Лекция 21 Обход двоичного дерева

21.1. ***Определение двоичного дерева***. *Двоичное дерево* – это множество узлов, которое либо является пустым, либо состоит из *корня* и двух непересекающихся двоичных деревьев, которые называются *левым* и *правым* поддеревьями данного корня.

21.1.1. ***Замечание***. Двоичное дерево *не является* частным случаем обычного дерева, хотя у этих структур много общего. Основные отличия: (1) Пустое дерево является двоичным деревом, но не является обычным деревом. (2) Деревья (*A*(*B*, *NULL*)) и (*A*(*NULL*, *B*)) различны, если их рассматривать как двоичные деревья, и одинаковы, если их рассматривать как обычные деревья.

21.2. Представление двоичного дерева в памяти компьютера (см. рисунок)

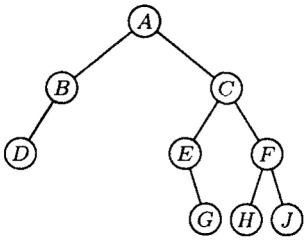

Рис. 1. Двоичное дерево

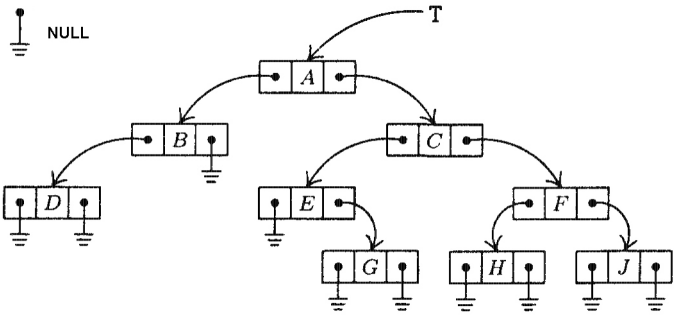

Рис. 2 Представление дерева с рис.1 в компьютере.

Описание узла двоичного дерева на Си:

```
typedef struct bin_tree {
          struct bin_tree *left;
          char info;
          struct bin_tree *right;
        } node;
```

21.3. ***Обход двоичного дерева***.

21.3.1. Обход дерева позволяет получить линейно упорядоченную последовательность (массив) узлов дерева, причем порядок узлов дерева в этом массиве таков, что их можно обрабатывать в цикле **`for(i = 0; i < N; i++)`**, так как при переходе к обработке очередного узла все узлы, от которых он зависит, будут уже обработаны. Такой порядок называется *топологическим*.

21.3.2. Различные способы обхода двоичного дерева.

(1) Обход в *прямом порядке*: обработать корень, обойти левое поддерево, обойти правое поддерево. Порядок обработки узлов дерева с рис. 1: *A B D C E G F H J*. Линейная последовательность узлов, полученная при прямом обходе, отражает «спуск» информации от корня дерева к листьям.

(2) Обход в *обратном порядке*: обойти левое поддерево, обойти правое поддерево, обработать корень. Порядок обработки узлов дерева с рис. 1: *D B G E H J F C A*.

Линейная последовательность узлов, полученная при прямом обходе, отражает «подъем» информации от листьев к корню дерева.

(3) *Симметричный* обход (обход в *симметричном порядке*): обойти левое поддерево, обработать корень, обойти правое поддерево. Порядок обработки узлов дерева с рис. 1: *D B A E G C H F J*

(4) Обход двоичного дерева *в ширину*: узлы дерева обрабатываются «по уровням» (*уровень* составляют все узлы, находящиеся на одинаковом расстоянии от корня). Порядок обработки узлов дерева с рис. 1: *A B C D E F G H J*

21.3.3. Функции, реализующие обходы двоичного дерева, позволяют по указателю каждого узла дерева **P** вычислить указатели узлов **P_next_pre**, **P_next_post** и **P_next_in**, а также указатели узлов **P_pred_pre**, **P_pred_post** и **P_pred_in**.

21.3.4. Рекурсивные функции обхода двоичного дерева на Си:

(1)
```
void preorder (*node r) {
      if(r == NULL) return;
      if(r->info) printf("%c", r->info);
      preorder (r->left);
      preorder (r->right);
  }
```

(2)
```
void postorder (*node r) {
      if(r == NULL) return;
      postorder (r->left);
      postorder (r->right);
      if(r->info) printf("%c", r->info);
  }
```

(3)
```
void inorder (*node r) {
      if(r == NULL) return;
      inorder (r->left);
      if(r->info) printf("%c", r->info);
      inorder (r->right);
  }
```

(4)

21.3.5. Нерекурсивная функция обхода двоичного дерева (управление стеком ведется не автоматически, а в самой функции).

21.3.5.1. **T** – указатель на корень дерева; **P** – указатель на корень обрабатываемого (текущего) поддерева; **Stack[D]** – массив, на котором моделируется стек, **D** – глубина стека (устанавливается препроцессором с помощью **#DEFINE**) **bottom = Stack = Stack[0]** – указатель дна стека; **top** – указатель вершины стека;

21.3.5.2. ***Алгоритм***:

(1) [Инициализация]. Сделать стек пустым, т.е. затолкнуть **NULL** на дно стека: **Stack[0] = NULL;** установить указатель стека на дно стека: **top = 0;** установить указатель **P** на корень дерева: **P = T**.

(2) [Конец ветви]. Если **P == NULL** перейти к шагу (4).

(3) [Продолжение ветви]. Затолкнуть **P** в стек: **Stack[top] = P; top += 1;** установить **P = P->left** и вернуться к шагу (2).

(4) [К обработке правой ветви]. Вытолкнуть верхний элемент стека в **P**: **`P = Stack[top]; top -= 1;`** Если **`P == NULL`** выполнение алгоритма прекращается, иначе обработать данные узла, на который указывает **P**, и перейти к шагу (5).

(5) [Начало обработки правой ветви]. Установить **`P = P->right`** и вернуться к шагу (2).

21.3.5.3. ***Идея алгоритма*** в сохранении указателя **P** в стеке с последующим обходом левого поддерева; когда обход левого поддерева заканчивается, из стека извлекается указатель родителя текущего узла, обрабатывается информация, содержащаяся в узле, указатель которого извлечен из стека, и начинается обход его правого поддерева.

21.3.5.4. ***Си-функция***:

```
void Inorder(node r, int *order) {
   node *P;
   int Stack[D];
   int top, i = 0;
   Stack[0] = Stack[1] = NULL; top = 1; P = r;
   while(P != NULL) {
     while(P != NULL) {
       Stack[top] = P;
       top++;
       P = P->left;
     }
     P = Stack[top];
     top--;
     order[i] = P->info;
     i++;
     P = P->right;
   }
```

21.4. ***Прошитое двоичное дерево***.

21.4.1. Рассмотрим двоичное дерево, показанное на рисунке 2. Легко видеть, что у этого дерева нулевых указателей, больше, чем ненулевых (10 против 8). Это – типичный случай. Поэтому было предложено записывать вместо нулевых указателей указатели

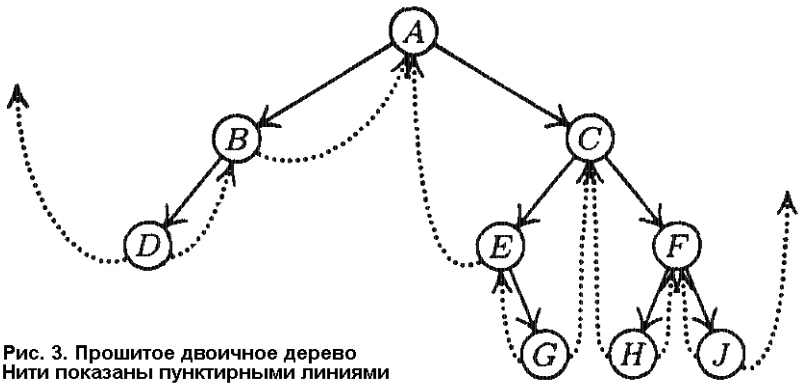

Рис. 3. Прошитое двоичное дерево
Нити показаны пунктирными линиями

на родителей (или более далеких предков) соответствующих узлов (такие указатели называются *нитями*). Это позволит при обходе дерева не использовать стек. Правда, теперь в структуре, описывающей узел дерева, необходимо иметь два поля (тега), принимающих значение 0 для указателя, и 1 для нити (на рисунке 3 нити показаны пунктирными линиями). Полученная структура называется *прошитым двоичным деревом*. В крайнем левом и крайнем правом узлах будут «свободные нити» (они пока никуда не указывают и имеют значение **NULL**), но это только пока: со временем и им будет найдено применение.

21.4.2. Описание узла прошитого двоичного дерева на Си:

```
typedef struct bin_tree {
          int left_tag;
          struct bin_tree *left;
          char info;
          int right_tag;
          struct bin_tree *right;
        } threaded_node;
```

21.4.3. Нити устанавливаются таким образом, чтобы указывать на предшественников (левые нити) или последователей (правые нити) текущего узла при соответствующем обходе дерева. Например, в случае симметричного обхода

Обычное дерево	***Прошитое дерево***
`P->left == NULL`	`P->left_tag == 1, P->left == P_pred_in`
`P->left == Q`	`P->left_tag == 0, P->left == Q`
`P->right == NULL`	`P->right_tag == 1, P->right == P_next_in`
`P->right == Q`	`P->right_tag == 0, P->right == Q`

21.4.4. Нити существенно упрощают алгоритмы обхода двоичных деревьев. Например, для вычисления для каждого узла **P** указатель узла **P_next_in** можно использовать следующий простой ***алгоритм***:

```
node Next_in(node *P) {
     node Q;
     Q = P->right;
     if(P->right_tag ==1) return Q;
     while(Q->left_tag == 0) Q = Q->left;
     return Q;
}
```

Аналогичным образом можно вычислить **P_next_pred** и **P_next_post**

21.4.5. Алгоритм (функция) **Next_in** фактически реализует симметричный обход дерева, так как позволяет для произвольного узла дерева **P** найти **P_next_in**, т.е. применяя эту функцию несколько раз, можно вычислить топологический порядок узлов двоичного дерева, соответствующий симметричному обходу.

Аналогичным образом, применяя функции **P_next_pred** (либо **P_next_post**) можно вычислить топологический порядок узлов, соответствующий прямому (либо обратному) обходу.

21.4.6. ***Замечания***. (1) С помощью обычного представления невозможно для произвольного узла **P** вычислить **P_next_in**, не вычисляя всей последовательности узлов.

(2) Функции **Next_in** не требуется стек ни в явной, ни в неявной (рекурсия) форме.

(3) Сравнительный анализ функций **Inorder()** и **Next_in()** позволяет сделать следующие выводы:

(a) Если **P** – произвольно выбранный узел дерева, то следующий фрагмент функции **Next_in()**:

```
Q = P->right;
if(P->right_tag ==1) return Q;
```

выполняется только один раз.

(b) Обход прошитого дерева выполняется несколько быстрее, так как для него не нужно выполнять операции со стеком.

(c) Для функции **Inorder()** требуется больше памяти, чем для функции **Next_in()**, из-за использования стека, реализованного на массиве **Stack[D]**, при этом, с одной стороны, желательно, чтобы **D** было не очень большим, а с другой стороны, **D** не может быть меньше высоты двоичного дерева, которое необходимо обойти. При переполнении стека могут возникнуть крайне неприятные последствия, так что на практике **D** – достаточно большое число. Это особенно верно, когда в программе необходимо выполнить обход нескольких двоичных деревьев. Кроме того, можно доказать, что функция **Inorder()** работает примерно в два раза дольше, чем функция **Next_in()**.

(d) Алгоритм, реализуемый функцией **Next_in()**, более общий, чем алгоритм, реализуемый функцией **Inorder()**, так как он позволяет непосредственно перейти от узла **P** к узлу **P_next_in**, не выполняя обхода соответствующего двоичного дерева.

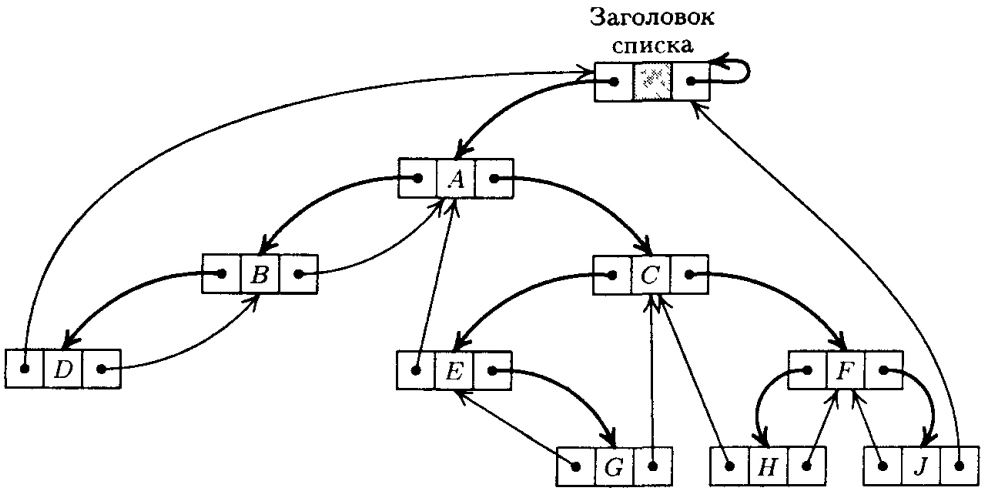

Рис. 4. Прошитое двоичное дерево с заголовком

21.4.7. В функции **Inorder()** используется указатель **r** на корень двоичного дерева. Желательно, применив функцию **Next_in()** к корню **r**, получить указатель узла дерева, следующего за для выбранного порядка обхода. Поэтому к дереву добавляется еще один узел – заголовок дерева (см. рисунок 4). При этом, поля структуры

```
struct bin_tree {
        int left_tag;
        struct bin_tree *left;
```

```
            char info;
            int right_tag;
            struct bin_tree *right;
        } *header;
```

заполняются следующим образом (рисунок 4):

```
header->left_tag = header->right_tag = 0;
header->left = r; header->right = header.
```

На рисунке 4 дуги дерева показаны более жирными линиями, чем нити.