Тема 2: РЕКУРСИЯ и ДЕРЕВЬЯ

1. Введение

В этой теме будут даны некоторые дополнительные объяснения и иллюстративные примеры использования рекурсии и деревьев в программах.

Вам необходимо вспомнить программу рекурсивного вычисления факториала и написанную на ее основе рекурсивную программу печати связного линейного списка.

Будет также описана программа, которая рекурсивно строит двоичное дерево, и функция, которая позволяет рекурсивно обходить его. Используя их, вы должны будете дополнить программу так, чтобы она искала в данном дереве определенную информацию (подробнее о заданиях см. п.5).

Теперь мы проиллюстрируем применение рекурсии и стандартный подход к построению двоичного дерева, а также дадим несколько комментариев к нашим программам и функциям. Советуем также заглянуть в соответствующее место ваших конспектов или учебников.

2. Применение рекурсии

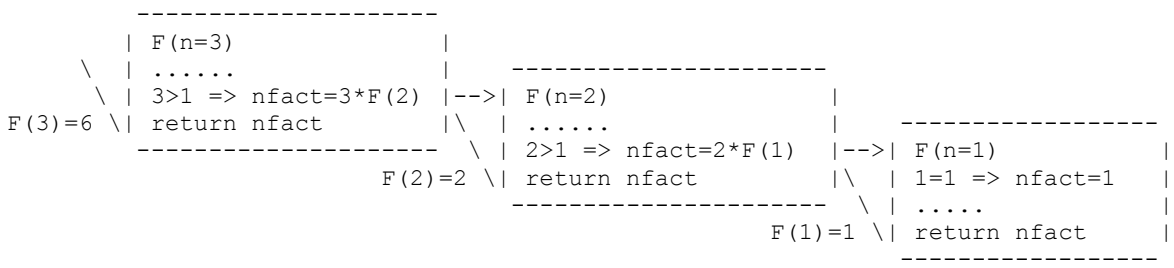
Рекурсия — мощное средство для решения ряда задач на компьютере. Рассмотрим суть рекурсии на простом примере вычисления факториала целого числа. Для факториала известно рекуррентное соотношение:

```
n! = 1, если n = 0 или 1;
n! = (n-1)! n, если n > 1,
```

На языке Си функция вычисления факториала может быть записана в рекурсивном виде так

```
int F (int n)
{
 int nfact;
 if (n==0||n==1) nfact=1; /* !!! Условие окончания рекурсии !!! */
 if (n>1) nfact = n*F(n - 1);
 return nfact;
}
```

Схематично работу (рекурсивный вызов) данной функции для n = 3 можно представить следующей схемой:



При n=3 происходит рекурсивный вызов функции для n=2 и далее еще раз для n=1. Обратите внимание на то, что после окончания работы функции при n=1 происходит возврат в ту же точку, из которой она вызывалась, и продолжение работы при n=2. По окончании ее имеет место возврат в функцию при n=3 и завершение работы.

Рекурсия оказывается эффективной при решении многих задач искусственного интеллекта. Широко применяется рекурсия и при работе с нелинейными структурами данных, например с деревьями.

3. Работа с деревьями

Рассмотрим наиболее часто встречающиеся двоичные деревья. Каждый элемент такого дерева обычно состоит из трех полей:

```
a - содержит значение Ai (для простоты целое);
l - содержит адрес памяти Li корня левого поддерева (тип указатель)
    или NULL;
r - содержит адрес памяти Ri корня правого поддерева (тип указатель)
    или NULL;
```

где поддерево есть дерево, имеющее соответственно на один уровень в глубину меньше исходного дерева, а NULL — это некоторая известная компьютеру (компилятору) "ссылка в никуда" (в СИ это обычно "0").

Тогда i-й элемент дерева будет:

```
 ____ ____ ____
|_Ai_|_Li_|_Ri_|
```

Если P — это указатель на корень дерева ("верхний" элемент), то для трех элементов дерево может быть таким:

```
          ____ ____ ____
P--->|_A1_|_L1_|_R1_|--
             |          |    ____ ______ ______
             |           -->|_A3_|_NULL_|_NULL_|
             |     ____ ______ ______
              -->|_A2_|_NULL_|_NULL_|
```

Элементы дерева которые не имеют потомков называются листьями, а остальные внутренними узлами.

Будем строить дерево по следующему принципу: *значения всех элементов левого поддерева меньше значения корня, а значения всех элементов правого поддерева — больше либо равны ему*. При этом под корнем будем понимать как корень самого дерева, так и корни его поддеревьев. Включать новый элемент будем в виде листа дерева, т.е. новый лист не будет иметь потомков.

Приведем пример построения такого двоичного дерева для следующей последовательности целых чисел:

```
6, 3, 10, 8, 11, 4, 7, 9, 0.
```

Очевидно, что первое число должно стать корнем такого дерева из одного элемента:

```
          _____ ______ ______
P--->|__6__|_NULL_|_NULL_|
```

или в более наглядном абстрактном виде:

```
        6
       / \
   NULL   NULL
```

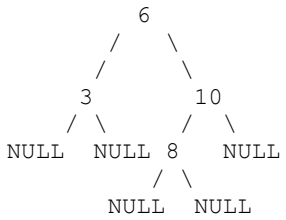
Поскольку второе число меньше 6, оно включается в левое поддерево, и получаем

```
        6
       / \
      /   \
     3     NULL
    / \
NULL   NULL
```

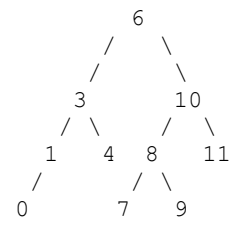
Третье число больше 6, и поэтому оно включается в правое поддерево

```
        6
      /   \
     /     \
    3       10
   / \     /  \
  /   \   /    \
NULL  NULL NULL  NULL
```

Четвертое число равно 8. Поскольку 8 > 6, это число должно быть включен в правое поддерево дерева с корнем 6. Это поддерево содержит в свою очередь дерево с корнем 10. Поскольку 8 < 10, то число 8 включается в левое поддерево дерева с корнем 10. В результате получается



Продолжая таким образом, для данной последовательности чисел в конце концов будет построено следующее дерево:



где все NULL для простоты опущены. Легко видеть, что построенное дерево удовлетворяет заданным правилам: *значения всех элементов левого поддерева меньше значения корня, а значения всех элементов правого поддерева — больше либо равны ему*.

4. Комментарии к программам

4.1. Программа расчета факториала тривиальна и не требует дополнительных комментариев (см. ее текст выше).

4.2. Вторая программа (tr_head.h, tr_main.c, tr_in.c, tr_find.c, tr_pass.c) строит дерево и затем обходит.

После описаний локальных переменных, в функции main() (см. tr_main.c), происходит их инициализация

```
Root = NULL;
NewInfo = 0;
```

Затем в цикле печатается приглашение –

```
Введите целое число:
```

После ввода числа, если оно не равно 9999, происходит вызов рекурсивной функции включения элемента в дерево — InclElem (см. tr_in.c).

Если введенное число равно 9999, то построение дерева заканчивается и происходит обход дерева с использованием функции PassTree (см. tr_pass.c).

4.3. Функция InclElem (NewInfo, Tree) включает элемент со значением NewInfo в дерево Tree и возвращает ссылку на корень нового дерева.

```
1) ЕСЛИ корень дерева (поддерева) Root = NULL (Определяем,
   закончен ли обход соответствующего дерева и можно ли
   включать новый элемент), ТО:
   а) создаем новый элемент дерева, используя функцию malloc,
      и указатель на новый элемент (значение функции malloc)
      заносим в переменную Root;
   б) заносим в поле Info нового элемента значение NewInfo,
      переданное в функцию;
   в) заносим в поля LeftNext и RightNext константу NULL
      (т.е. указатель "в никуда").

2) ИНАЧЕ производим сравнение включаемого значения (NewInfo)
   со значением, находящемся в корне (в поле Info) (Определяем
```

```
   в правое или в левое поддерево должен быть включен элемент), -
   а) если первое меньше, то включаем элемент в левое поддерево:
      полю LeftNext корня присваиваем значение
      функции InclElem, вызванной с аргументами NewInfo и
      Root->LeftNext (указатель на левое поддерево);
   б) в противном случае, включаем элемент в правое поддерево:
      полю RightNext корня присваиваем значение
      функции InclElem, вызванной с аргументами NewInfo и
      Root->RightNext (указатель на правое поддерево);

3) Возвращаем указатель на узел дерева (корень поддерева). Значение
   данного указателя присваивается соотвественно полю LeftNext или
   RightNext (см. п. 2) узла дерева находящегося на один уровень выше
   рассматриваемого.
```

4.4. Функция PassTree (Tree) делает обход дерева Tree.

Напомним, что обойти дерево значит побывать в каждом его элементе один и только один раз. В основе функции следующий алгоритм:

```
ЕСЛИ ссылка Root на элемент дерева (или, что тоже самое,
на корень исходящего из него поддерева!) не NULL,
ТО (если нужно, проанализировать заключенную в элементе информацию и)
перейти к обходу левого поддерева, а по окончанию этого
к обходу правого поддерева,
ИНАЧЕ завершить обход поддерева (т.е. подняться по дереву
рекурсии на один уровень выше).
```

Более кратко данный алгоритм записывается так:

```
Обойти дерево (поддерево):
  1. Анализировать корень.
  2. Обойти левое поддерево.
  3. Обойти правое поддерево.
```