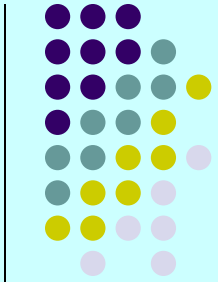
Тема 4
НЕЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ



Нелинейные структуры данных позволяют выражать более сложные отношения между элементами, нежели линейные отношения соседства, как это было в линейных списках.

К нелинейным структурам данных относят

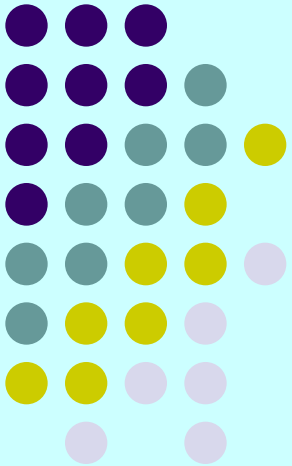
- графы,
- деревья
- и леса.

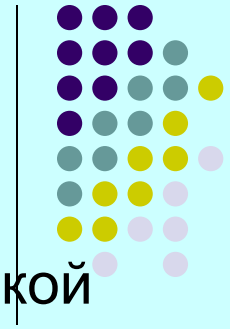
Эти структуры находят широкое применение при решении практических задач.

В данной теме достаточно полно будут рассмотрены способы построения и представления деревьев и основные операции над ними.

Вопросы представления графов и алгоритмы на графах рассмотрим позже.

Деревья





Деревья представляют собой иерархическую структуру некой совокупности элементов.

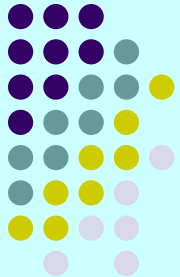
Примерами деревьев могут служить генеалогические и организационные диаграммы.

Деревья используются при анализе электрических цепей, при представлении структур математических формул.

Они также естественным путем возникают во многих областях компьютерных наук.

Например, деревья используются для организации информации в системах управления базами данных и для представления синтаксических структур в компиляторах программ.

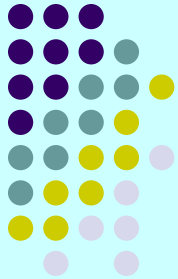
Основная терминология



Дерево - это совокупность элементов, называемых ***узлами*** (один из которых определен как корень), и ***отношений*** ("родительских"), образующих иерархическую структуру узлов.

Узлы, так же, как и элементы списков, могут быть элементами любого типа. Мы часто будем изображать узлы буквами, строками или числами.

Основная терминология



Формально дерево можно рекуррентно определить следующим образом:

1. Один узел является деревом. Этот же узел также является корнем этого дерева.
2. Пусть n - это узел, а T_1, T_2, …, T_k - деревья с корнями n_1, n_2, …, n_k соответственно. Можно построить новое дерево, сделав n родителем узлов n_1, n_2, …, n_k. В этом дереве n будет корнем, а T_1, T_2, …, T_k - поддеревьями этого корня. Узлы n_1, n_2, …, n_k называются сыновьями узла n.

Часто в это определение включают понятие ***нулевого дерева***, т. е. "дерева" без узлов, такое дерево мы будем обозначать символом **Λ**.

Пример 1

Рассмотрим оглавление книги, схематически представленное на рис. 1, а. Это оглавление является деревом, которое в другой форме показано на рис. 1, б.

Отношение родитель-сын отображается в виде линии. Деревья обычно рисуются сверху вниз, как на рис. 1, б, так, что родители располагаются выше "детей".

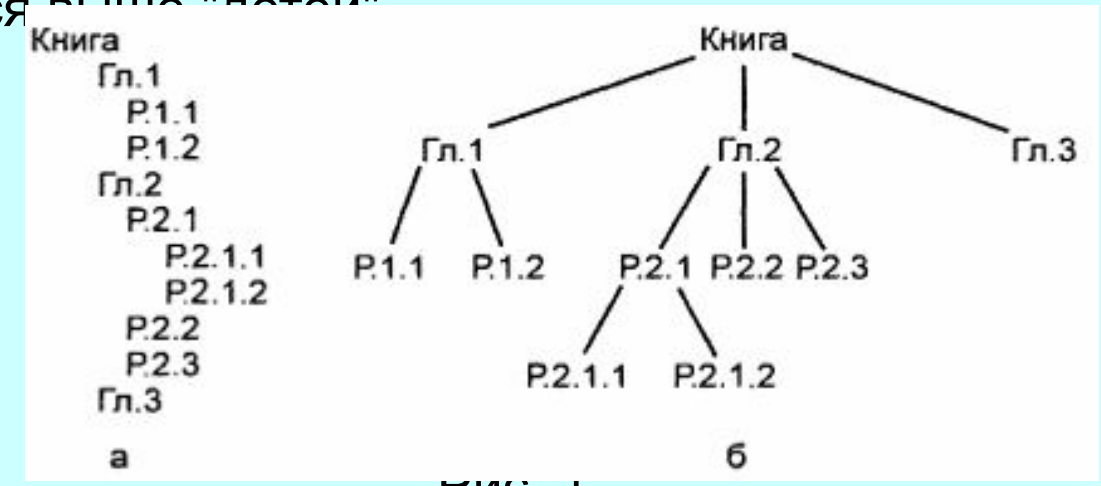


Рис. 1

Пример 1 показывает типичные данные, для которых наилучшим представлением будут деревья. В этом примере родительские отношения устанавливают подчиненность глав и разделов: родительский узел связан с узлами сыновей (и указывает на них), как узел ***Книга*** подчиняет себе узлы ***Гл.1***, ***Гл.2*** и ***Гл.3***.

Основная терминология

Путем из узла n_1 в узел n_k называется последовательность узлов n_1, n_2, …, n_k, где для всех i, $1 \le i \le k$, узел n_i является родителем узла n_{i+1}.

Длиной пути называется число, на единицу меньшее числа узлов, составляющих этот путь. Таким образом, путем нулевой длины будет путь из любого узла к самому себе. На рис. 1 путем длины 2 будет, например, путь от узла ***Гл.2*** к узлу ***Р.2.1.2***.

Основная терминология

Если существует путь из узла ***a*** в ***b***, то в этом случае узел ***a*** называется ***предком*** узла ***b***, а узел ***b*** - ***потомком*** узла ***a***.

Например, на рис. 1 предками узла ***P.2.1*** будут следующие узлы: сам узел ***P.2.1*** и узлы ***Гл.2*** и ***Книга***, тогда как потомками этого узла являются опять сам узел ***P.2.1*** и узлы ***P.2.1.1*** и ***P.2.1.2***.

Отметим, что любой узел одновременно является и предком, и потомком самого себя.

Предок или потомок узла, не являющийся таковым самого себя, называется ***истинным предком*** или ***истинным потомком*** соответственно.

В дереве только корень не имеет истинного предка.

Узел, не имеющий истинных потомков, называется ***листом***.

Теперь поддерево какого-либо дерева можно определить как узел (корень поддерева) вместе со всеми его потомками.

Основная терминология

Высотой узла дерева называется длина самого длинного пути из этого узла до какого-либо листа.

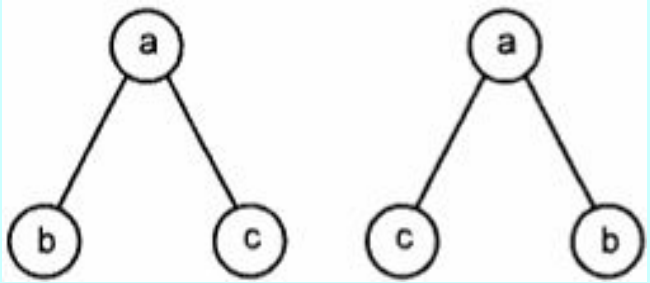
На рис.1 высота узла ***Гл.1*** равна 1, узла ***Гл.2*** - 2, а узла ***Гл.3*** - 0.

Высота дерева совпадает с высотой корня.

Глубина узла определяется как длина пути (он единственный) от корня до этого узла.

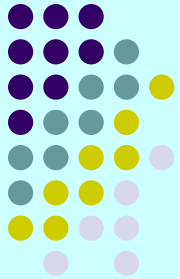
Порядок узлов

- Сыновья узла обычно упорядочиваются слева направо. Поэтому два дерева на рисунке различны, так как порядок сыновей узла ***a*** различен.



- Если порядок сыновей игнорируется, то такое дерево называется ***неупорядоченным***, в противном случае дерево называется ***упорядоченным***.

Порядок узлов



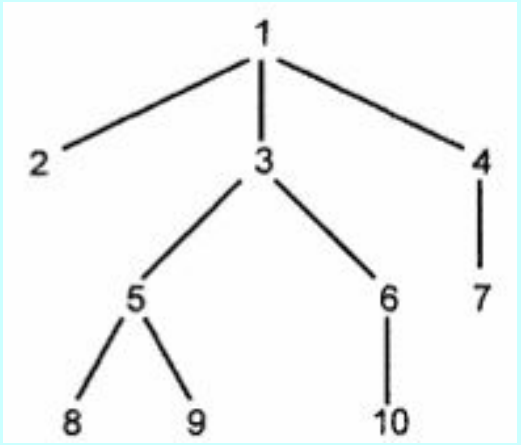
- Упорядочивание слева направо сыновей ("родных детей" одного узла) можно использовать для сопоставления узлов, которые не связаны отношениями предки-потомки.
- Соответствующее правило звучит следующим образом:

  если узлы ***a*** и ***b*** являются сыновьями одного родителя и узел ***a*** лежит слева от узла ***b***, то все потомки узла ***a*** будут находиться слева от любых потомков узла ***b***.

Пример 2.

Рассмотрим дерево на рисунке.

Узел 8 расположен справа от узла 2, слева от узлов 9, 6, 10, 4 и 7 и не имеет отношений справа-слева относительно предков 1, 3 и 5.



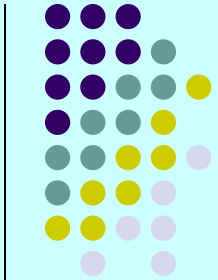
Существует простое правило, позволяющее определить, какие узлы расположены слева от данного узла ***n***, а какие - справа.

Для этого надо прочертить путь от корня дерева до узла ***n***.

Тогда все узлы и их потомки, расположенные слева от этого пути, будут находиться слева от узла ***n***.

И, аналогично, все узлы и их потомки, расположенные справа от этого пути, будут находиться справа от узла ***n***.

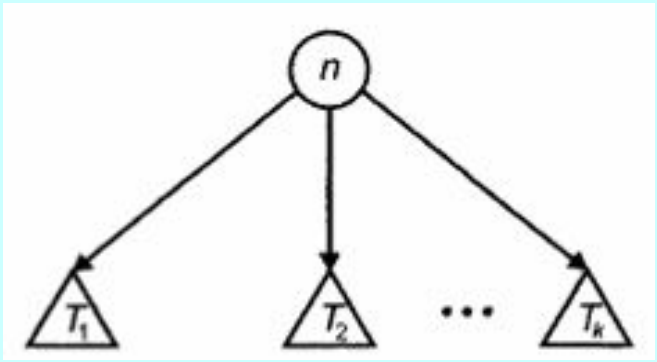
Прямой, обратный и симметричный обходы дерева



- Существует несколько способов обхода (прохождения) всех узлов дерева. ***Обход узлов дерева*** равнозначен упорядочиванию по какому-либо правилу этих узлов. Поэтому в данном разделе мы будем использовать слова "обход узлов" и "упорядочивание узлов" как синонимы.
- Три наиболее часто используемых способа обхода дерева называются
  - ***обход в прямом порядке***,
  - ***обход в обратном порядке***
  - и ***обход во внутреннем порядке*** (последний вид обхода также часто называют ***симметричным обходом***).
- Все три способа обхода рекурсивно можно определить следующим образом.

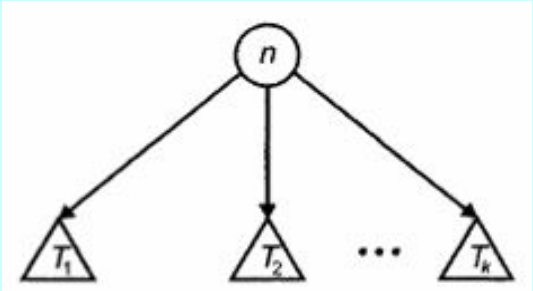
Прямой, обратный и симметричный обходы дерева

- Если дерево ***T*** является нулевым деревом, то в список обхода заносится пустая запись.
- Если дерево ***T*** состоит из одного узла, то в список обхода записывается этот узел.
- Далее, пусть ***T*** - дерево с корнем ***n*** и поддеревьями T_1, T_2, ..., T_k, как показано на рисунке.



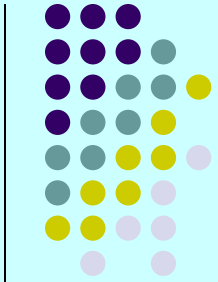
Прямой, обратный и симметричный обходы дерева

- Тогда для различных способов обхода имеем следующее.



1. При прохождении в ***прямом порядке*** (т.е. при прямом упорядочивании) узлов дерева ***T*** сначала посещается корень ***n***, затем узлы поддерева T_1, далее все узлы поддерева T_2 и т. д. Последними посещаются узлы поддерева T_k.
2. При ***симметричном обходе*** узлов дерева ***T*** сначала посещаются в симметричном порядке все узлы поддерева T_1, далее корень ***n***, затем последовательно в симметричном порядке все узлы поддеревьев T_2, …, T_k.
3. Во время обхода в ***обратном порядке*** сначала посещаются в обратном порядке все узлы поддерева T_1, затем последовательно посещаются все узлы поддеревьев T_2, …, T_k, также в обратном порядке, последним посещается корень ***n***.

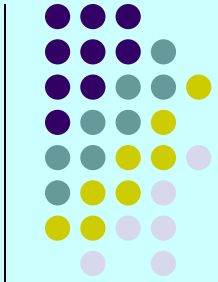
Листинг 4.1. Рекурсивные процедуры обхода деревьев



```
procedure  PREORDER ( n: узел );
begin
(1) занести в список обхода узел n;
(2) для каждого сына c узла n в порядке слева направо
    do
        PREORDER(c)
end;        { PREORDER }
```

- Здесь показан набросок процедуры ***PREORDER*** (Прямое упорядочивание), составляющей список узлов дерева при обходе его в прямом порядке.
- Чтобы из этой процедуры сделать процедуру, выполняющую обход дерева в обратном порядке, надо просто поменять местами строки (1) и (2).

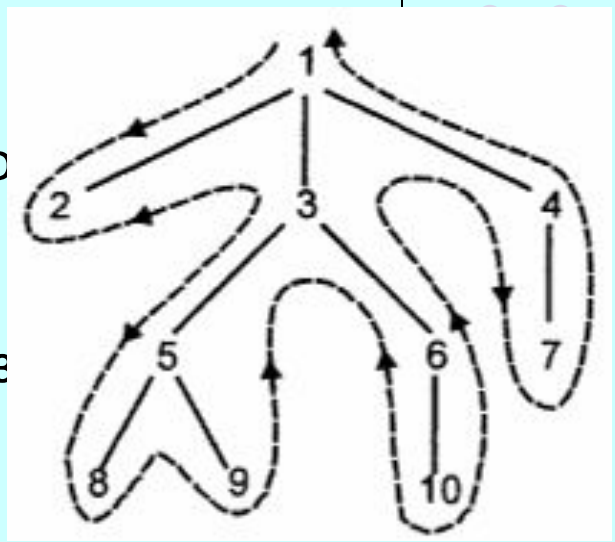
Листинг 4.1. Рекурсивные процедуры обхода деревьев



```
procedure   INORDER ( n: узел );
begin
    if   n - лист   then
     занести в список обхода узел n;
    else    begin
     INORDER(самый левый сын узла n);
     занести в список обхода узел n;
     для каждого сына с узла n, исключая самый
    левый, в порядке слева направо   do
                INORDER(c)
    end
end;        { INORDER }
```

Здесь представлен набросок процедуры ***INORDER*** (Внутреннее упорядочивание).

Пример 3. Прямой обход дерева



Сначала запишем (посетим) узел 1, затем вызовем процедуру ***PREORDER*** для обхода первого поддерева с корнем 2. Это поддерево состоит из одного узла, которое мы и записываем.

Далее обходим второе поддерево, выходящее из корня 1, это поддерево имеет корень 3. Записываем узел 3 и вызываем процедуру ***PREORDER*** для обхода первого поддерева, выходящего из узла 3. В результате получим список узлов 5, 8 и 9 (именно в таком порядке).

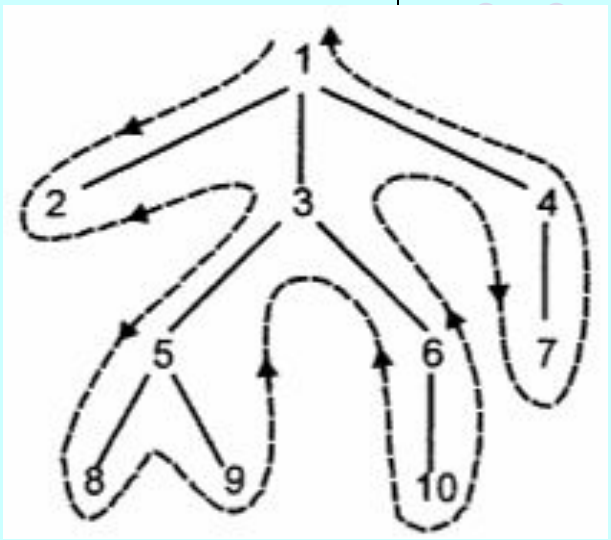
Продолжая этот процесс, в конце мы получим полный список узлов, посещаемых при прохождении в прямом порядке исходного дерева:

1, 2, 3, 5, 8, 9, 6, 10, 4 и 7.

Пример 3. Обратный и симметричный обходы дерева

- Подобным образом, предварительно преобразовав процедуру ***PREORDER*** в процедуру, выполняющую обход в обратном порядке (как указано выше), можно получить обратное упорядочивание узлов дерева в следующем виде:

  2, 8, 9, 5, 10, 6, 3, 7, 4 и 1.

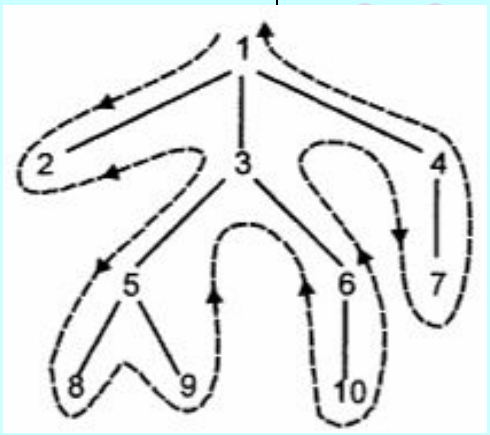


- Применяя процедуру ***INORDER***, получим список симметрично упорядоченных узлов этого же дерева:

  2, 1, 8, 5, 9, 3, 10, 6, 7 и 4.

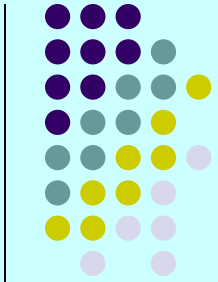
Пример 3. Обратный и симметричный обходы дерева

- При обходе деревьев можно применить следующий полезный прием. Надо нарисовать непрерывный контур вокруг дерева, начиная от корня дерева, рисуя контур против часовой стрелки и поочередно обходя все наружные части дерева.
- При ***прямом упорядочивании узлов*** надо просто записать их в соответствии с нарисованным контуром.



- При ***обратном упорядочивании*** после записи всех сыновей переходим к их родителю.
- При ***симметричном (внутреннем) упорядочивании*** после записи самого правого листа мы переходим не по ветви в направлении к корню дерева, а к следующему "внутреннему" узлу, который еще не записан.
- При любом упорядочивании листья всегда записываются в порядке слева направо, при этом в случае симметричного упорядочивания между сыновьями может быть записан родитель.

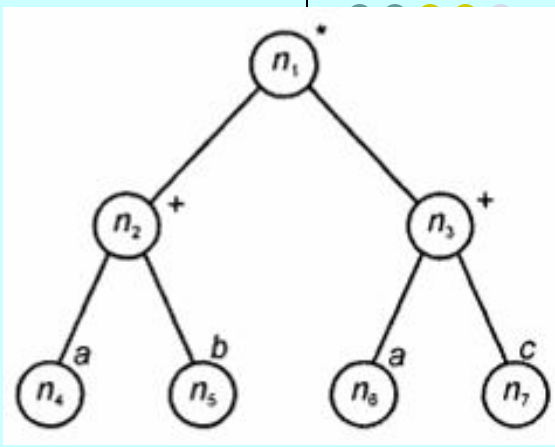
Помеченные деревья и деревья выражений



- Часто бывает полезным сопоставить каждому узлу дерева метку (***label***) или значение, точно так же, как мы в предыдущей теме сопоставляли элементам списков определенные значения.
- Дерево, у которого узлам сопоставлены метки, называется ***помеченным деревом***.
- ***Метка узла*** - это не имя узла, а значение, которое "хранится" в узле.
- В некоторых приложениях требуется изменять значение метки, поскольку имя узла сохраняется постоянным.
- Полезна следующая аналогия:

  дерево-список, узел-позиция, метка-элемент.

Пример 4. Дерево выражения с метками



На рисунке показано дерево с метками, представляющее арифметическое выражение (a+b)*(a+c), где n_1, ..., n_7 - имена узлов (метки на рисунке проставлены рядом с соответствующими узлами).

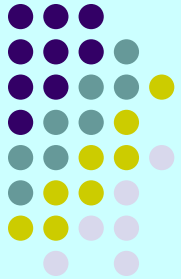
Правила соответствия меток деревьев элементам выражений следующие.

1. Метка каждого листа соответствует операнду и содержит его значение, например узел n_4 представляет операнд ***a***.

2. Метка каждого внутреннего (родительского) узла соответствует оператору. Предположим, что узел ***n*** помечен бинарным оператором **θ** (например, **+** или *) и левый сын этого узла соответствует выражению E_1, а правый - выражению E_2. Тогда узел ***n*** и его сыновья представляют выражение $(E_1)\theta(E_2)$. Можно удалять родителей, если это необходимо.

- Например, узел n_2 имеет оператор **+**, а левый и правый сыновья представляют выражения (операнды) ***a*** и ***b*** соответственно. Поэтому узел n_2 представляет (***a***) + (***b***), т.е. ***a***+***b***.
- Узел n_1 представляет выражение (***a***+***b***)*(***a***+***c***), поскольку оператор * является меткой узла n_1, выражения ***a***+***b*** и ***a***+***c*** представляются узлами n_2 и n_3 соответственно.

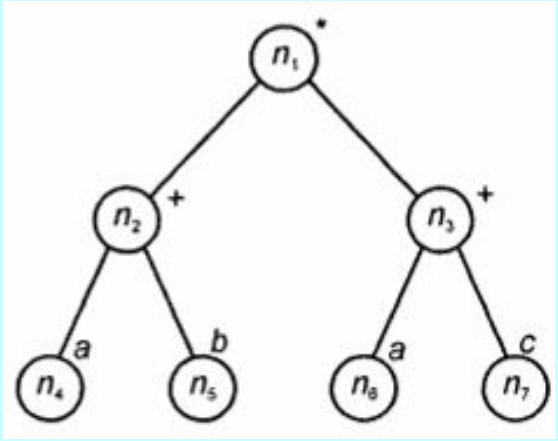
Помеченные деревья и деревья выражений



- Часто при обходе деревьев составляется список не имен узлов, а их меток.
- В случае дерева выражений при прямом упорядочивании получаем известную ***префиксную форму выражений***, где оператор предшествует и левому, и правому операндам.
- Для точного описания префиксной формы выражений сначала положим, что префиксным выражением одиночного операнда ***a*** является сам этот операнд.
- Далее, префиксная форма для выражения $(E_1)\theta(E_2)$, где θ - бинарный оператор, имеет вид $\theta P_1 P_2$, здесь P_1, P_2 - префиксные формы для выражений E_1, E_2.

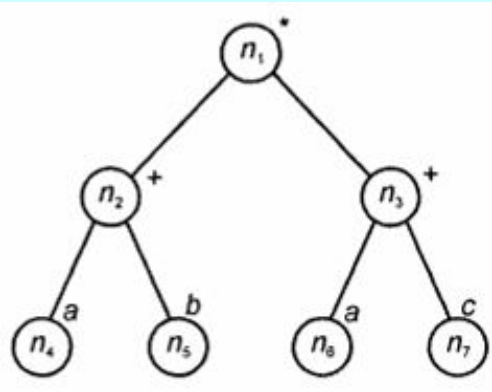
Помеченные деревья и деревья выражений

- Отметим, что в префиксных формах нет необходимости отделять или выделять отдельные префиксные выражения скобками, так как всегда можно просмотреть префиксное выражение $\theta P_1 P_2$ и определить единственным образом P_1 как самый короткий префикс выражения $P_1 P_2$.
- Например, при ***прямом упорядочивании узлов*** (точнее, меток) дерева, показанного на рис., получаем префиксное выражение ***+ab+ac**. Самым коротким корректным префиксом для выражения **+ab+ac** будет префиксное выражение узла n_2: **+ab**.



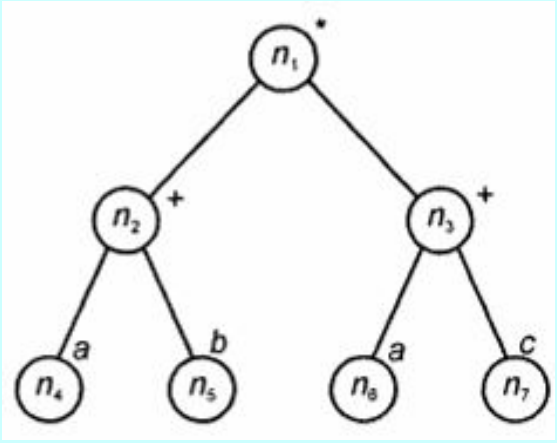
Помеченные деревья и деревья выражений

- ***Обратное упорядочивание меток дерева*** выражений дает так называемое ***постфиксное представление выражений***.
- Выражение $\theta P_1 P_2$ в постфиксной форме имеет вид $P_1 P_2 \theta$, где P_1, P_2 - постфиксные формы для выражений E_1 и E_2 соответственно.
- При использовании постфиксной формы в применении скобок нет необходимости, поскольку для любого постфиксного выражения $P_1 P_2$ легко проследить самый короткий суффикс P_2, что и будет корректным составляющим постфиксным выражением.
- Например, постфиксная форма выражения для дерева на рис. имеет вид ***ab+ac+****. Если записать это выражение как $P_1 P_2$*, то P_2 (т.е. выражение *ac*+) будет самым коротким суффиксом для ***ab+ac+*** и, следовательно, корректным составля постфиксным выражением.



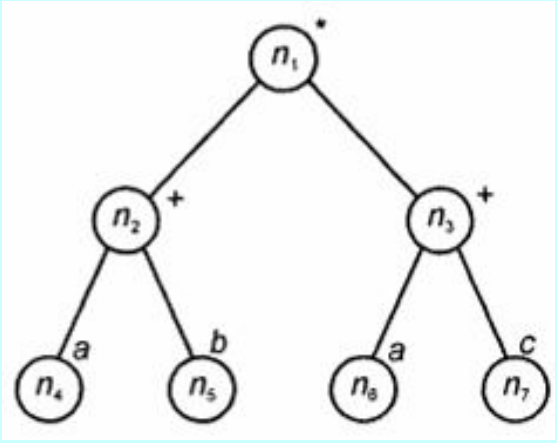
Помеченные деревья и деревья выражений

- При ***симметричном обходе дерева*** выражений получим так называемую ***инфиксную форму выражения***, которая совпадает с привычной "стандартной" формой записи выражений, но также не использует скобок. Для дерева на рис. инфиксное выражение запишется как ***a+b * a+c***.
- Таким образом, требуется разработать алгоритм обхода дерева выражений, который бы выдавал инфиксную форму выражения со всеми необходимыми парами скобок.

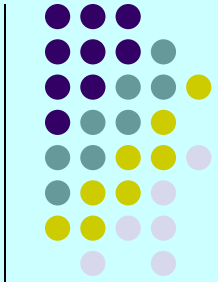


Вычисление „наследственных" данных

- Обход дерева в прямом или обратном порядке позволяет получить данные об отношениях предок-потомок узлов дерева.
- Пусть функция ***postorder(n)*** вычисляет позицию узла ***n*** в списке узлов, упорядоченных в обратном порядке. Например, для узлов n_2, n_4 и n_5 дерева, представленного на рис., значения этой функции будут 3, 1 и 2 соответственно.
- Определим также функцию ***desc(n)***, значение которой равно числу истинных потомков узла ***n***.



Вычисление „наследственных" данных

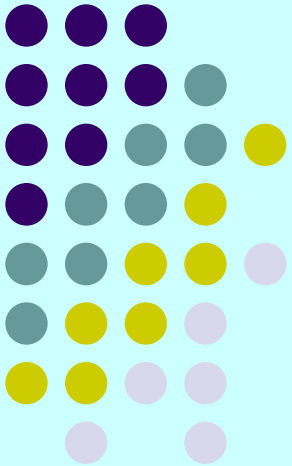


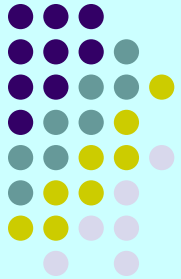
- Эти функции позволяют выполнить ряд полезных вычислений.
- Например, все узлы поддерева с корнем ***n*** будут последовательно занимать позиции от ***postorder(n)–desc(n)*** до ***postorder(n)*** в списке узлов, упорядоченных в обратном порядке.
- Для того чтобы узел ***x*** был потомком узла ***y***, надо, чтобы выполнялись следующие неравенства:

  $$postorder(y) - desc(y) \le postorder(x) \le postorder(y).$$

- Подобные функции можно определить и для списков узлов, упорядоченных в прямом порядке.

Абстрактный тип данных TREE (дерево)





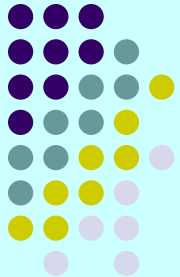
В теме 4 списки, стеки и очереди получили трактовку как абстрактные типы данных (АТД). В этой теме рассмотрим деревья как АТД и как структуры данных.

Одно из наиболее важных применений деревьев – это использование их при разработке реализаций различных АТД.

Далее мы покажем, как можно использовать "дерево двоичного поиска" при реализации АТД, основанных на математической модели множеств. Будут представлены примеры применения деревьев при реализации различных абстрактных типов данных.

- В этом разделе мы представим несколько полезных операторов, выполняемых над деревьями, и покажем, как использовать эти операторы в различных алгоритмах.

Операторы, выполняемые над деревьями



1. ***PARENT(n, T).***

   Эта функция возвращает родителя (parent) узла ***n*** в дереве ***T***.

   Если ***n*** является корнем, который не имеет родителя, то в этом случае возвращается **Λ** ("нулевой узел“), что указывает на то, что мы выходим за пределы дерева.

2. ***LEFTMOST_CHILD(n, T).***

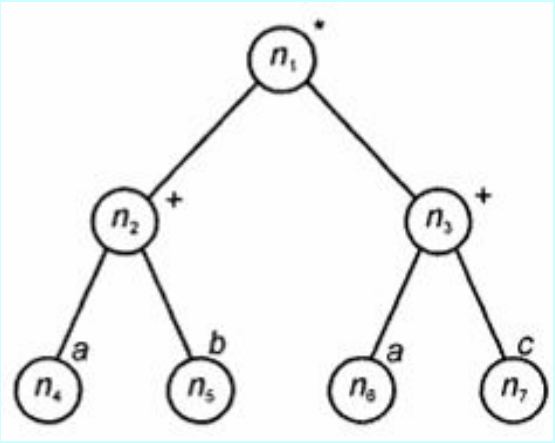
   Данная функция возвращает самого левого сына узла ***n*** в дереве ***T***. Если ***n*** является листом (и поэтому не имеет сына), то возвращается **Λ**.

Операторы, выполняемые над деревьями

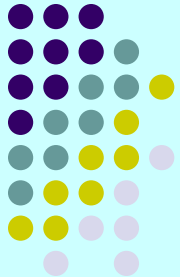
3. ***RIGHT_SIBLING(n, T).***

Эта функция возвращает правого брата узла ***n*** в дереве ***T*** и значение **Λ**, если такового не существует. Для этого находится родитель ***p*** узла ***n*** и все сыновья узла ***p***, затем среди этих сыновей находится узел, расположенный непосредственно справа от узла ***n***.

Например, для дерева на рис.
LEFTMOST_CHILD(n_2) = n_4,
RIGHT_SIBLING(n_4) = n_5,
RIGHT_SIBLING(n_5) = Λ.



Операторы, выполняемые над деревьями



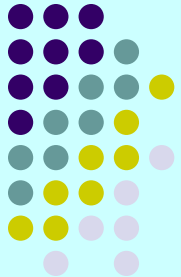
4. ***LABEL(n, T).***

Возвращает метку узла ***n*** дерева ***T***. Для выполнения этой функции требуется, чтобы на узлах дерева были определены метки.

5. ***CREATEi(v, T_1, T_2, ..., T_i)***

- это обширное семейство "созидающих" функций, которые для каждого ***i***=0, 1, 2, ... создают новый корень ***r*** с меткой ***v*** и далее для этого корня создает ***i*** сыновей, которые становятся корнями поддеревьев ***T_1, T_2, ..., T_i***.
- Эти функции возвращают дерево с корнем ***r***.
- Для ***i***=0 возвращается один узел ***r***, который одновременно является и корнем, и листом.

Операторы, выполняемые над деревьями



6. ***ROOT(T).***

   Возвращает узел, являющимся корнем дерева ***T***. Если ***T*** - пустое дерево, то возвращается **Λ**.

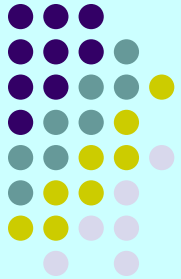
7. ***MAKENULL(T).***

   Этот оператор делает дерево ***T*** пустым деревом.

Пример 5.

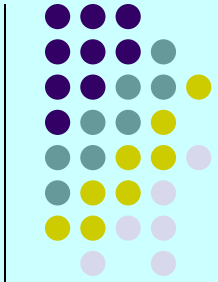
- Напишем рекурсивную ***PREORDER*** и нерекурсивную ***NPREORDER*** процедуры обхода дерева в прямом порядке и составления соответствующего списка его меток.
- Предположим, что для узлов определен тип данных ***node*** (узел), так же, как и для типа данных ***TREE*** (Дерево), причем ***АТД TREE*** определен для деревьев с метками, которые имеют тип данных ***labeltype*** (тип метки).
- В листинге 4.2 приведена рекурсивная процедура, которая по заданному узлу ***n*** создает список в прямом порядке меток поддерева, корнем которого является узел ***n***.
- Для составления списка всех узлов дерева ***T*** надо выполнить вызов ***PREORDER(ROOT(T))***.

Листинг 4.2. Рекурсивная процедура обхода дерева в прямом порядке



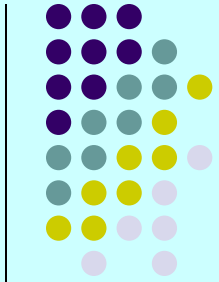
```
procedure   PREORDER ( n: node );
var   c: node;
begin
    print(LABEL(n, T) ) ;
    c: = LEFTMOST_CHILD(n, T) ;
    while c <> Λ do begin
     PREORDER(c);
     c:= RIGHT_SIBLING(c, T)
    end
end;          { PREORDER }
```

Пример 5.



- Теперь напишем нерекурсивную процедуру для печати узлов дерева в прямом порядке.
- Чтобы совершить обход дерева, используем стек ***S***, чей тип данных ***STACK*** уже объявлен как "стек для узлов".
- Основная идея разрабатываемого алгоритма заключается в том, что, когда мы дошли до узла ***n***, стек хранит путь от корня до этого узла, причем корень находится на "дне" стека, а узел ***n*** - в вершине стека.
- Один и подходов к реализации обхода дерева в прямом порядке показан на примере программы ***NPREORDER*** в листинге 4.3.

Пример 5.

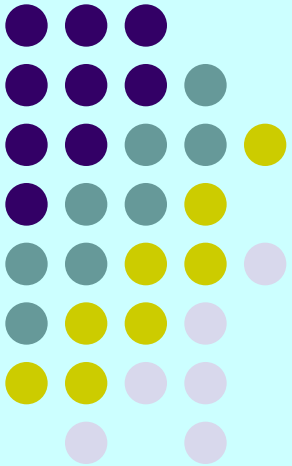


- Программа ***NPREORDER*** выполняет два вида операций, т.е. может находиться как бы в одном из двух режимов.
- Операции первого вида (первый режим) осуществляют обход по направлению к потомкам самого левого еще не проверенного пути дерева до тех пор, пока не встретится лист, при этом выполняется печать узлов этого пути и занесение их в стек.
- Во втором режиме выполнения программы осуществляется возврат по пройденному пути с поочередным извлечением узлов из стека до тех пор, пока не встретится узел, имеющий еще "не описанного" правого брата. Тогда программа опять переходит в первый режим и исследует новый путь, начиная с этого правого брата.
- Программа начинается в первом режиме с нахождения корня дерева и определения, является ли стек пустым.

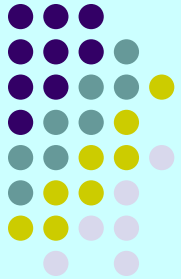
Листинг 4.3. Нерекурсивная процедура обхода дерева в прямом порядке

```
procedure   NPREORDER ( T: TREE );
var  m: node;          {переменная для временного хранения узлов}
     S: STACK;      {стек узлов, хранящий путь от корня до
            родителя TOP(S) текущего узла m }
begin        { инициализация }
     MAKENULL(S);          m:= ROOT (T) ;
     while   true   do
          if   m <> Λ   then   begin
               print(LABEL(m, T));      PUSH(m, S) ;
               { исследование самого левого сына узла m }
               m:= LEFTMOST_CHILD(m, T)
          end
          else   begin
               { завершена проверка пути, содержащегося в стеке }
               if   EMPTY(S)   then  return;
               { исследование правого брата узла, находящегося в вершине
                  стека }
               m:= RIGHT_SIBLING(TOP(S), T);
               POP(S)
          end
end;          { NPREORDER }
```

Реализация деревьев



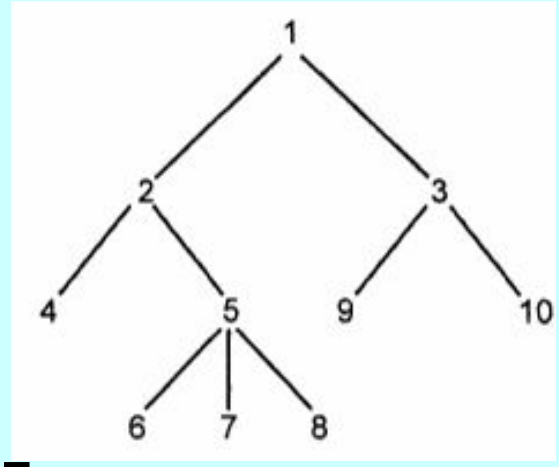
Представление деревьев с помощью массивов



- Пусть ***T*** - дерево с узлами 1, 2, ..., ***n***.
- Возможно, самым простым представлением дерева ***T***, поддерживающим оператор ***PARENT*** (Родитель), будет линейный массив ***A***, где каждый элемент ***A[i]*** является указателем или курсором на родителя узла ***i***.
- Корень дерева ***T*** отличается от других узлов тем, что имеет нулевой указатель или указатель на самого себя как на родителя.
- В языке Pascal указатели на элементы массива недопустимы, поэтому мы будем использовать схему с курсорами, тогда ***A [i]=j***, если узел ***j*** является родителем узла ***i***, и ***A[i]***=0, если узел ***i*** является корнем.

Представление деревьев с помощью массивов

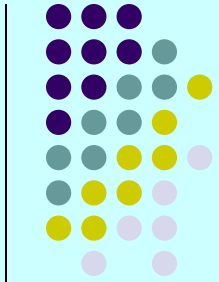
Дерево и курсоры на родителей:



	1	2	3	4	5	6	7	8	9	10
A	0	1	1	2	2	5	5	5	3	3

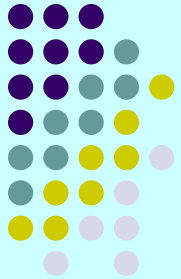
- Данное представление использует то свойство деревьев, что каждый узел, отличный от корня, имеет только одного родителя.
- Используя это представление, родителя любого узла можно найти за фиксированное время. Прохождение по любому пути, т.е. переход по узлам от родителя к родителю, можно выполнить за время, пропорциональное количеству узлов пути.
- Для реализации оператора ***LABEL*** можно использовать другой массив ***L***, в котором элемент ***L[i]*** будет хранить метку узла ***i***, либо объявить элементы массива ***A*** записями,

Представление деревьев с помощью массивов



- Использование указателей или курсоров на родителей не помогает в реализации операторов, требующих информацию о сыновьях.
- Используя описанное представление, крайне тяжело для данного узла ***n*** найти его сыновей или определить его высоту.
- Кроме того, в этом случае невозможно определить порядок сыновей узла (т.е. какой сын находится правее или левее другого сына). Поэтому нельзя реализовать операторы, подобные ***LEFTMOST_CHILD*** и ***RIGHT_SIBLING***.

Представление деревьев с помощью массивов

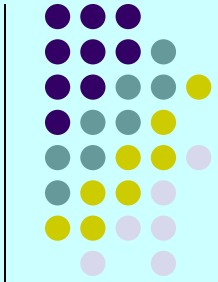


- Можно ввести искусственный порядок нумерации узлов, например нумерацию сыновей в возрастающем порядке слева направо. Используя такую нумерацию, можно реализовать оператор ***RIGHT_SIBLING***, код для этого оператора приведен в листинге 4.4.
- Для задания типов данных ***node*** (узел) и ***TREE*** (Дерево) используется следующее объявление:

  type node = integer;

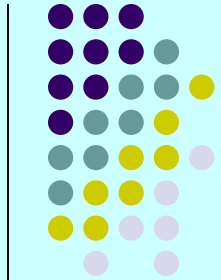
  TREE = array [1..maxnodes] of node;
- В этой реализации мы предполагаем, что нулевой узел **Λ** представлен 0.

Листинг 4.4. Оператор определения правого брата



```
function  RIGHT_SIBLING ( n: node; T: TREE ) : node;
var   i, parent: node;
begin
    parent:= T[n] ;
    RIGHT_SIBLING:=0       { правый брат не найден }
    for   i:=n+1  to   maxnodes  do
    if  T[i]=parent  then
     RIGHT_SIBLING:=i;
end;          { RIGHT_SIBLING }
```

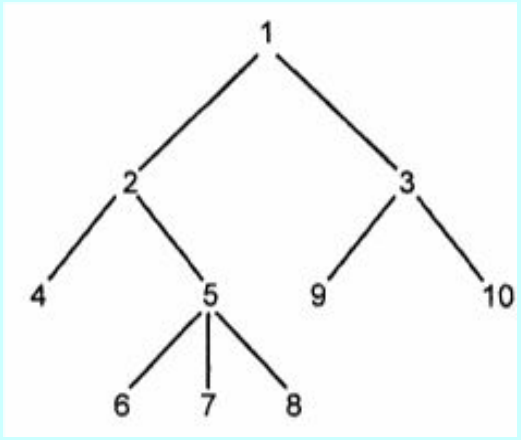
Представление деревьев с помощью списков сыновей

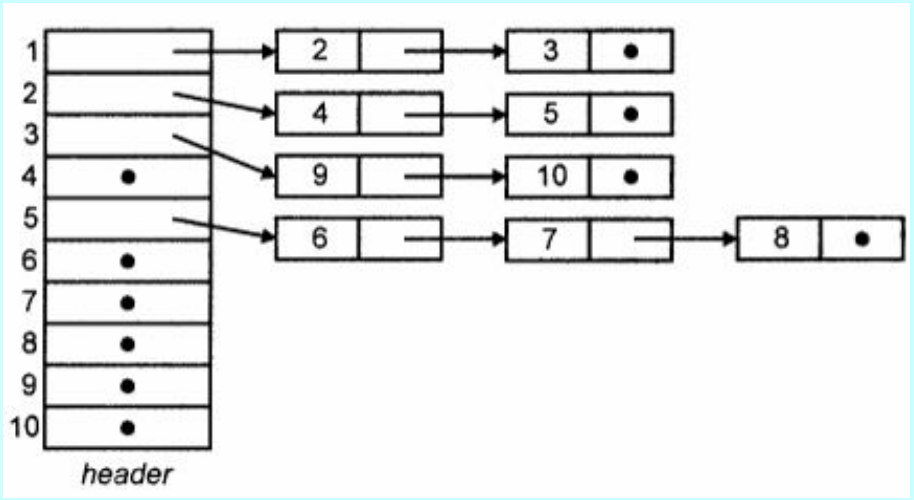


- Важный и полезный способ представления деревьев состоит в формировании для каждого узла списка его сыновей.
- Эти списки можно представить любым методом, описанным в теме 3, но, так как число сыновей у разных узлов может быть разное, чаще всего для этих целей применяются связанные списки.

Представление деревьев с помощью списков сыновей

- На рис. показано, как таким способом представить следующее дерево:





- Здесь есть массив ячеек заголовков, индексированный номерами (они же имена) узлов. Каждый заголовок (***header***) указывает на связанный список, состоящий из "элементов"-узлов. Элементы списка ***header[i]*** являются сыновьями узла ***i***, например узлы 9 и 10 - сыновья узла 3.

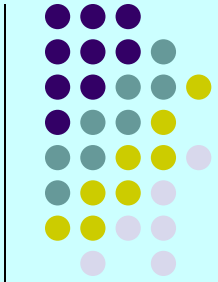
Представление деревьев с помощью списков сыновей

- Прежде чем разрабатывать необходимую структуру данных, нам надо в терминах абстрактного типа данных ***LIST*** (список узлов) сделать отдельную реализацию списков сыновей и посмотреть, как эти абстракции согласуются между собой. Начнем со следующих объявлений типов:

```
type   node = integer;
    LIST= { соотв. определение для списка узлов } ;
    position= { соотв. определение позиций в списках } ;
    TREE = record
            header: array[1..maxnodes]  of  LIST;
            labels: array[1..maxnodes]  of  labeltype;
            root: node
    end;
```

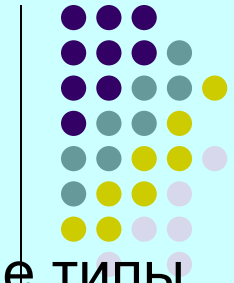
- Мы предполагаем, что корень каждого дерева хранится отдельно в поле ***root*** (корень). Для обозначения нулевого узла используется 0.
- В листинге 4.5 представлен код функции ***LEFTMOST_CHILD***.

Листинг 4.5. Функция нахождения самого левого сына



```
function   LEFTMOST_CHILD ( n: node; T: TREE ): node;
    { возвращает самого левого сына узла n дерева T }
var  L: LIST;    { скоропись для списка сыновей узла n }
begin
    L:= T.header[n];
    if  EMPTY(L)  then      { n является листом }
        LEFTMOST_CHILD:=0
    else
        LEFTMOST_CHILD:= RETRIEVE(FIRST(L), L)
end;         { LEFTMOST_CHILD }
```

Представление деревьев с помощью списков сыновей

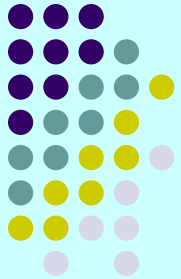


- Теперь представим конкретную реализацию списков, где типы ***LIST*** и ***position*** имеют тип целых чисел, последние используются как курсоры в массиве записей ***cellspace*** (область ячеек):

```
var  cellspace: array[1..maxnodes]  of  record
        node: integer;
        next: integer
end;
```

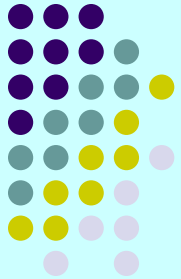
- Для упрощения реализации можно положить, что списки сыновей не имеют ячеек заголовков. Точнее, мы поместим ***T.header[n]*** непосредственно в первую ячейку списка, как показано на рисунке слайда 47.
- В листинге 4.6 представлен переписанный с учетом этого упрощения код функции ***LEFTMOST_CHILD***, а также показан код функции ***PARENT***, использующий данное представление списков. Эта функция более трудна для реализации, так как определение списка, в котором находится заданный узел, требует просмотра всех списков сыновей.

Листинг 4.6. Функции, использующие представление деревьев посредством связанных списков



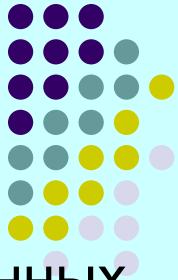
```
function   LEFTMOST_CHILD ( n: node; T: TREE ): node;
{ возвращает самого левого сына узла n дерева T }
var  L: integer;     { курсор на начало списка сыновей узла n }
begin
    L:= T.header[n];
    if   L = 0   then       { n является листом }
        LEFTMOST_CHILD:=0
    else
    LEFTMOST_CHILD:=cellspace[L].node
end;                  { LEFTMOST_CHILD }
```

Листинг 4.6. Функции, использующие представление деревьев посредством связанных списков



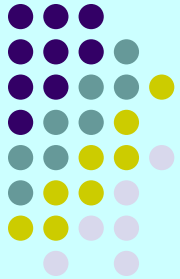
```
function   PARENT ( n: node; T: TREE ): node;
{ возвращает родителя узла n дерева T }
var   p: node;       { пробегает возможных родителей узла n }
      i: position;   { пробегает список сыновей p }
begin
   PARENT:=0      { родитель не найден }
   for  p:=1  to  maxnodes  do  begin
      i:= T.header [p];
      while  i <> 0  do      { проверка на наличие сыновей узла p }
         if  cellspace[i].node = n  then  PARENT:=p
         else  i:= cellspace[i].next
   end;
end; { PARENT }
```

Представление через левых сыновей и правых братьев



- Среди прочих недостатков описанная выше структура данных не позволяет также с помощью операторов ***CREATEi*** создавать большие деревья из малых.
- Это является следствием того, что все деревья совместно используют массив ***cellspace*** для представления связанных списков сыновей; по сути, каждое дерево имеет собственный массив заголовков для своих узлов. А при реализации, например, оператора ***CREATE2(v, T_1, T_2)*** надо скопировать деревья ***T_1, T_2*** в третье дерево и добавить новый узел с меткой ***v*** и двух его сыновей - корни деревьев ***T_1, T_2***.
- Если мы хотим построить большое дерево на основе нескольких малых, то желательно, чтобы все узлы всех деревьев располагались в одной общей области.

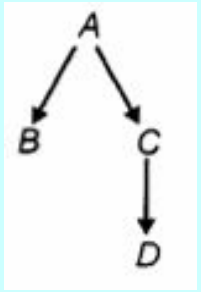
Представление через левых сыновей и правых братьев

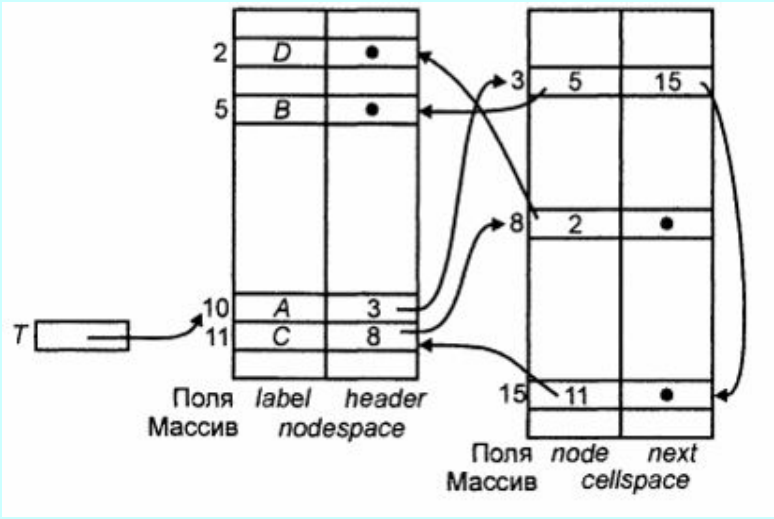


- Логическим продолжением представления дерева, показанного на слайде 47, будет замена массива заголовков на отдельный массив ***nodespace*** (область узлов), содержащий записи с произвольным местоположением в этом массиве.
- Содержимое поля ***header*** этих записей соответствует "номеру" узла, т.е. номеру записи в массиве ***cellspace***.
- В свою очередь поле ***node*** массива ***cellspace*** теперь является курсором для массива ***nodespace***, указывающим позицию узла.
- Тип ***TREE*** в этой ситуации просто курсор в массиве ***nodespace***, указывающий позицию корня.

Пример представления через левых сыновей и правых братьев

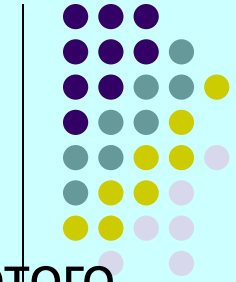
- На рисунке показаны дерево и структура данных, где узлы этого дерева, помеченные как ***A***, ***B***, ***C*** и ***D***, размещены в произвольных позициях массива ***nodespace***. В массиве ***cellspace*** также в произвольном порядке размещены списки сыновей.





- Показанная на рисунке структура данных уже подходит для того, чтобы организовать слияние деревьев с помощью операторов ***CREATEi***.

Представление через левых сыновей и правых братьев

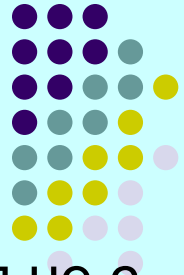


- Но и эту структуру можно значительно упростить. Для этого заметим, что цепочка указателей поля ***next*** массива ***cellspace*** перечисляет всех правых братьев. Используя эти указатели, можно найти самого левого сына следующим образом.

  Предположим, что ***cellspace[i].node = n***. (Т.к. "имя" узла, в отличие от его метки, является индексом в массиве ***nodespace*** и этот индекс записан в поле ***cellspace[i].node***.)

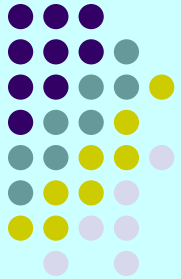
  Тогда указатель ***nodespace[n].header*** указывает на ячейку самого левого сына узла ***n*** в массиве ***cellspace***, поскольку поле ***node*** этой ячейки является именем этого узла в массиве ***nodespace***.

Представление через левых сыновей и правых братьев



- Можно упростить структуру, если идентифицировать узел не с помощью индекса в массиве ***nodespace***, а с помощью индекса ячейки в массиве ***cellspace***, который соответствует данному узлу как сыну.
- Тогда указатель ***next*** (переименуем это поле в ***right_sibling*** - правый брат) массива ***cellspace*** будет точно указывать на правого брата, а информацию, содержащуюся в массиве ***nodespace***, можно перенести в новое поле ***leftmost_child*** (самый левый сын) массива ***cellspace***.
- Здесь тип ***TREE*** является целочисленным типом и используется как курсор в массиве ***cellspace***, указывающий на корень дерева.

Представление через левых сыновей и правых братьев

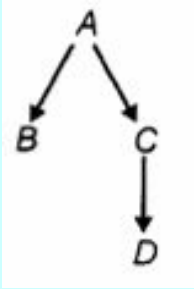


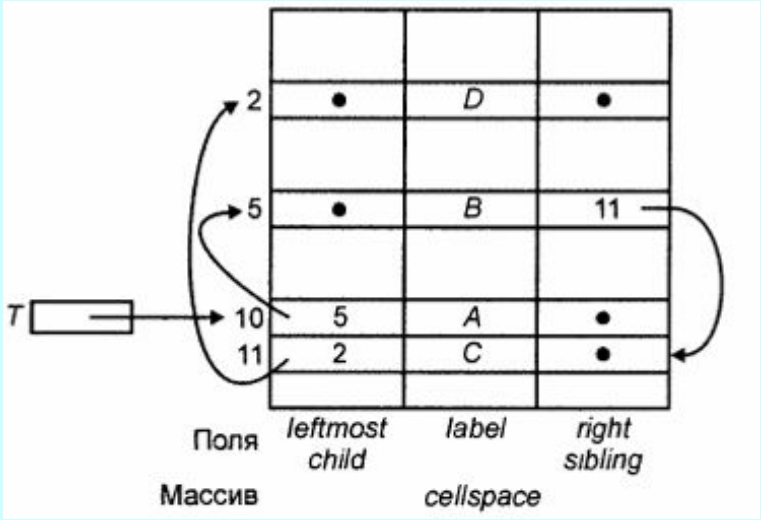
Массив ***cellspace*** можно описать как следующую структуру:

```
var   cellspace: array[1..maxnodes]  of  record
        label: labeltype;
        leftmost_child: integer;
        right_sibling: integer
end;
```

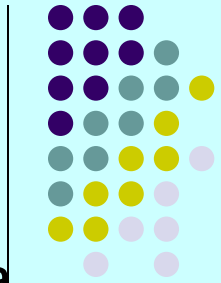
Пример представления через левых сыновей и правых братьев

- Новое представление для дерева, схематически изображено на рис. Узлы дерева расположены в тех же ячейках массива, что и на слайде 55.





Представление через левых сыновей и правых братьев



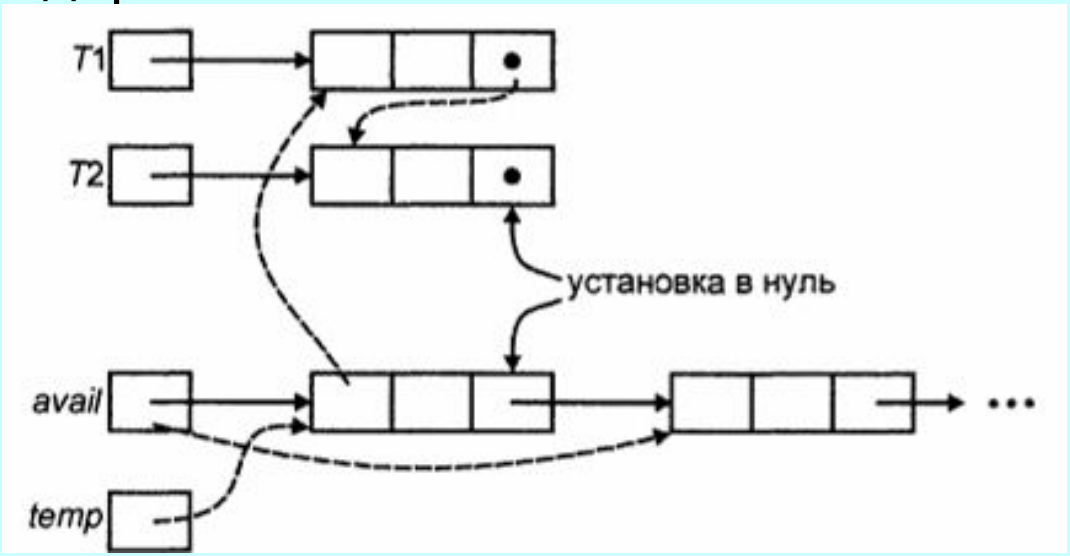
- Используя описанное представление, все операторы, за исключением ***PARENT***, можно реализовать путем прямых вычислений. Оператор ***PARENT*** требует просмотра всего массива ***cellspace***.
- Если необходимо эффективное выполнение оператора ***PARENT***, то можно добавить четвертое поле в массив ***cellspace*** для непосредственного указания на родителей.
- В качестве примера операторов, использующих структуры данных слайда 59, напишем код функции ***CREATE2***, показанный в листинге 4.7.

Листинг 4.7. Функция CREATE2

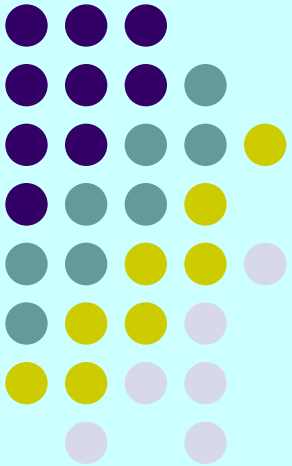
```
function   CREATE2 ( v: labeltype; T1, T2: integer ): integer;
{ возвращает новое дерево с корнем v и поддеревьями T1 и T2 }
var  temp: integer; { хранит индекс первой свободной ячейки
                      для корня нового дерева }
begin
    temp;= avail;
    avail:= cellspace[avail].right_sibling;
    cellspace[temp].leftmost_child:= T1;
    cellspace[temp].label:= v;
    cellspace[temp].right_sibling:= 0;
    cellspace[T1].right_sibling:= T2;
    cellspace[T2].right_sibling:= 0;   {необязательный оператор}
    CREATE2:=temp
end;          { CREATE2 }
```

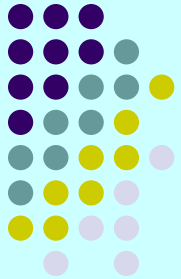
Представление через левых сыновей и правых братьев

- Здесь мы предполагаем, что неиспользуемые ячейки массива ***cellspace*** связаны в один свободный список, который в листинге назван как ***avail***, и ячейки этого списка связаны посредством поля ***right_sibling***.
- На рис. показаны изменения указателей при выполнении функции CREATE2. Сплошные линии отображают старые указатели, а пунктирные линии - новые указатели в процессе создания нового дерева.



Двоичные деревья





- Деревья, которые мы определили выше, называются ***упорядоченными ориентированными деревьями***, поскольку сыновья любого узла упорядочены слева направо, а пути по дереву ориентированы от начального узла пути к его потомкам.
- Двоичное (или бинарное) дерево - совершенно другой вид.
- Двоичное дерево может быть или пустым деревом, или деревом, у которого любой узел или не имеет сыновей, или имеет либо левого сына, либо правого сына, либо обоих.
- Тот факт, что каждый сын любого узла определен как левый или как правый сын, существенно отличает двоичное дерево от упорядоченного ориентированного дерева.

Пример 6.

- Если принять соглашение, что на схемах двоичных деревьев левый сын всегда соединяется с родителем линией, направленной влево и вниз от родителя, а правый сын - линией, направленной вправо и вниз, тогда на рис. 1, а, б представлены два различных дерева, хотя они оба похожи на обычное (упорядоченное ориентированное) дерево, показанное на рис. 2.
- Деревья на рис. 1,а, б различны и не эквивалентны дереву на рис. 2.

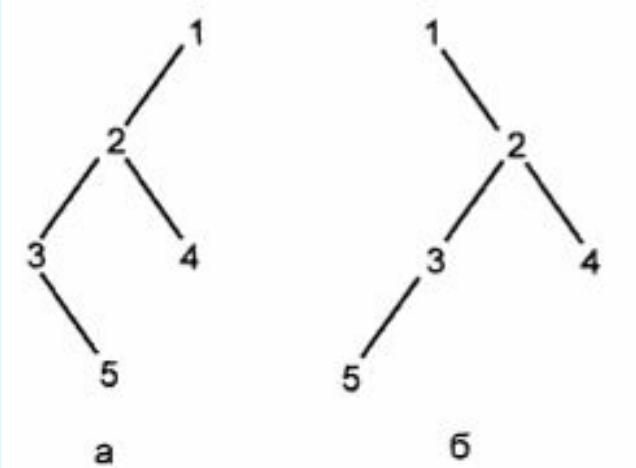


Рис. 1

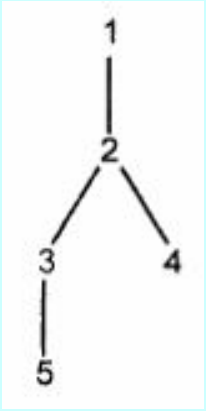
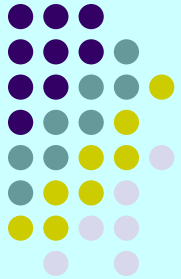
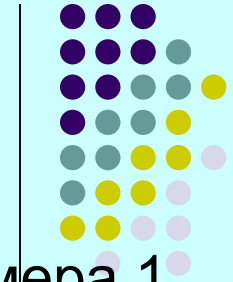


Рис. 2



- ***Обход двоичных деревьев*** в прямом и обратном порядке в точности соответствует таким же обходам обычных деревьев.
- При ***симметричном обходе двоичного дерева*** с корнем ***n*** левым поддеревом T_1 и правым поддеревом T_2 сначала проходится поддерево T_1 затем корень ***n*** и далее поддерево T_2. Например, симметричный обход дерева
- на рис. 1,а даст последовательность узлов 3, 5, 2, 4, 1.

Представление двоичных деревьев



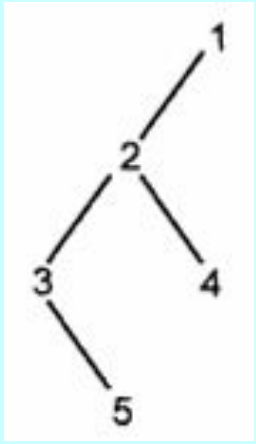
- Если именами узлов двоичного дерева являются их номера 1, 2, ..., ***n***, то подходящей структурой для представления этого дерева может служить массив ***cellspace*** записей с полями ***leftchild*** (левый сын) и ***rightchild*** (правый сын), объявленный следующим образом:

```
var
cellspace: array[1..maxnodes] of record
leftchild: integer;
rightchild: integer
end;
```

- В этом представлении ***cellspace[i].leftchild*** является левым сыном узла ***i***, а ***cellspace[i].rightchild*** - правым сыном. Значение 0 в обоих полях указывает на то, что узел ***i*** не имеет сыновей.

Представление двоичных деревьев

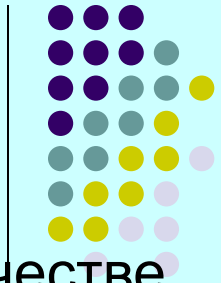
Двоичное дерево:



Представление двоичного дерева:

	Значение поля *leftchild*	Значение поля *rightchild*
1	2	0
2	3	4
3	0	5
4	0	0
5	0	0

Коды Хаффмана



- Приведем пример применения двоичных деревьев в качестве структур данных. Для этого рассмотрим задачу конструирования ***кодов Хаффмана***.
- Предположим, мы имеем сообщения, состоящие из последовательности символов. В каждом сообщении символы независимы и появляются с известной вероятностью, не зависящей от позиции в сообщении.

Например, мы имеем сообщения, состоящие из пяти символов

a, ***b***, ***c***, ***d***, ***e***,

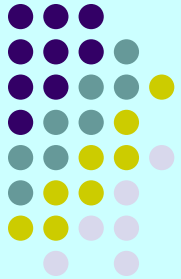
которые появляются в сообщениях с вероятностями

0.12, 0.4, 0.15, 0.08 и 0.25

соответственно.

- Мы хотим закодировать каждый символ последовательностью из нулей и единиц так, чтобы код любого символа являлся префиксом кода сообщения, состоящего из последующих символов. Это префиксное свойство позволяет декодировать строку из нулей и единиц последовательным удалением префиксов (т.е. кодов символов) из этой строки.

Коды Хаффмана



- В таблице показаны две возможные кодировки для наших пяти символов.

Символ	Вероятность	Код 1	Код 2
a	0.12	000	000
b	0.40	001	11
c	0.15	010	01
d	0.08	011	001
e	0.25	100	10

- Ясно, что первый код обладает префиксным свойством, поскольку любая последовательность из трех битов будет префиксом для другой последовательности из трех битов; другими словами, любая префиксная последовательность однозначно идентифицируется символом.
- Алгоритм декодирования для этого кода очень прост:
  - надо поочередно брать по три бита и преобразовать каждую группу битов в соответствующие символы.
- Например, последовательность 001010011 соответствует исходному сообщению ***bed***.

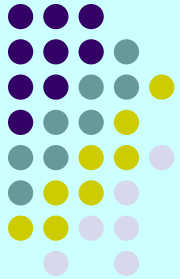
Коды Хаффмана

- ***Задача конструирования кодов Хаффмана*** заключается в следующем:

  имея множество символов и значения вероятностей их появления в сообщениях, построить такой код с префиксным свойством, чтобы средняя длина кода (в вероятностном смысле) последовательности символов была минимальной.

- Мы хотим минимизировать среднюю длину кода для того, чтобы уменьшить длину вероятного сообщения (т.е. чтобы сжать сообщение). Чем короче среднее значение длины кода символов, тем короче закодированное сообщение.

Коды Хаффмана

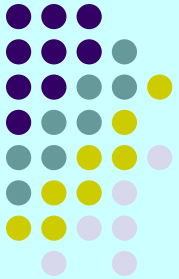


- В частности, первый код из примера слайда 71 имеет среднюю длину кода 3. Это число получается в результате умножения длины кода каждого символа на вероятность появления этого символа.
- Второй код имеет среднюю длину 2.2, поскольку символы ***a*** и ***d*** имеют суммарную вероятность появления 0.20 и длина их кода составляет три бита, тогда как другие символы имеют код длиной 21.

- Отсюда следует очевидный вывод, что

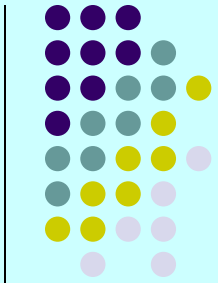
  символы с большими вероятностями появления должны иметь самые короткие коды.

Коды Хаффмана



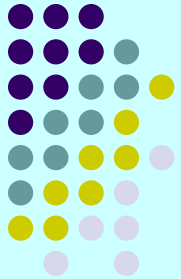
- Можно ли придумать код, который был бы лучше второго кода?
- Ответ положительный: существует код с префиксным свойством, средняя длина которого равна 2.15.
- Это наилучший возможный код с теми же вероятностями появления символов.

Коды Хаффмана



- Способ нахождения оптимального префиксного кода называется ***алгоритмом Хаффмана***:
  - Находятся два символа ***a*** и ***b*** с наименьшими вероятностями появления и заменяются одним фиктивным символом, например ***x***, который имеет вероятность появления, равную сумме вероятностей появления символов ***a*** и ***b***.
  - Используя эту процедуру рекурсивно, находим оптимальный префиксный код для меньшего множества символов (где символы ***a*** и ***b*** заменены одним символом ***x***). Код для исходного множества символов получается из кодов замещающих символов путем добавления 0 и 1 перед кодом замещающего символа, и эти два новых кода принимаются как коды заменяемых символов.
  - Например, код символа ***a*** будет соответствовать коду символа ***x*** с добавленным нулем перед этим кодом, а для кода символа ***b*** перед кодом символа ***x*** будет добавлена единица.

Коды Хаффмана



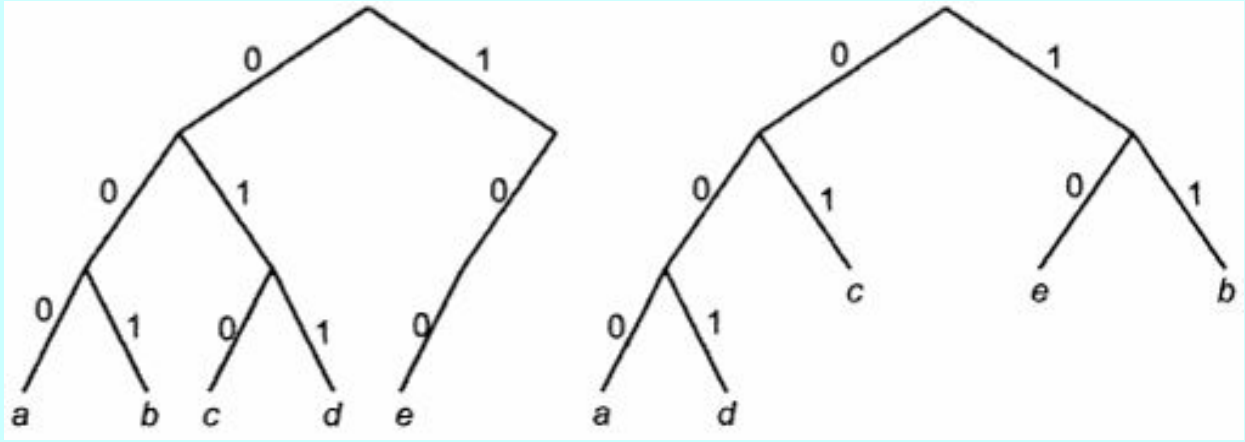
- Можно рассматривать префиксные коды как пути на двоичном дереве:

  прохождение от узла к его левому сыну соответствует 0 в коде, а к правому сыну - 1.
- Если мы пометим листья дерева кодируемыми символами, то получим представление префиксного кода в виде двоичного дерева.
- Префиксное свойство гарантирует, что нет символов, которые были бы метками внутренних узлов дерева (не листьев).
- И наоборот, помечая кодируемыми символами только листья дерева, мы обеспечиваем префиксное свойство кода этих символов.

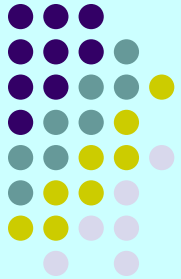
Коды Хаффмана

Символ	Вероятность	Код 1	Код 2
a	0.12	000	000
b	0.40	001	11
c	0.15	010	01
d	0.08	011	001
e	0.25	100	10

- Двоичные деревья для кодов 1 и 2:

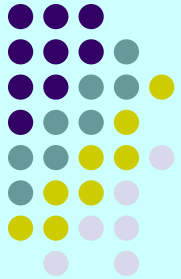


Коды Хаффмана



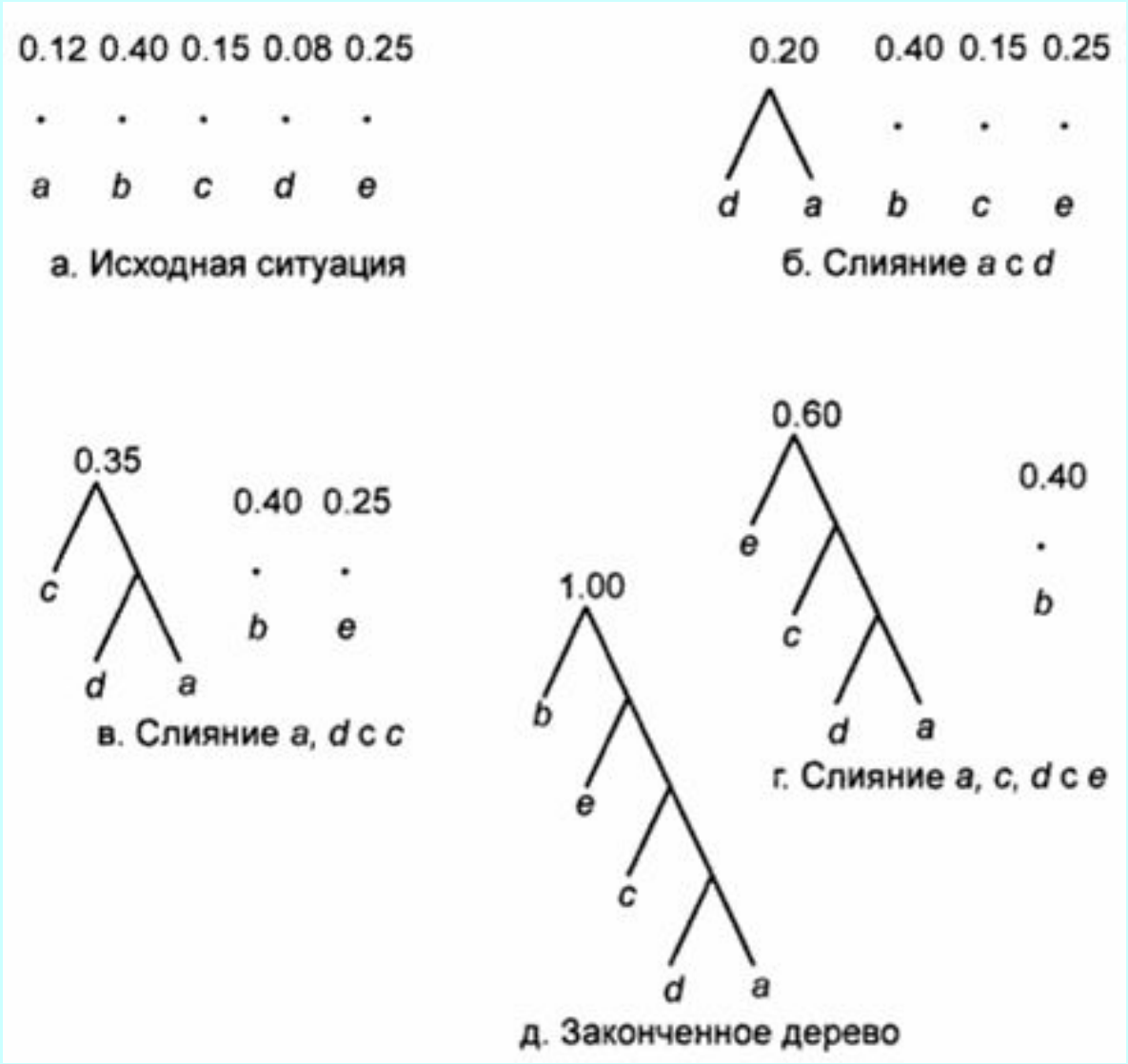
- Для реализации алгоритма Хаффмана мы используем ***лес***, т. е. совокупность деревьев, чьи листья будут помечены символами, для которых разрабатывается кодировка, а корни помечены суммой вероятностей всех символов, соответствующих листьям дерева. Мы будем называть эти суммарные вероятности ***весом дерева***.
- Вначале каждому символу соответствует дерево, состоящее из одного узла, в конце работы алгоритма мы получим одно дерево, все листья которого будут помечены кодируемыми символами.
- В этом дереве путь от корня к любому листу представляет код для символа-метки этого листа, составленный по схеме, согласно которой левый сын узла соответствует 0, а правый - 1 (как на рис. слайда 76).

Коды Хаффмана

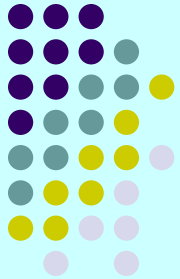


- Важным этапом в работе алгоритма является выбор из леса двух деревьев с наименьшими весами.
- Эти два дерева комбинируются в одно с весом, равным сумме весов составляющих деревьев. При слиянии деревьев создается новый узел, который становится корнем объединенного дерева и который имеет в качестве левого и правого сыновей корни старых деревьев.
- Этот процесс продолжается до тех пор, пока не получится только одно дерево. Это дерево соответствует коду, который при заданных вероятностях имеет минимально возможную среднюю длину.

Этапы создания дерева Хаффмана



Коды Хаффмана

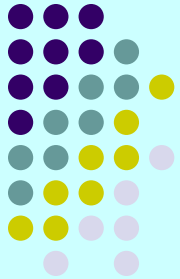


- Теперь опишем необходимые структуры данных.
- Во-первых, для представления двоичных деревьев мы будем использовать массив ***TREE*** (Дерево), состоящий из записей следующего типа:

```
record
    leftchild: integer;
    rightchild: integer;
    parent: integer
end
```

Указатели в поле ***parent*** (родитель) облегчают поиск путей от листа к корню при записи кода символов.

Коды Хаффмана

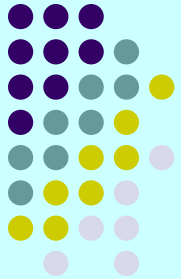


- Во-вторых, мы используем массив ***ALPHABET*** (Алфавит), также состоящий из записей, которые имеют следующий тип:

```
record
    symbol: char;
    probability: real;
    leaf: integer  { курсор }
end
```

В этом массиве каждому символу (поле ***symbol***), подлежащему кодированию, ставится, в соответствие вероятность его появления (поле ***probability***) и лист, меткой которого он является (поле ***leaf***).

Коды Хаффмана



- В-третьих, для представления непосредственно деревьев необходим массив ***FOREST*** (Лес). Этот массив будет состоять из записей с полями

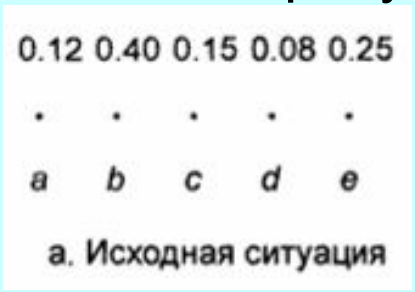
weight (вес) и ***root*** (корень)

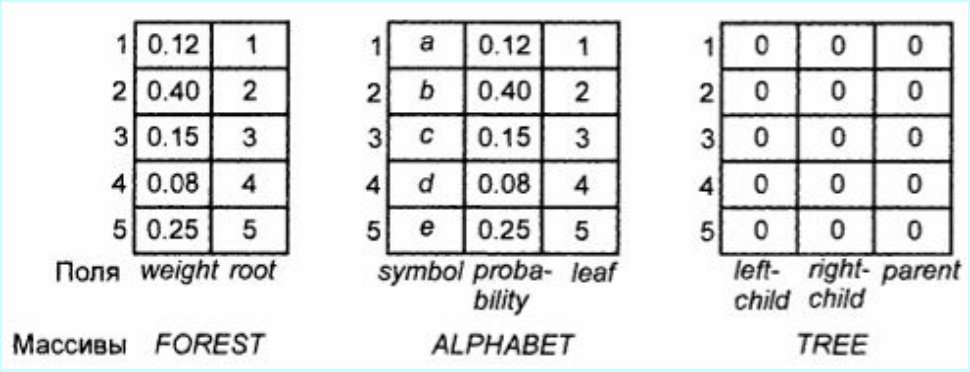
следующего типа:

```
record
   weight: real;
   root: integer
end
```

Коды Хаффмана

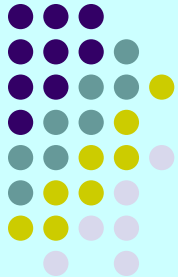
- Начальные значения всех трех массивов, соответствующих данным на рисунке *а* показаны ниже.





- Эскиз программы построения дерева Хаффмана на псевдокоде представлен в листинге 4.8.

Листинг 4.8. Программа построения дерева Хаффмана



(1) ***while*** **существует более одного дерева в лесу** ***do begin***

(2) ***i:=*** **индекс дерева в** ***FOREST*** **с наименьшим весом*;***

(3) ***j:=*** **индекс дерева в** ***FOREST*** **со вторым наименьшим весом*;***

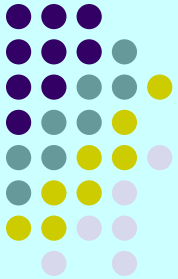
(4) **Создание нового узла с левым сыном** ***FOREST[i].root*** **и правым сыном** ***FOREST[j].root;***

(5) **Замена в** ***FOREST*** **дерева** ***i*** **деревом, чьим корнем является новый узел и чей вес равен** ***FOREST[i].weight + FOREST[j].weight;***

(6) **Удаление дерева** ***j*** **из массива** ***FOREST***

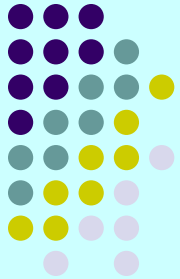
end;

Листинг 4.8. Программа построения дерева Хаффмана



- Для реализации строки (4) листинга 4.8, где увеличивается количество используемых ячеек массива ***TREE***, и строк (5) и (6), где уменьшается количество ячеек массива ***FOREST***, мы будем использовать курсоры ***lasttree*** (последнее дерево) и ***lastnode*** (последний узел), указывающие соответственно на массив ***FOREST*** и массив ***TREE***.
- Предполагается, что эти курсоры располагаются в первых ячейках соответствующих массивов.
- Для этапа чтения данных, который мы опускаем, необходим также курсор для массива ***ALPHABET***, который бы "следил" за заполнением данными этого массива.
- Мы также предполагаем, что все массивы имеют определенную объявленную длину, но здесь мы не будем проводить сравнение этих ограничивающих значений со значениями курсоров.

Коды Хаффмана



- В листинге 4.9 приведены коды двух полезных процедур.
- Процедура ***lightones***, выполняет реализацию строк (2) и (3) листинга 4.8 по выбору индексов двух деревьев с наименьшими весами.
- Функция ***create(n_1, n_2)***, создает новый узел и делает заданные узлы n_1 и n_2 левым и правым сыновьями этого узла.

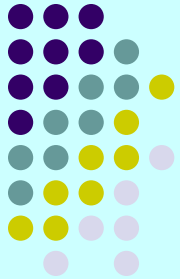
Листинг 4.9.

```
procedure  lightones ( var  least, second: integer );
    { присваивает переменным least и second индексы массива
     FOREST, соответствующие деревьям с наименьшими весами.
    Предполагается, что lasttree > 2. }
var   i: integer;
begin { инициализация least и second, рассм-ся первые два дерева }
   if   FOREST[1].weight <= F0REST[2].weight   then  begin
          least:= 1;       second:= 2     end
  else   begin
          least:= 2;  second:= 1     end
          for   i:= 3  to   lasttree   do
           if   FOREST[i].weight < FOREST[least].weight  then
              begin
                   second:= least;   least:= i     end
          else
               if  FOREST[i].weight < FOREST[second].weight then
                    second:= i
end;          { lightones }
```

Листинг 4.9.

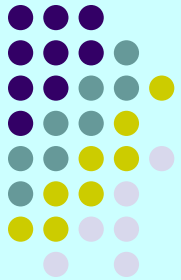
```
function   create ( lefttree, righttree: integer ): integer;
    { возвращает новый узел, у которого левым и правым сыновьями
    становятся FOREST[lefttree].root и FOREST[righttree].root }
begin
   lastnode:= lastnode + 1;
   { ячейка TREE[lastnode] для нового узла }
   TREE[lastnode].leftchild:= FOREST[lefttree].root;
   TREE[lastnode].rightchild:= FOREST[righttree].root;
   { теперь введем указатели для нового узла и его сыновей }
   TREE[lastnode].parent:= 0;
   TREE[FOREST[lefttree].root].parent:= lastnode;
   TREE[FOREST[righttree].root].parent:= lastnode;
   create:=lastnode
end;              { create }
```

Коды Хаффмана



- Теперь все неформальные операторы листинга 4.8 можно описать подробнее.
- В листинге 4.10 приведен код процедуры ***Huffman***, которая не осуществляет ввод и вывод, а работает со структурами, показанными на рис. слайда 83, которые объявлены как глобальные.

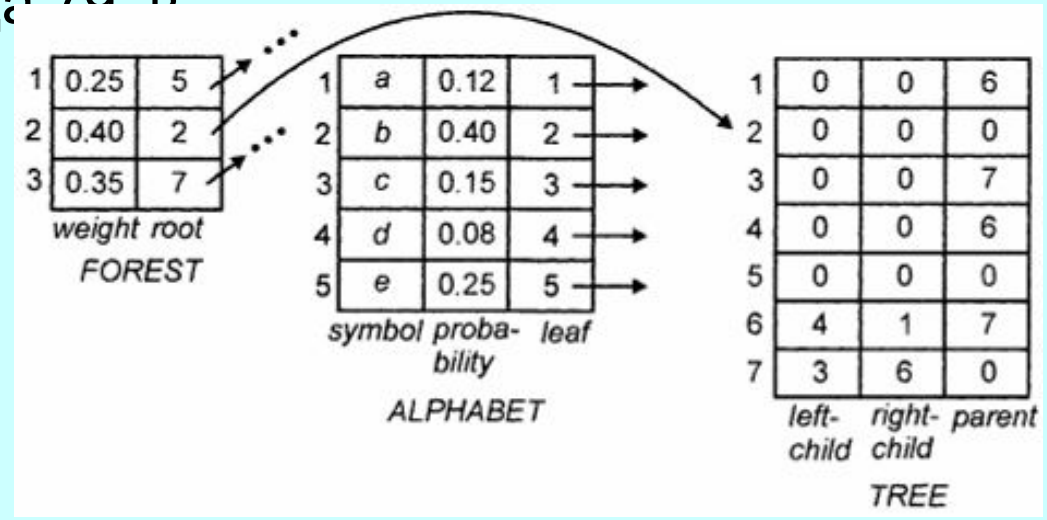
Листинг 4.10. Реализация алгоритма Хаффмана



```
procedure Huffman;
var  i, j:integer; {два дерева с наименьшими весами из FOREST }
    newroot: integer;
begin
    while  lasttree > 1  do  begin
        lightones(i, j);
        newroot:= create(i, j);
        { далее дерево i заменяется деревом с корнем newroot }
        FOREST[i].weight:=FOREST[i].weight +FOREST[j].weight;
        FOREST[i].root:= newroot;
        { далее дерево j заменяется на дерево lasttree,
        массив FOREST уменьшается на одну запись }
        FOREST[j]:= FOREST[lasttree] ;
        lasttree:= lasttree - 1
    end
end;                { Huffman }
```

Коды Хаффмана

- На рисунке показана структура данных из рис. слайда 83 после двух итераций, т.е. после того, как значение переменной ***lasttree*** уменьшено до 3, т.е. лес имеет вид, показанный на рис. слайда 79, в

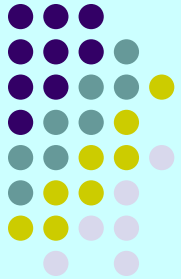


Коды Хаффмана

После завершения работы алгоритма код каждого символа можно определить следующим образом.

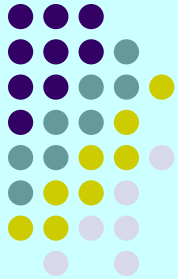
- Найдем в массиве ***ALPHABET*** запись с нужным символ в поле ***symbol***.
- Затем по значению поля ***leaf*** этой же записи определим местоположение записи в массиве ***TREE***, которая соответствует листу, помеченному рассматриваемым символом.
- Далее последовательно переходим по указателю ***parent*** от текущей записи, например соответствующей узлу ***n***, к записи в массиве ***TREE***, соответствующей его родителю ***p***.
- По родителю ***p*** определяем, в каком его поле, ***leftchild*** или ***rightchild***, находится указатель на узел ***n***, т.е. является ли узел ***n*** левым или правым сыном, и в соответствии с этим печатаем 0 (для левого сына) или 1 (для правого сына).
- Затем переходим к родителю узла ***p*** и определяем, является ли его сын ***p*** правым или левым, и в соответствии с эти печатаем следующую 1 или 0, и т. д. до самого корня дерева.

Коды Хаффмана



Таким образом, код символа будет напечатан в виде последовательности битов, но в обратном порядке. Чтобы распечатать эту последовательность в прямом порядке, надо каждый очередной бит помещать в стек, а затем распечатать содержимое стека в обычном порядке.

Реализация двоичных деревьев с помощью указателей

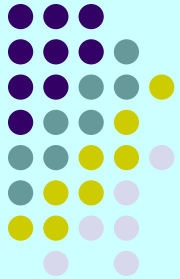


Для указания на правых и левых сыновей (и родителей, если необходимо) вместо курсоров можно использовать настоящие указатели языка Pascal. Например, можно сделать объявление

```
type   node = record
   leftchild: ^ node;
   rightchild: ^ node;
   parent: ^ node
end
```

Используя этот тип данных узлов двоичного дерева, функцию ***create*** (листинг 4.9) можно переписать так, как показано в следующем листинге 4.11.

Листинг 4.11. Функция create при реализации двоичного дерева с помощью указателей



```
function  create ( lefttree, righttree: ^node): ^node;
var   root: ^node;
begin
    new(root) ;
    root^.leftchild:= lefttree;
    root^.rightchild:= righttree;
    root^.parent:= 0;
    lefttree^.parent:= root;
    righttree^.parent:= root;
    create:=root
end;                { create }
```