Лекция 22 Топологическая сортировка.

22.1. ***Представление произвольного дерева в виде двоичного.***

22.1.1. В отличие от двоичного дерева произвольное дерево не может быть пустым (по определению оно должно содержать хотя бы один узел – корень). Каждый узел произвольного дерева может иметь 0, 1, 2, 3 и более детей.

Поддеревья любого узла произвольного дерева образуют *лес* (лесом называется упорядоченный набор 0, 1, 2, и более деревьев).

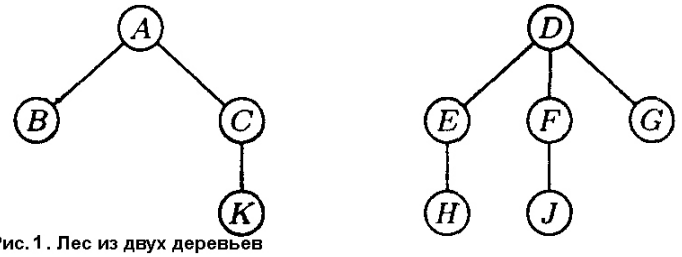

Рис. 1. Лес из двух деревьев

22.1.2. Существует естественный способ представления любого леса в виде двоичного дерева. Пример лес из двух деревьев, представленный на рисунке 1 можно представить в виде двоичного дерева (рисунок 2).

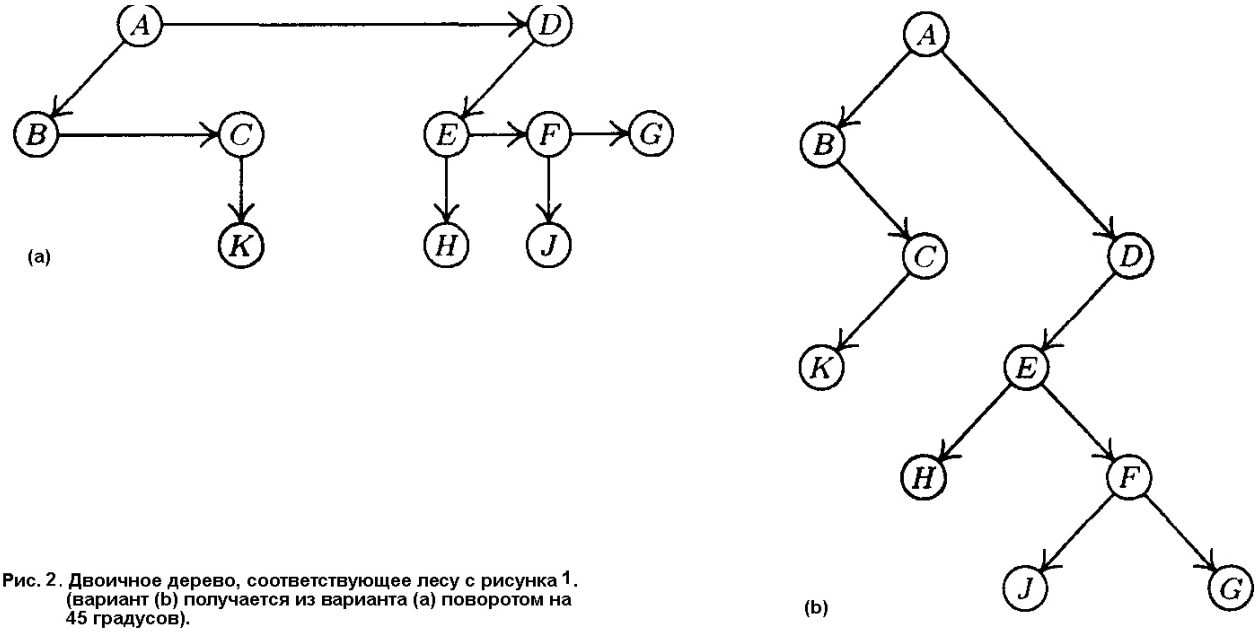

Рис. 2. Двоичное дерево, соответствующее лесу с рисунка 1. (вариант (b) получается из варианта (a) поворотом на 45 градусов).

Такое приведение леса к двоичному дереву называется *естественным соответствием* между лесом и двоичными деревьями.

22.1.3. ***Определение***. Пусть $F = (T_1, T_2, \ldots, T_n)$ – некоторый лес деревьев. Тогда двоичное дерево $B(F)$, соответствующее F, можно определить следующим образом:
(1) Если $n = 0$, то $B(F)$ – пустое дерево.
(2) Если $n > 0$, то корнем $B(F) = B(T_1, T_2, \ldots, T_n)$ является корень дерева T_1, левым поддеревом $B(F)$ является $B(T_{11}, T_{12}, \ldots, T_{1m})$, где $T_{11}, T_{12}, \ldots, T_{1m}$ – поддеревья корня T_1 (лес), правым поддеревом $B(F)$ является $B(T_2, \ldots, T_n)$.

22.1.4. Для $B(F)$ можно построить прошитое дерево ($B(F)$ можно «прошить»). Прошитое дерево, соответствующее двоичному дереву с рисунка 6(а), представлено на рисунке 3. Отметим, что *правые нити всегда проходят от младшего* (*крайнего справа*) *ребенка к его родителю*. Левые нити не имеют такой естественной интерпретации из-за отсутствия симметрии между левой и правой сторонами дерева.

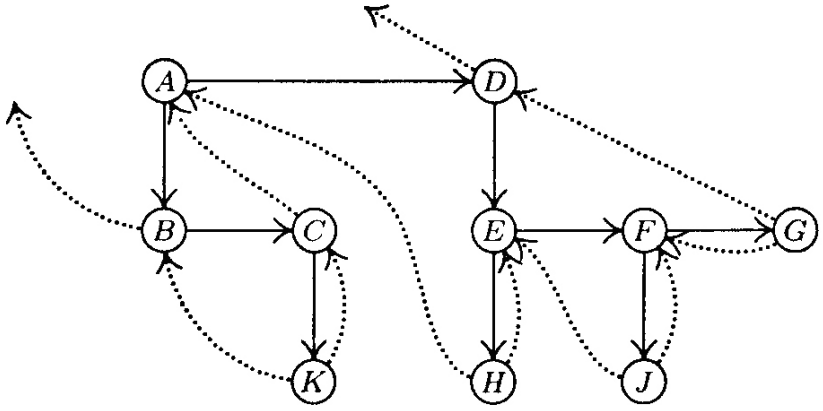

Рис. 3. Прошитое двоичное дерево, соответствующее двоичному дереву с рисунка 2(а)

22.1.5. Идеи обхода двоичных деревьев можно применить к произвольным деревьям (и лесам). К сожалению, отсутствие симметрии при отображении леса на двоичное дерево не позволяет легко сформулировать симметричный обход. Но прямой (preorder) и обратный (postorder) обходы можно перенести без всяких затруднений.

22.1.5.1. Прямой обход леса (произвольного дерева).
(1) Обработать корень первого дерева (T_1).
(2) Обойти поддеревья первого дерева ($T_{11}, T_{12}, \ldots, T_{1m}$).
(3) Обойти остальные поддеревья ($T_2, \ldots, T_n$).

22.1.5.2. Обратный обход леса (произвольного дерева).
(1) Обойти поддеревья первого дерева ($T_{11}, T_{12}, \ldots, T_{1m}$).
(2) Обработать корень первого дерева (T_1).
(3) Обойти остальные поддеревья ($T_2, \ldots, T_n$).

22.1.5.3. ***Пример***.
Лес с рис. 5 можно представить с помощью следующей скобочной записи: ($A(B, C(K)), D(E(H), F(J), G)$). При прямом обходе узлы двух деревьев обходятся в порядке *A B C K D E H F J G*. Последняя последовательность идентична скобочной записи, если из нее удалить скобки и узлы.

Прямой порядок обхода называется также *династическим*, так как с помощью такого обхода генеалогического дерева династии определяется наследник престола.

При обратном порядке обхода каждый узел располагается после своих наследников, что приводит к следующей скобочной структуре: ((*B*, *K*(*C*))*A*, (*H*(*E*), *J*(*F*), *G*)*D*). Порядок обхода в этом случае будет: *B K C A H E J F G D*.

Таким образом, обход леса в прямом порядке выполняется точно так, как обход соответствующего двоичного дерева в прямом порядке. Обход леса в обратном порядке выполняется точно так, как *симметричный* обход соответствующего двоичного дерева (***обратите внимание!***).

Следовательно, можно без изменений использовать алгоритмы (функции) **`Inorder()`** и **`Next_in()`**.

22.1.5.4. ***Пример***.

Правую часть формулы $y = 3 \cdot \ln(x + 1) - a / x^2$ можно представить в виде дерева (рисунок 4). Отметим, что хотя это дерево очень похоже на двоичное, здесь оно рассматривается как *обычное дерево*. Для этого дерева можно построить прошитое двоичное дерево (оно показано на рисунке 5).

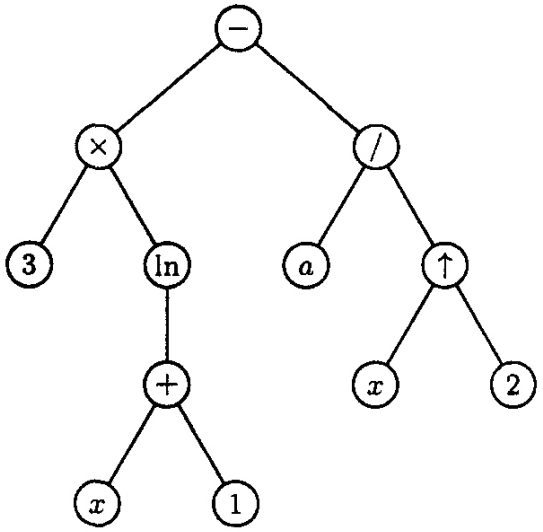

Рис. 4. Представление формулы $y = 3ln(x+1) - a/x^2$ **в виде дерева**

Это дерево содержит дополнительный узел *y*, являющийся заголовком прошитого двоичного дерева.

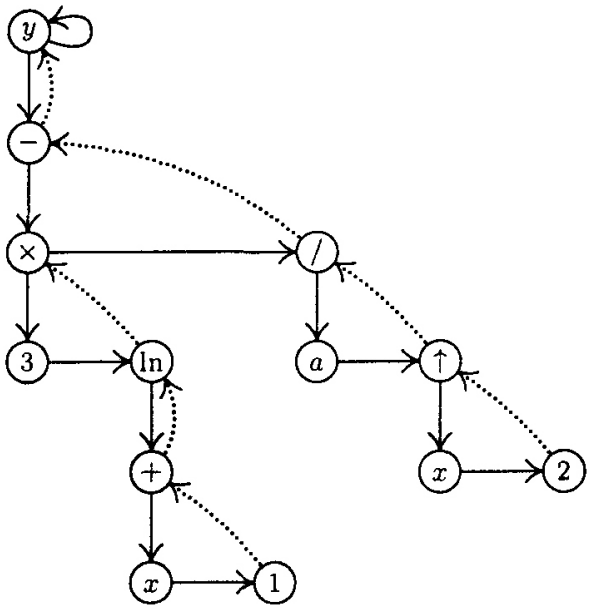

Рис. 5. Прошитое двоичное дерево, соответствующее дереву с рисунка 4.

При обходе узлов двоичного дерева, показанного на рисунке 9, получим:

для прямого обхода $- \times 3 \ln + x\ 1 / a \uparrow x\ 2$ (1)

для обратного обхода $3\ x\ 1 + \ln \times a\ x\ 2 \uparrow / -$ (2)

Полученные алгебраические выражения называются *бесскобочной польской записью* алгебраических выражений; при этом (1) называется префиксной (или прямой) польской записью, а (2) – постфиксной (или обратной) польской записью. Название связано с тем, что такая запись была впервые использована польским математиком Яном Лукасевичем.

Представление выражений в виде польской записи или в виде дерева, представленного на рисунке 9, используются в компиляторах, а также в системах символьного преобразования выражений (например, в системах символьного дифференцирования, символьного интегрирования, символьного преобразования алгебраических выражений и т.п.).

22.2. ***Топологическая сортировка узлов ациклического ориентированного графа* (*элементов частично-упорядоченного множества*).**

22.2.1. Топологическая сортировка применима и к произвольным ациклическим ориентированным графам (двоичное дерево – частный случай такого графа). Ациклический граф можно использовать для графического изображения *частично упорядоченного множества* (когда в множестве есть отношение порядка, но не для всех элементов: некоторые элементы несравнимы). Цель топологической сортировки преобразовать частичный порядок в линейный. Графически это означает, что все узлы графа нужно расположить на одной прямой таким образом, чтобы все дуги графа, соединяющие его узлы, были направлены в одну сторону.

22.2.2. ***Пример***.

Пусть частичный порядок задается следующим набором отношений между ключами:

$$a < b,\ b < d,\ d < f,\ b < l,\ d < h,\ f < c,\ a < c, \qquad (*)$$
$$c < e,\ e < h,\ g < e,\ g < k,\ k < d,\ k < l$$

Этот набор отношений можно представить в виде ациклического графа (рисунок 6), причем каждое отношение представляет дугу указанного графа

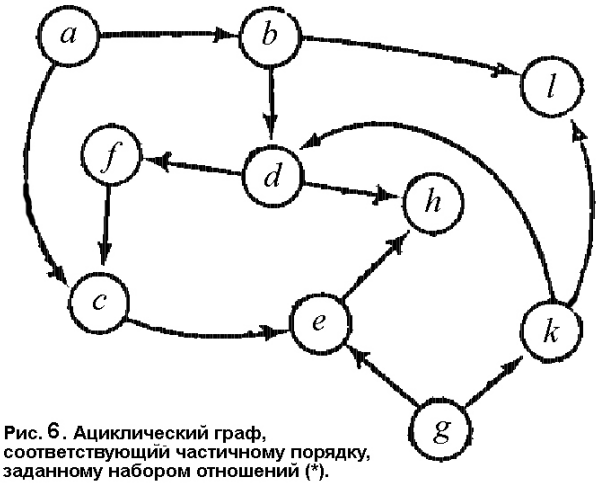

Рис. 6. Ациклический граф, соответствующий частичному порядку, заданному набором отношений (*).

Задача состоит в том, чтобы привести граф с рисунка 6 к линейному графу, показанному на рисунке 7. Ключи после топологической сортировки будут расположены, например, в следующем порядке: ***g k a b d f c e h l*** (понятно, что топологическая сортировка неоднозначна, так что это один из возможных топологических порядков). Последовательная обработка полученного линейного списка узлов графа эквивалентна их обработке в порядке обхода графа.

22.2.3. ***Описание алгоритма. Структуры данных***. Находим узел *x* графа, у которого нет предшествующих узлов (по крайней мере один такой элемент должен существовать, так как иначе у графа были бы циклы). Этот узел, будем называть его «ведущим»

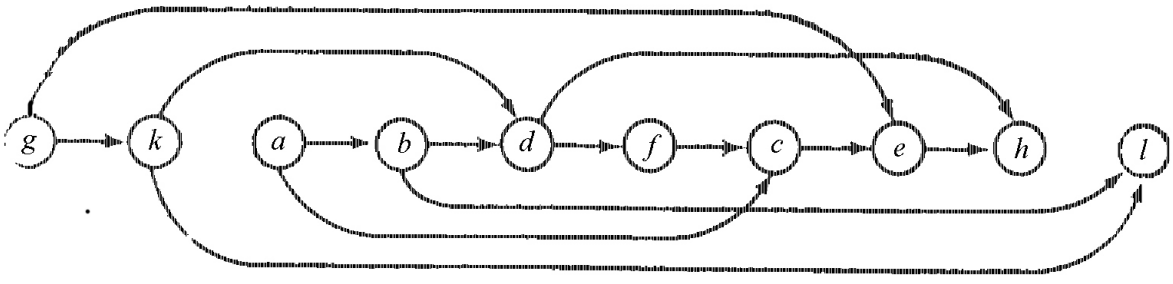

Рис. 7. Линейное расположение узлов графа с рисунка 6

узлом, поместим в начало линейного списка узлов и удалим из графа. Поскольку оставшийся граф останется ациклическим, можно применять описанное преобразование до тех пор, пока все узлы графа не станут «ведущими» и перейдут в линейный список.

Для более строгого описания алгоритма необходимо выбрать структуру представления узлов графа. Так как основной операцией является нахождение «ведущего» узла *a*, дескриптор каждого «ведущего» узла должен, помимо ключа *a*, содержать следующие характеристики: количество предшественников *count* (у «ведущего» узла значение *count* равно 0, по мере удаления узлов значение *count* уменьшается), указатель списка «ведомых» узлов; кроме перечисленных характеристик *дескриптор* узла (рисунок 8)

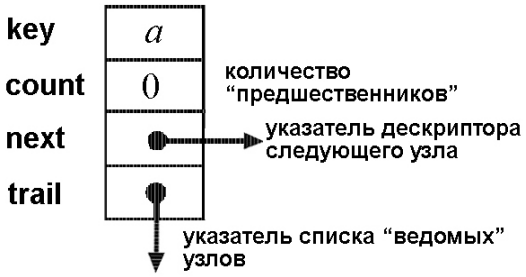

Рис. 8. Дескриптор узла.

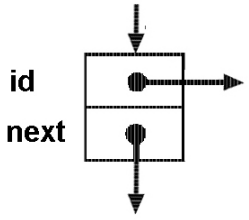

Рис. 9. Дескриптор "ведомого" узла

должен содержать указатель следующего элемента линейного списка «ведущих» узлов. Каждый «ведомый» узел имеет еще один дескриптор (эти дескрипторы помещаются в списки «ведомых» узлов), который содержит указатель списка дескрипторов «ведомых» узлов и указатель на дескриптор соответствующего узла в списке «ведущих» узлов (рисунок 9).

22.2.4. ***Описание алгоритма. Фаза ввода***.

Пусть исходные данные представлены в виде множества пар ключей (*). Пары ключей вводятся в произвольном порядке. После ввода очередной пары $x < y$ ключи ищутся в списке «ведущих» (с помощью функции **`find`**) и в случае отсутствия добавляются к нему. Затем в список «ведомых» для x добавляется дескриптор y и счетчик предшественников y увеличивается на 1 (сначала значения всех счетчиков равны 0).
По окончании фазы ввода будет сформирована структура, показанная на рисунке 10.

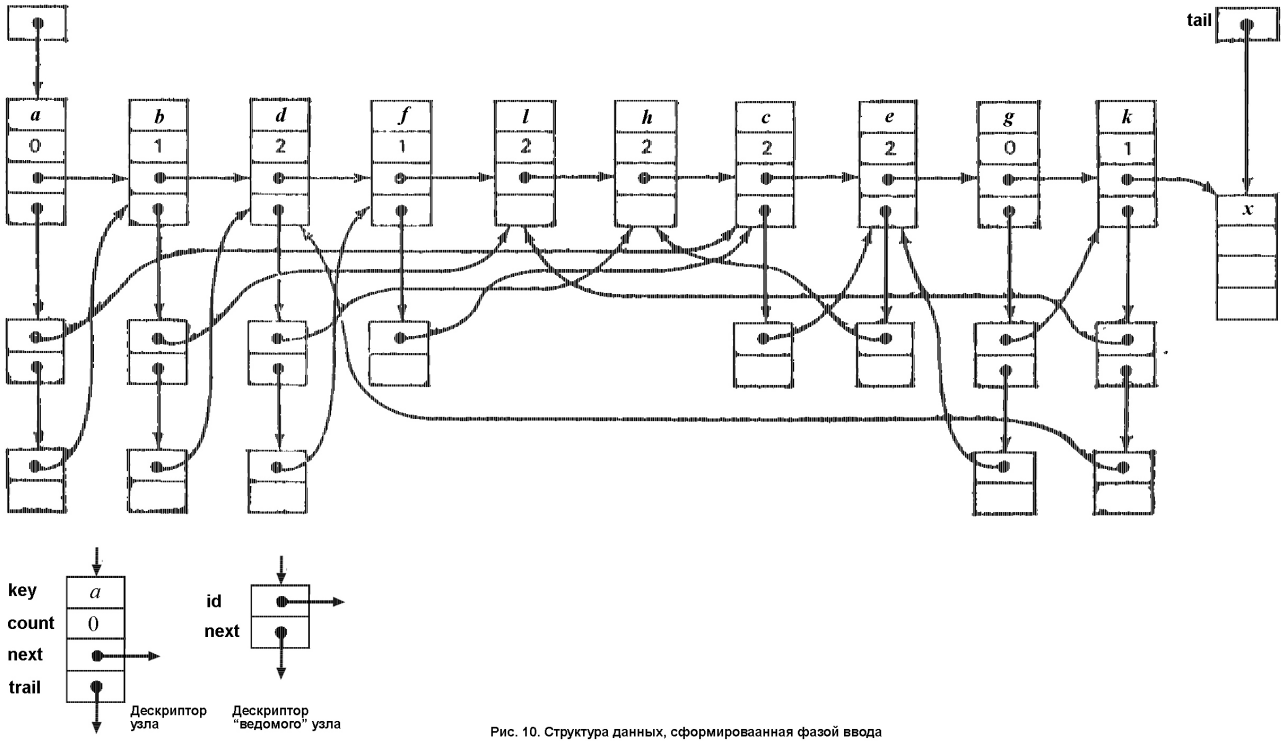

Рис. 10. Структура данных, сформированная фазой ввода

22.2.5. ***Описание алгоритма. Фаза сортировки***.

На фазе сортировки строим цепочку дескрипторов, у которых счетчик имеет значение 0, одновременно корректируя счетчики «ведомых» узлов. Так как с каждой такой коррекцией значение счетчика уменьшается на 1, постепенно значения всех счетчиков становятся равными 0, и соответствующие узлы включаются в результирующую цепочку.

22.2.6. ***Описание алгоритма топологической сортировки на языке Си***.

Программный файл:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct ldr {
   char key;
   int count;
   struct ldr *next;
   struct trl *trail;
} leader;                          /*дескриптор ведущего узла*/
typedef struct tlr {
   struct ldr *id;
   struct trl *next;
} trailer;                         /*дескриптор ведомого узла*/
leader *head, *tail; /*два вспомогательных узла*/
int n;                          /*счетчик ведущих узлов*/

leader *find(char w) {             /* поиск по ключу w */
```

```
    leader *h;
    h = head;
    tail->key = w;    /* "барьер" на случай отсутствия w*/
    while(h->key != w) h = h->next;
    if(h == tail) {   /* генерация нового ведущего узла */
      tail = malloc(sizeof(leader)); /* новый tail */
      /* старый tail становится новым элементом списка */
      n += 1;
      h->count = 0;
      h->trail = NULL;
      h->next = tail;
    }
    return h;
}

void init_list() {
    /* инициализация списка «ведущих» */
    leader *p, *q;
    trailer *t;
    char x, y;
    head = malloc(sizeof(leader));
    tail = head;
    n = 0;       /* начальная установка */
    while(признак конца ввода) {
      scanf("%c %c", &x, &y); /* ввод очередной пары: x y*/
      /* включение пары в список */
      p = find(x);
      q = find(y);
      /* коррекция списка */
      t = malloc(sizeof(trailer));
      t->id = q;
      t->next = p->trail;
      p->trail = t;
      q->count += 1;
    }
}

/* Исходный список построен. Организация нового списка */

void sort_list() {
    leader *p, *q;
    trailer *t;

/* В новый список включаются все узлы старого с count == 0 */
    p = head;
    head = NULL;
    while(p != tail) {
      q = p;
```

```
      p = q->next;
      if(q->count == 0) { /* включение q в новый список */
       q->next = head;
       head = q;
      }
    }

/* Фаза сортировки и вывода результатов из нового списка */
    q = head;
    while(q != NULL) {
      printf("%c\n", q->key);
      n -= 1;
      t = q->trail;
      q = q->next;
      while(t != NULL) {
        p = t->id;
        p->count -= 1;
        if(p->count == 0){
         p->next = q;
         q = p;
        }
        t = t->next;
        /* здесь можно удалить ведомый элемент */
      }
    }
}
int main() {
       init_list();
       sort_list();
       return 0;
}
```