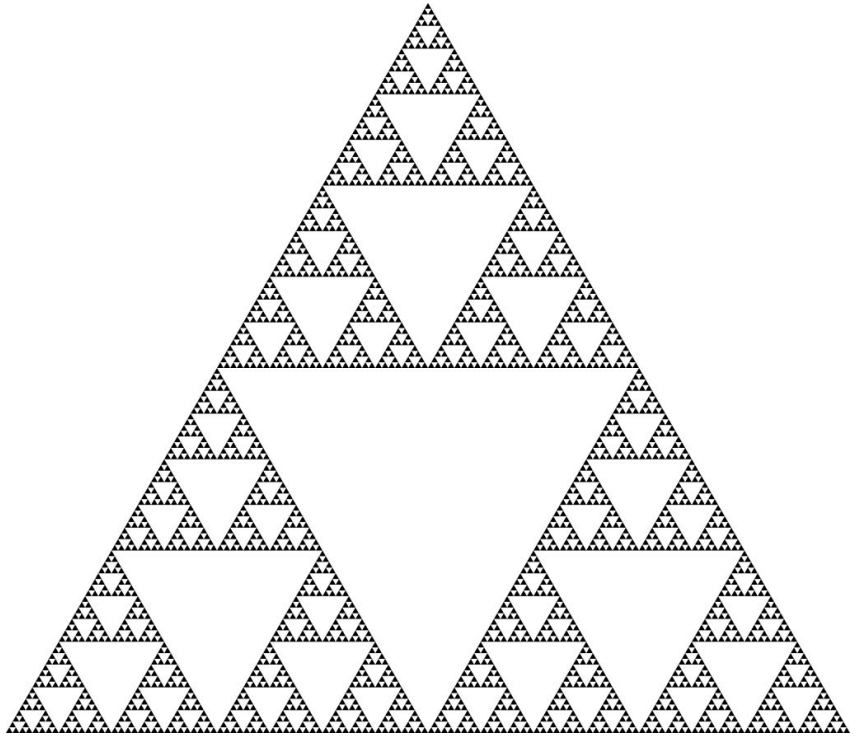
Рекурсия и деревья

ЛЕКЦИЯ №8

Рекурсия

Рекурсивным называется такой способ построения объекта, в котором определение объекта включает аналогичный объект в виде некоторой его части. Существуют рекурсивные структуры данных



Линейная рекурсия

Простейшим примером рекурсии является линейная рекурсия, когда функция содержит единственный условный вызов самой себя. В таком случае рекурсия становится эквивалентной обычному циклу. Действительно, любой циклический алгоритм можно преобразовать в линейно-рекурсивный и наоборот. Продемонстрируем это на примере вычисления факториала:

```
int fact(int n)
{
if (n==1) return n;
return n * fact(n-1);
}
```

Рекурсия и поисковые задачи

С помощью рекурсии легко решаются задачи, связанные с поиском, основанном на полном или частичном переборе возможных вариантов.

Принцип рекурсивности заключается здесь в следующем:

1. Процесс поиска разбивается на шаги;
2. На каждом выбирается и проверяется очередной элемент из множества;
3. Алгоритм поиска повторяется, но уже для «оставшихся» данных. При этом вовсе не важно, каким образом цепочка шагов достигнет цели и сколько вариантов будет перебираться.

Само множество, в котором производится поиск, обычно реализуется в виде глобальных данных, в которых каждый шаг выбирает необходимые элементы, а по завершении поиска возвращает их обратно.

Результат рекурсивной функции

Если рекурсивная функция имеет результат **void**, то она не может повлиять на характер протекания процесса поиска и реализуемый алгоритм будет выполнять полный перебор всех возможных вариантов;

Если рекурсивная функция выполняет поиск первого попавшегося варианта, то результатом ее является как правило **логическое значение** (в Си - 0/1). При этом «**ИСТИНА**» соответствует **успешному** завершению поиска, а «**ЛОЖЬ**» - **неудачному**. Общим для всех алгоритмов поиска является: если рекурсивный вызов возвращает «ИСТИНУ», то она должна быть немедленно «передана наверх», то есть текущий вызов также должен быть завершен со значением «ИСТИНА». Если рекурсивный вызов возвращает «ЛОЖЬ», по поиск должен быть продолжен. При завершении полного перебора всех вариантов рекурсивная функция также должна возвратить «ЛОЖЬ»;

Если в процессе поиска производится более сложный анализ и сравнение вариантов, то рекурсивная функция и, соответственно, шаг процесса должны **производить выбор между подходящими вариантами, в целью выбора наиболее оптимального**. Обычно для этого используется минимум или максимум какой-либо характеристики выбираемого варианта. Тогда рекурсивная функция **возвращает значение, которое является оценкой для оставшихся не просмотренных элементов, а текущий рекурсивный вызов выбирает из них минимум или максимум с учетом данных текущего шага**.

Рекурсивная структура данных

По аналогии с рекурсивным вызовом функции существуют структуры данных, допускающие рекурсивное определение: элемент структуры данных содержит один или несколько указателей на элементы такого же типа. Формально это соответствует тому факту, что в определении структурированного типа содержится указатель на себя самого.

```
struct xxx
{struct xxx *pp; ...}
```

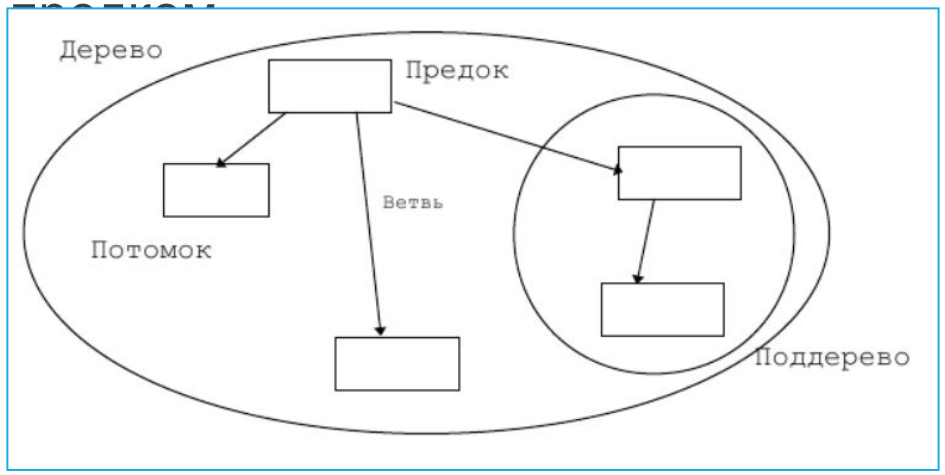
Из подобных структур данных были рассмотрены списки. Однако функции, работающие с ними, не были рекурсивными.

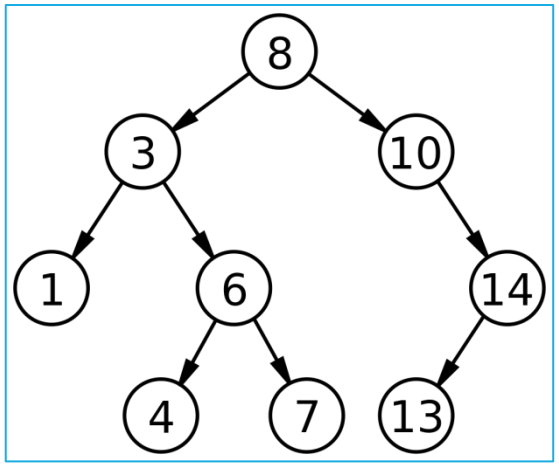
Деревья

Определение дерева имеет исключительно рекурсивную природу. Элемент этой структуры данных называется вершиной (**узлом**).

Дерево представляет собой либо отдельную вершину, либо вершину, имеющую ограниченное число указателей другие деревья (ветвей).

Нижележащие деревья для текущей вершины называются поддеревьями, а их вершины – потомками. По отношению к потомкам текущая вершина называется





Деревья

С точки зрения лексики языка Си, вершину дерева можно определить как:

```
struct tree
{
int val; // Значение элемента
tree *child[4]; // Указатели на потомков
};
```

Само дерево обычно задается в программе указателем на его главную вершину.
Из определения элемента дерева следует только тот факт, что он имеет ограниченное число указателей на подобные элементы.
Как и во всех динамических структурах данных характер связей между элементами определяется функциями, которые их устанавливают.

Работа с деревьями

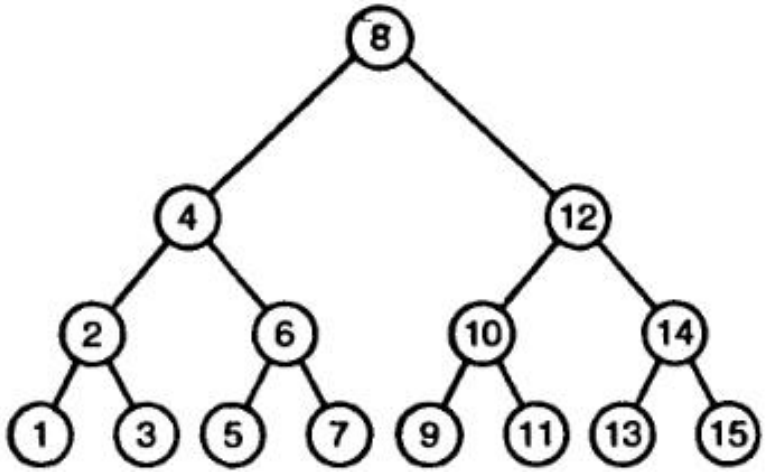
```
//------Рекурсивный обход дерева
void ScanTree(tree *p)
{
int i;
if (p == NULL)
    return; // следующей вершины нет
for (i = 0; i < NUM_NODS; i++) // рекурсивный обход
    ScanTree(p->child[i]); // потомков с передачей
} // указателей на них
```

Часто обход дерева используется для получения информации, которая затем возвращается через результат рекурсивной функции. Существует три типа обхода деревьев. Префиксный (сверху вниз), инфиксный (слева направо), постфиксный (снизу вверх)

Бинарные деревья

Бинарное (двоичное) дерево поиска – это бинарное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

1. Оба поддерева – левое и правое, являются двоичными деревьями поиска;
2. У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, чем значение ключа данных самого узла X;
3. У всех узлов правого поддерева произвольного узла X значения ключей данных не меньше, чем значение ключа данных узла X.



Данные в каждом узле должны обладать ключами, на которых определена операция сравнения.
Как правило, информация, представляющая каждый узел, является записью, а не единственным полем данных.

Бинарные деревья. Структура и обход

```
struct tnode
{ int field; // поле данных
struct tnode *left; // левый потомок struct
tnode *right; // правый потомок
};
```

```
void treeprint(tnode *tree) {
  if (tree!=NULL) { //Пока не встретится пустой узел
    treeprint(tree->left); //Рекурсивная функция для левого поддерева
    treeprint(tree->right); //Рекурсивная функция для правого поддерева
  }
}
```

Добавление нового элемента в бинарное дерево

```
struct tnode * addnode(int x, tnode *tree) {
  if (tree == NULL) { // Если дерева нет, то формируем корень
    tree = (tnode)malloc(sizeof(tnode)); // память под узел
    tree->field = x;   // поле данных
    tree->left =  NULL;
    tree->right = NULL; // ветви инициализируем пустотой
  }else  if (x < tree->field)   // условие добавление левого потомка
    tree->left = addnode(x,tree->left);
  else    // условие добавление правого потомка
    tree->right = addnode(x,tree->right);
  return(tree);
}
```

Удаление элемента из дерева

```
void freemem(tnode *tree) {
  if(tree!=NULL) {
    freemem(tree->left);
    freemem(tree->right);
    delete tree;
  }
}
```

Пример. Сортировка элементов массива с помощью дерева

Написать программу, подсчитывающую частоту встречаемости слов входного потока.

Поскольку список слов заранее не известен, мы не можем предварительно упорядочить его. Неразумно пользоваться линейным поиском каждого полученного слова, чтобы определять, встречалось оно ранее или нет, т.к. в этом случае программа работает слишком медленно.

Один из способов — постоянно поддерживать упорядоченность уже полученных слов, помещая каждое новое слово в такое место, чтобы не нарушалась имеющаяся упорядоченность. Воспользуемся бинарным

В дереве каждый узел содержит:

1. указатель на текст слова;
2. счетчик числа встречаемости;
3. указатель на левого потомка;
4. указатель на правого потомка.