти оставшиеся деревья (в концевом порядке);

в) посетить корень первого дерева.

Более формально обход леса в прямом порядке можно записать следующим образом:

```
procedure PreOrder (F: Forest); {прямой}
begin
  if not Null (F) then
  begin
    посетить (Root (Head (F));
    PreOrder (Listing (Head (F)));
    PreOrder (Tail (F));
  end
end{PreOrder}
```

Аналогично записываются процедуры обхода леса в обратном и концевом порядках.

Если необходимо применить какой-либо из обходов к лесу (дереву), можно сначала построить бинарное дерево, представляющее этот лес, а затем применить соответствующий обход бинарного дерева.

3.5. Представления и реализации бинарных деревьев

Рассмотрим варианты представления и реализации структуры данных бинарного дерева. Пусть базовый тип узлов есть *Elem*.

Ссылочная реализация бинарного дерева в связанной памяти основана на представлении типа *BT* (*Elem*) рекурсивными типами *BinT* и *Node*:

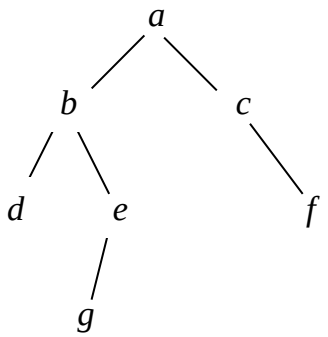
```
type
  BinT = ^Node;              {представление бинарного дерева}
  Node = record              {узел: }
           Info: Elem;       {– содержимое}
           LSub: BinT;       {– левое поддерево}
           RSub: BinT        {– правое поддерево}
         end {Node}
```

Здесь каждый узел дерева рассматривается как корень соответствующего поддерева, и этому поддереву сопоставляется запись из трех полей: поле *Info* хранит значение корня (типа *Elem*), а поля *LSub* и *RSub* – указатели на левое и правое поддеревья. Пустому дереву сопоставляется константа *NilBT* (на абстрактном уровне обозначаемая ранее как Λ). На рис. 3.9, *а*, *б* изображены бинарное дерево и его представление в ссылочной реализации.

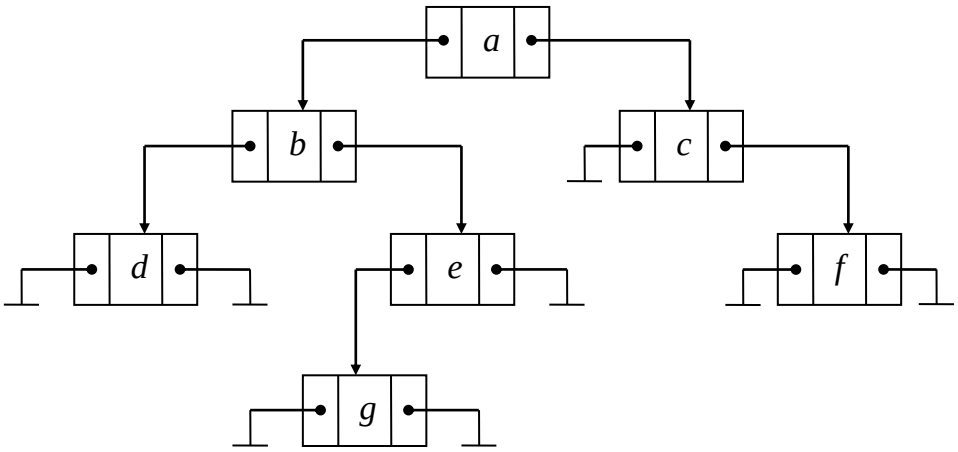
Интерфейсная часть модуля для работы с бинарным деревом на основе ссылочной реализации представлена на рис. 3.10.

Здесь в сравнении с функциональной спецификацией из 3.2 добавлены функция *CreateBT* и процедура *DestroyBT*, используемые для начала и завершения работы с экземпляром динамической структуры. Функция *CreateBT* формально специфицируется соотношениями

$$CreateBT: \rightarrow BT;\ \ Null\ (CreateBT) = true.$$



а



б

Рис. 3.9. Пример бинарного дерева (*а*) и его ссылочного представления (*б*)

Кроме того, добавлена процедура *Otkaz*, вызываемая при попытке некорректного применения основных функций.

Тип узлов дерева *Elem* должен быть задан в модуле *GlobalBT*, например, таким образом:

```
Unit GlobalBT;
Interface type Elem = Char;
Implementation
begin
end.
```

```
{Модуль для работы с бинарными деревьями}
Unit BinTree;
 Interface
   uses GlobalBT;
   const
       NilBT = nil;
   type
       BinT = ^Node;                {представление бинарного дерева}
                                    {тип Elem описан в GlobalBT}
       Node = record                {узел:
                   Info: Elem;      {– содержимое}
                   LSub: BinT;      {– левое поддерево}
                   RSub: BinT       {– правое поддерево}
                end {Node};
function CreateBT: BinT;
function NullBT (t: BinT): Boolean;
function RootBT (t: BinT): Elem;
function LeftBT (t: BinT): BinT;
function RightBT (t: BinT): BinT;
function ConsBT (e: Elem; LS, RS: BinT): BinT;
procedure DestroyBT (var b: BinT);
procedure Otkaz (n: Byte);
{-------------------------------------------------------------------------------}
Implementation
...
end.
```

Рис. 3.10. Модуль для работы с бинарными деревьями

Часть **Implementation** модуля *BinTree* может иметь следующий вид:

```
Implementation

procedure Otkaz (n: Byte);
begin
 Case n of
     1: Write ('ОТКАЗ: RootBT (Null_Bin_Tree)!');
     2: Write ('ОТКАЗ: LeftBT (Null_Bin_Tree)!');
     3: Write ('ОТКАЗ: RightBT (Null_Bin_Tree)!');
     4: Write ('ОТКАЗ: исчерпана память!')
     else  Write ('ОТКАЗ: ?')
 end;
 Halt
end {Otkaz};

function CreateBT: BinT;
begin
    CreateBT := nil
end {CreateBT};

function NullBT (t: BinT): Boolean;
begin
    NullBT := (t = nil)
end {NullBT};

function RootBT (t: BinT): Elem;
begin
   if t <> nil then RootBT := t^.Info else Otkaz(1)
end {RootBT};

function LeftBT (t: BinT): BinT;
begin
  if t <> nil then LeftBT := t^.LSub  else Otkaz(2)
end {LeftBT};
```

```
function RightBT (t: BinT): BinT;
begin
   if t <> nil then RightBT := t^.RSub else Otkaz(3)
end {RightBT};

function ConsBT (e: Elem; LS, RS: BinT): BinT;
 var b: BinT;
begin
     if MaxAvail >= SizeOf (Node) then
      begin
       New (b);
       b^.Info:= e;
       b^.LSub:= LS;
       b^.RSub:= RS;
       ConsBT:= b
      end
     else
       Otkaz(4)
end {ConsBT};

procedure DestroyBT(var b: BinT);
begin
    if b <> nil then
     begin
       DestroyBt (b^.LSub);
       DestroyBt (b^.RSub);
       Dispose (b)
     end
end {DestroyBT};
begin
end.
```

Далее приведем пример использования модуля *BinTree*, в котором для ввода бинарного дерева из файла и вывода его на экран используется КЛП-обход бинарного дерева, описанный в 3.4.

```
{Пример работы с бинарными деревьями}
Uses BinTree;
var b: BinT;
    Fin: Text;

function EnterBt: BinT;
 {ввод узлов в КЛП-порядке и построение бинарного дерева}
 var c: Char;
begin
    Read (Fin, c);
    if c = '/'
     then EnterBT:= NilBT  {Create}
     else EnterBT:= ConsBT (c, EnterBT, EnterBT)
end {EnterBT};

procedure OutBT(b: BinT);
 {вывод узлов бинарного дерева в КЛП-порядке}
begin
 if not NullBT (b) then
  begin
     Write (RootBT (b));
     OutBT (LeftBT (b));
     OutBT (RightBT (b))
  end
 else Write ('/')
end {OutBT};

procedure DisplayBT (b: BinT);
 {построчный вывод повернутого изображения бинарного дерева}
begin
 if NullBt (b) then  {?}
    else
     begin
        Write (RootBT (b));
        if not NullBT (RightBT (b)) then
```

```
            begin
             Write (' ');              {вправо}
             DisplayBT (RightBT (b));
             Write (#8); Write (#8); {возврат влево}
            end;
        if not NullBT (LeftBT (b)) then
            begin
             Write (#10); {вниз}
             Write (' ');     {вправо}
             DisplayBT (LeftBT (b));
             Write (#8);  Write (#8); {возврат влево}
            end;
     end {else}
end {DisplayBT};
begin
    Assign (Fin, 'inbintree.dat');   Reset (Fin);
    b := EnterBT;
    WriteLn ('Построили бинарное дерево по его КЛП-представлению,',
          'вот КЛП-представление :');
    OutBT (b);       WriteLn;
    WriteLn ('... а теперь – вид дерева ...');
    DisplayBT (b);
    WriteLn;
    DestroyBt (b);
end.
```

Отметим, что в процедуре *DisplayBT* использованы особенности оператора *Write* языка Турбо-Паскаль: перемещение «каретки» на предыдущую позицию в строке – *Write* (#8) и на ту же позицию, но на следующей строке – *Write* (#10). Более универсальная процедура *DisplayBT*1, не использующая эти особенности языка Турбо-Паскаль, имеет следующий вид:

```
procedure  DisplayBT1 (b: BinT; n: integer);
{построчный вывод повернутого изображения бинарного дерева без возврата
   каретки}
```

```
{n – уровень узла}
var i: integer;
begin
  if NullBT (b) then  {Writeln}
    else
      begin
        Write (' ', RootBT (b));
        if NullBT (RightBT (b))
          then Writeln  {вниз}
          else DisplayBT1 (RightBT (b), n+1);
        if not NullBT (LeftBT (b)) then
            begin
              for i := 1 to n do Write ('  '); {вправо}
              DisplayBT1 (LeftBT (b), n+1);
            end;
        end;
 end {DisplayBT1}
```

Определение структуры данных бинарного дерева через описанный в 3.2 набор базовых операций естественно с математической точки зрения и удобно при функциональном (рекурсивном) стиле программирования (в особенности в языках функционального программирования, например в Лиспе). Однако при итеративном стиле программирования в случае необходимости модификации дерева возникают некоторые неудобства, связанные с употреблением функции *ConsBT*. Дело в том, что если перестраивать бинарное дерево, работая только через базовые функции, т. е. не используя особенности ссылочного представления и работу с указателями, то приходится сначала «разбирать» текущее поддерево с помощью селекторов, а затем «собирать» заново с помощью конструктора *ConsBT*. При этом производятся соответствующие операции с динамической памятью для замены старого узла на новый. Это приводит к дополнительным накладным расходам при выполнении программы, которых во многих случаях можно

избежать, вводя более удобный набор конструкторов. Например, полезно выделить как самостоятельные следующие дополнительные функции:

1) функцию *MakeRoot*, порождающую бинарное дерево из одного узла (корня);

2) функцию *SetLeft*, модифицирующую бинарное дерево таким образом, что на месте его пустого левого поддерева появляется заданный лист;

3) функцию *SetRight*, аналогичную *SetLeft*, но для правого поддерева.

Функциональная спецификация этих функций имеет вид:

$$MakeRoot: \alpha \to NonNullBT;$$

$$SetLeft: \alpha \otimes NonNullBT \to NonNullBT;$$

$$SetRight: \alpha \otimes NonNullBT \to NonNullBT$$

с набором аксиом:

- аксиомы для *MakeRoot* ($\forall u$: α):
  - $Root\ (MakeRoot\ (u)) = u$;
  - $Null\ (Left\ (MakeRoot\ (u))) = true$;
  - $Null\ (Right\ (MakeRoot\ (u))) = true$;
- аксиомы для *SetLeft* ($\forall u$: α; $\forall b$: *NonNullBT*: $Null\ (Left\ (b))$):
  - $Left\ (SetLeft\ (u, b)) = MakeRoot\ (u)$;
  - $Root\ (SetLeft\ (u, b)) = Root\ (b)$;
  - $Right\ (SetLeft\ (u, b)) = Right\ (b)$;
- аксиомы для *SetRight* ($\forall u$: α; $\forall b$:*NonNullBT*: $Null\ (Right\ (b))$):
  - $Right\ (SetRight\ (u, b)) = MakeRoot\ (u)$;
  - $Root\ (SetRight\ (u, b)) = Root\ (b)$;
  - $Left\ (SetRight\ (u, b)) = Left\ (b)$.

На уровне функциональной спецификации все эти функции могут быть выражены через функцию *ConsBT*. При этом приводимые далее соотношения справедливы для $\forall u$: α и для таких переменных $b1$ и $b2$, что $\forall b2$: *NonNullBT*: $Null\ (Right\ (b2))$; $\forall b1$: *NonNullBT*: $Null\ (Left\ (b1))$:

- $MakeRoot(u) = ConsBT(u, \Lambda, \Lambda)$;
- $SetLeft\ (u, b1) = ConsBT\ (Root\ (b1), MakeRoot\ (u), Right\ (b1))$;
- $SetRight\ (u, b2) = ConsBT\ (Root\ (b2), Left\ (b2), MakeRoot\ (u))$.

Абстрактные функции *SetLeft* и *SetRight* удобнее реализовать в виде процедур, например:

```
procedure SetLeft (a: Elem; b: BinT);
{Pred: Not Null (b) & Null (Left (b))}
Begin
    if Null (b) then Otkaz (5)
    else if Not Null (Left (b)) then Otkaz (6)
        else b^.LSub := MakeRoot (a)
end {SetLeft}
```

Рассмотрим ***ссылочную реализации ограниченного бинарного дерева на базе вектора***:

```
type Adr = 0 .. MaxAdr;          {диапазон «адресов» в векторе Mem}
     BinT = Adr;                 {представление бинарного дерева}
     Node = record               {узел: }
              Info: Elem;        {– содержимое}
              LSub: BinT;        {– левое  поддерево}
              RSub: BinT         {– правое поддерево}
     end {Node};
     Mem = array [Adr] of Node   {вектор для хранения дерева}
```

Здесь вектор типа *Mem* представляет собой память для хранения одного бинарного дерева (или нескольких бинарных деревьев, ограниченных в совокупности). Элемент вектора есть запись типа *Node*, содержащая узел и ссылки на поддеревья. На рис. 3.11 приведен пример представления бинарного дерева. Дерево представляется переменной b: *BinT*, причем значение $b = 3$ есть номер элемента массива, в котором хранится корень дерева. Фактически переменная b играет роль ссылки на корень дерева (или адреса корня в векторной памяти). Пустому дереву соответствовало бы значение $b = 0$, т. е. в этом представлении константа *NillBT* = 0. Элементы вектора, не занятые под хранение узлов дерева, образуют свободную память, которую удобно организовать в виде линейного циклического списка (для этого используется одно из полей звена *Node*, например, поле *LSub*). При этом элемент вектора с индексом (адресом) 0 играет роль ссылки на начало списка свободной памяти.

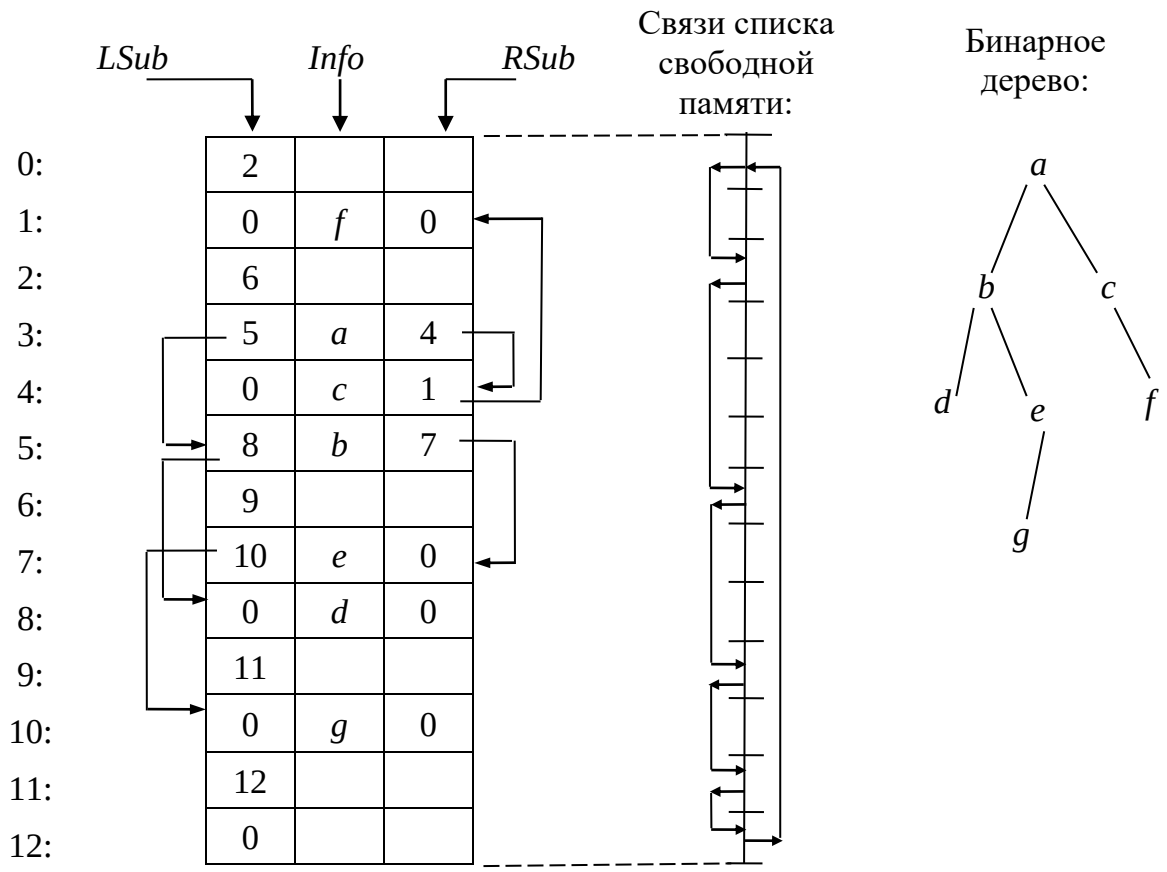


Рис. 3.11. Пример ссылочного представления бинарного дерева на базе вектора. *Справа* – бинарное дерево; *слева* – его размещение в массиве, играющем роль динамической памяти

Более подробное описание данной реализации рекомендуется в качестве самостоятельного упражнения, поскольку она во многом аналогична ссылочной реализации линейного списка на базе вектора, рассмотренной в 1.4.

Упражнения

При выполнении упражнений следует использовать вариант реализации бинарного дерева (см. 3.5), соответствующий варианту.

Для представления деревьев во входных данных рекомендуется использовать скобочную запись, кроме случаев, специально оговоренных в

условии задачи. В выходных данных рекомендуется представлять дерево (лес) в горизонтальном виде (т. е. с поворотом на 90°), а бинарные деревья – в виде уступчатого списка.

В заданиях 1 – 5, в зависимости от варианта, предлагается реализовать рекурсивные или нерекурсивные процедуры (функции); в последнем случае следует использовать стек и операции над ним.

1. Задано бинарное дерево *b* типа *BT* с типом элементов *Elem*. Для введенной пользователем величины *E* (**var** *E*: *Elem*):

- определить, входит ли элемент *E* в дерево *b*;

- определить число вхождений элемента *E* в дерево *b*;

- найти в дереве *b* длину пути (число ветвей) от корня до ближайшего узла с элементом *E* (если *E* не входит в *b*, за ответ принять –1).

2. Для заданного бинарного дерева *b* типа *BT* с произвольным типом элементов:

- определить максимальную глубину дерева *b*, т. е. число ветвей в самом длинном из путей от корня дерева до листьев;

- вычислить длину внутреннего пути дерева *b*, т. е. сумму по всем узлам длин путей от корня до узла.

3. Для заданного бинарного дерева *b* типа *BT* с произвольным типом элементов:

- напечатать элементы из всех листьев дерева *b*;

- подсчитать число узлов на заданном уровне *n* дерева *b* (корень считать узлом 1-го уровня).

4. Для заданного бинарного дерева *b* типа *BT* с произвольным типом элементов определить, есть ли в дереве *b* хотя бы два одинаковых элемента.

5. Заданы два бинарных дерева *b*1 и *b*2 типа *BT* с произвольным типом элементов. Проверить:

- подобны ли они (два бинарных дерева ***подобны***, если они оба пусты либо они оба непусты и их левые поддеревья подобны и правые поддеревья подобны);

- равны ли они (два бинарных дерева ***равны***, если они подобны и их соответствующие элементы равны);

- зеркально подобны ли они (два бинарных дерева ***зеркально подобны***, если они оба пусты либо они оба непусты и для каждого из них левое поддерево одного подобно правому поддереву другого);

- симметричны ли они (два бинарных дерева ***симметричны***, если они зеркально подобны и их соответствующие элементы равны).

6. Задано бинарное дерево *b* типа *BT* с произвольным типом элементов. Используя очередь, напечатать все элементы дерева *b* по уровням: сначала – из корня дерева, затем (слева направо) – из узлов, сыновних по отношению к корню, затем (также слева направо) – из узлов, сыновних по отношению к этим узлам, и т. д.

7. Для заданного леса с произвольным типом элементов:

- получить естественное представление леса бинарным деревом;
- вывести изображение леса и бинарного дерева;
- перечислить элементы леса в горизонтальном порядке (в ширину).

8. (Обратная задача.) Для заданного бинарного дерева с произвольным типом элементов:

- получить лес, естественно представленный этим бинарным деревом;
- вывести изображение бинарного дерева и леса;
- перечислить элементы леса в горизонтальном порядке (в ширину).

9. Рассматриваются бинарные деревья с элементами типа *Elem* (в качестве *Elem* использовать *char*). Заданы перечисления узлов некоторого дерева *b* в порядке КЛП и ЛКП. Требуется:

- восстановить дерево *b* и вывести его изображение;
- перечислить узлы дерева *b* в порядке ЛПК.

10. Рассматриваются бинарные деревья с элементами типа *Elem* (в качестве *Elem* использовать *char*). Заданы перечисления узлов некоторого дерева *b* в порядке ЛКП и ЛПК. Требуется:

- восстановить дерево *b* и вывести его изображение;
- перечислить узлы дерева *b* в порядке КЛП.

11–17. Формулу вида

< формула > ::= < терминал > | (< формула > < знак > < формула >)

< знак > ::= + | – | *

< терминал > ::= 0 | 1 | ... | 9 | *a* | *b* | ... | *z*

можно представить в виде бинарного дерева («***дерева-формулы***») с элементами типа *Elem=char* согласно следующим правилам:

- формула из одного терминала представляется деревом из одной вершины с этим терминалом;

- формула вида (f_1 *s* f_2) представляется деревом, в котором корень – это знак *s*, а левое и правое поддеревья – соответствующие представления формул f_1 и f_2. Например, формула (5 * (*a* + 3)) представляется деревом-формулой, показанной на рис. 3.12.

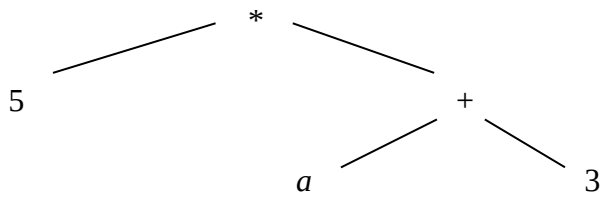


Рис. 3.12. Дерево-формула

Требуется:

- Для всех вариантов (11–17):

- для заданной формулы *f* построить дерево-формулу *t*;

- для заданного дерева-формулы *t* напечатать соответствующую формулу *f*;

Вариант 11:

- с помощью построения дерева-формулы *t* преобразовать заданную формулу *f* из инфиксной формы в префиксную (перечисление узлов *t* в порядке КЛП) и в постфиксную (перечисление в порядке ЛПК);

- если в дереве-формуле *t* терминалами являются только цифры, то вычислить (как целое число) значение дерева-формулы *t*.

Вариант 12:

- построить дерево-формулу *t* из строки, задающей формулу в префиксной форме (перечисление узлов *t* в порядке КЛП);

- преобразовать дерево-формулу *t*, заменяя в нем все поддеревья, соответствующие формуле (*f* + *f*), на поддеревья, соответствующие формуле (*2* * *f*).

Вариант 13:

- построить дерево-формулу *t* из строки, задающей формулу в постфиксной форме (перечисление узлов *t* в порядке ЛПК);

- упростить дерево-формулу *t*, выполнив в нем все операции вычитания, в которых уменьшаемое и вычитаемое – цифры. Результат вычитания – цифра или формула вида (*0* – цифра).

Вариант 14:

- преобразовать дерево-формулу t, заменяя в нем все поддеревья, соответствующие формулам $(f_1 * (f_2 + f_3))$ и $((f_1 + f_2) * f_3)$, на поддеревья, соответствующие формулам $((f_1 * f_2) + (f_1 * f_3))$ и $((f_1 * f_3) + (f_2 * f_3))$.

Вариант 15:

- преобразовать дерево-формулу t, заменяя в нем все поддеревья, соответствующие формулам $((f_1 * f_2) + (f_1 * f_3))$ и $((f_1 * f_3) + (f_2 * f_3))$, на поддеревья, соответствующие формулам $(f_1 * (f_2 + f_3))$ и $((f_1 + f_2) * f_3)$;

Вариант 16:

- упростить дерево-формулу t, заменяя в нем все поддеревья, соответствующие формулам $(f + 0)$, $(0 + f)$, $(f - 0)$, $(f * 1)$, $(1 * f)$, на поддеревья, соответствующие формуле f; поддеревья, соответствующие формулам $(f * 0)$, $(0 * f)$ и $(f - f)$, – на узел с 0; поддеревья, соответствующие формулам $(f_1 + (0 - f_2))$ и $((0 - f_2) + f_1)$, – на поддеревья, соответствующие формуле $(f_1 - f_2)$; поддеревья, соответствующие формуле $(f_1 - (0 - f_2))$, – на поддеревья, соответствующие формуле $(f_1 + f_2)$. Предусмотреть возможность упрощения в несколько этапов.

Вариант 17:

- построить дерево-формулу $t1$ – производную дерева-формулы t по заданной переменной.

18. Бинарное дерево называется бинарным деревом поиска, если для каждого его узла справедливо: все элементы правого поддерева больше этого узла, а все элементы левого поддерева – меньше этого узла.

Бинарное дерево называется пирамидой, если для каждого его узла справедливо: значения всех потомков этого узла не больше, чем значение узла.

Для заданного бинарного дерева с числовым типом элементов определить, является ли оно бинарным деревом поиска и является ли оно пирамидой.