Деревья

Дерево есть конечное множество **T** узлов, такое, что:

1. имеется один специально обозначенный узел, называемый корнем дерева.
2. остальные узлы содержатся в **m**≥0 попарно непересекающихся множествах $T_1, T_2, \ldots T_m$, каждое из которых является деревом.

Деревья $T_1, T_2, \ldots T_m$ называются поддеревьями данного корня.

Строго говоря, приведенное определение относится к специальному случаю дерева, называемому *корневым* деревом.

Деревья, в которых корень не выделен, называют *висячими*. Мы будем рассматривать исключительно корневые деревья.

Число поддеревьев узла называют его степенью. Узел нулевой степени называют *листом*.

Уровень узла в дереве определяется следующим образом:

Корень имеет уровень 1

Корни поддеревьев узла имеют уровень на 1 больший, чем уровень узла.

Высотой дерева называют наибольший уровень узла в нём.

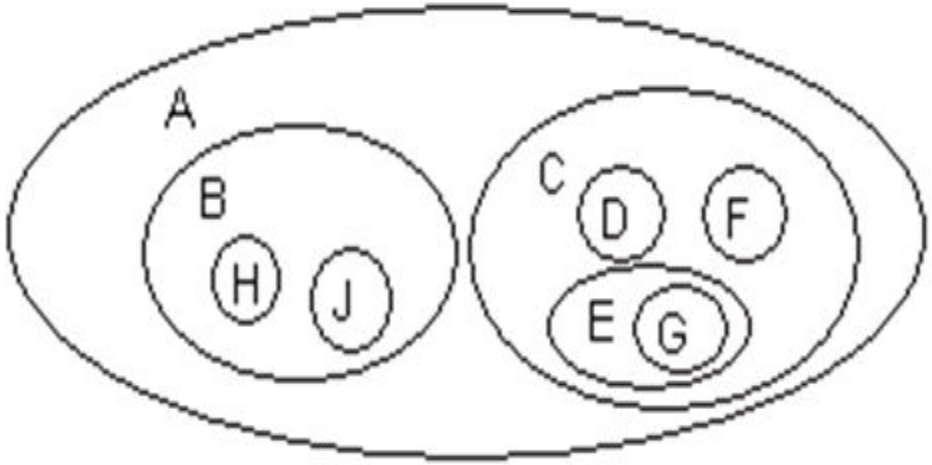
Если в определения дерева имеет значение относительный порядок следования поддеревьев $T_1, T_2, \ldots T_m$, то дерево называют упорядоченным.

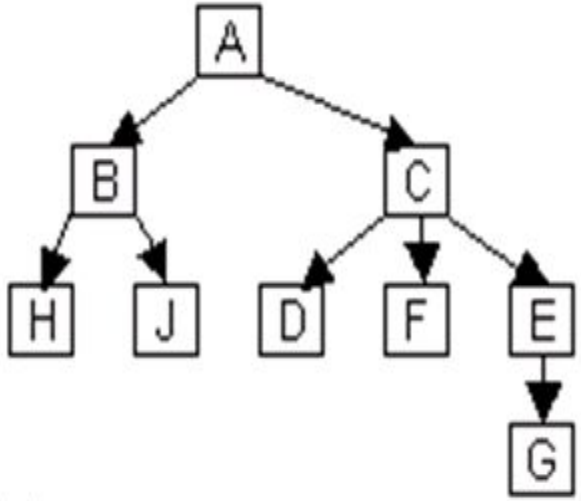
Лес – это множество, быть может, пустое, состоящее из некоторого числа непересекающихся деревьев.

Определение дерева рекурсивно и также рекурсивными являются большинство методов обработки деревьев.

Дерево можно изобразить различными способами.

Изображения деревьев на рис. эквивалентны.





(A(B(H)(J))(C(D)(E(G))(F)))

Различные изображения деревьев

Примерами древовидной структуры являются генеалогические деревья, оглавления, формулы, классификации. На рис. изображены деревья, изображающие формулу **a-b*(c/d+e/f)** и классификации товаров.

Примеры деревьев

Когда говорят о деревьях, часто используют такие термины, как "отец", "сын", "брат", "предок", "потомок".

Каждый узел является отцом корней своих поддеревьев. Последние являются братьями между собой и сыновьями своего отца.

В упорядоченном дереве левый брат считается старшим.

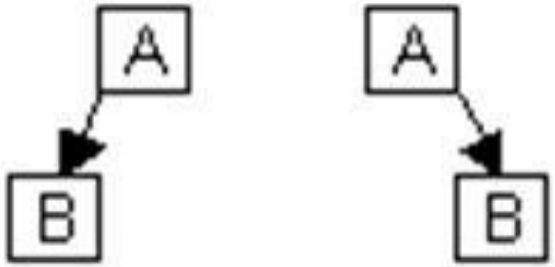
По аналогии можно ввести термины "дядя", "племянник" и другие.

Термины "предок" и "потомок" употребляются для обозначения родства, простирающегося на несколько уровней дерева.

Бинарные деревья

Бинарное дерево определяется как конечное множество узлов, которое или пусто, или состоит из корня и двух непересекающихся бинарных деревьев, называемых левым и правым поддеревьями корня.

Отметим, что деревья на рисунке различны, так как в одном случае пусто левое поддерево, а в другом правое.



Узел бинарного дерева может быть представлен структурой:

```
struct NODE{
   <тип> <поле данных>;
   NODE *Llink; // указатель на левого сына
   NODE *Rlink; // указатель на правого сына
};
```

Обход бинарного дерева

Для работы с древовидными структурами имеется множество алгоритмов, и многие из них используют одну и ту же идею, а именно идею прохождения или обхода дерева.

Обход дерева подразумевает такой порядок работы с его узлами, для которого каждый из них посещается точно один раз.

Для обхода бинарного дерева могут быть использованы три способа, определяемых рекурсивно.

Прямой обход.

1. Обработать корень
2. Обойти левое поддерево
3. Обойти правое поддерево

Обратный обход.

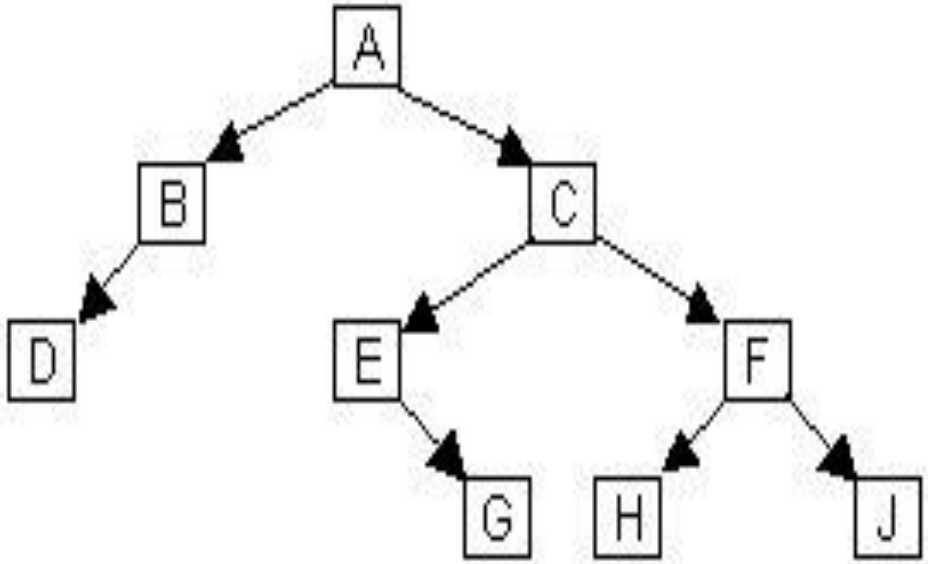
1. Обойти левое поддерево
2. Обработать корень
3. Обойти правое поддерево

Концевой обход.

1. Обойти левое поддерево
2. Обойти правое поддерево
3. Обработать корень

На рис. изображены все варианты обхода.

На рис. изображено бинарное дерево и порядок следования его узлов для различных методов обхода.



Прямой обход A B D C E G F H J

Обратный обход D B A E G C H F J

Концевой обход D B G E H J F C A

Пример рекурсивной функции, выполняющей обход дерева в прямом порядке.

```
void DirectBypass(NODE *Root){
// Root – указатель на корень дерева
if(Root==NULL) return; // дерево пусто
Обработка(Root); // делаем то, что нужно с узлом
DirectBypass(Root->Llink); // пройти левое поддерево
DirectBypass(Root->Rlink); // пройти правое поддерево
}
```

Обратный и концевой обходы отличаются только местоположением оператора

Идея реализации алгоритма с "ручным" ведения стека (это может потребоваться, если язык не допускает рекурсии) заключается в том, что мы запоминаем в стеке историю спуска по левым ветвям дерева для того, чтобы впоследствии можно было вернуться и обработать оставшиеся узлы.

```
const int MAXSTACK=50;
void InverseBypass(NODE *Root){
// Нерекурсивный обход бинарного дерева
NODE *stack[MAXSTACK];
NODE *s;
int v=0; // указатель на вершину стека

s=Root;
again:
if(s!=NULL){
    // спускаемся по левым ветвям, запоминая
     историю в стеке
    stack[v++]=s;
    s=s->Llink;
    goto again;
} else {
```

```
if(v==0){ // стек пуст
    return;
  }
  // взять узел из стека
  s=stack[--v];
  Обработка(s);
  // переходим к правому поддереву
  s=s->Rlink;
  goto again;
}
}
```

Рассмотрим две конкретные задачи, решаемые с помощью обхода дерева.

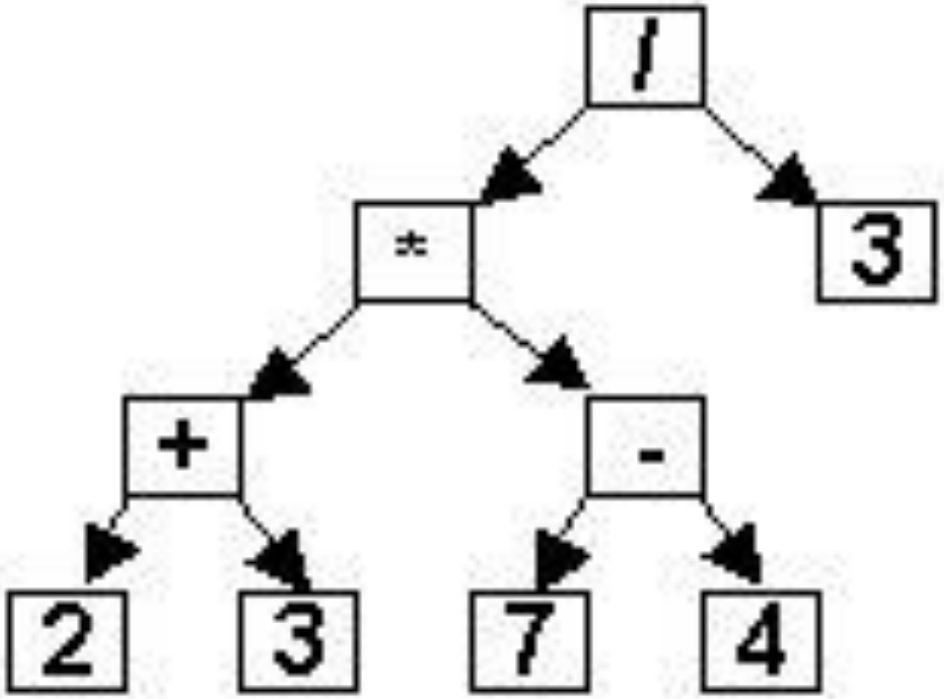
Задача 1. Копирование бинарного дерева

Для решения задачи естественно использовать прямой обход с тем,чтобы узел – отец создавался раньше, чем сыновья.

```
NODE *CopyBinTree(NODE *Root){
// функция имеет указатель на корень
// дерева – оригинала в качестве аргумента
// и возвращает указатель на корень дерева – копии
if(Root==NULL) return NULL;
// создадим корень в копии
NODE *RootCopy=new NODE;
// левый и правый сыновья в дереве – копии являются
// копиями левого и правого поддеревьев корня в
   оригинале
RootCopy->Llink=CopyBinTree(Root->Llink);
RootCopy->Rlink=CopyBinTree(Root->Rlink);
Return RootCopy;
}
```

Задача 2. Вычисление значения выражения, заданного деревом.

В качестве примера рассмотрим выражение ((2+3)*(7-4))/3. Порядок вычисления выражения можно изобразить в виде дерева



Узел дерева в поле данных содержит либо число, либо символ операции. Если узел содержит число, то это операнд, а если операцию, то значения левого и правого поддеревьев суть её операнды.

Вычисление естественно выполнять в порядке концевого обхода, поскольку для того, чтобы выполнить операцию, надо знать её операнды.

Структура узла имеет вид:

```
const int OPERATION=0; // признак: узел содержит операцию
const int NUMBER=1;    // признак: узел содержит число
struct UZEL{
    union {
     char Operation; // символ операции
     float Number;   // число
    };
    int Tag; // может принимать значения OPERATION или NUMBER
    UZEL *Left, *Right; // указатели на сыновей
};
```

Приведенная ниже функция вычисляет значение выражения, заданного деревом.

```
float TreeValue(UZEL *Root){
float Result;
 if(Root->Tag==NUMBER) return Root->Number;
// в узле операция. Найдем её операнды
float x=TreeValue(Root->Left);
float y=TreeValue(Root->Right); // левый и правый операнды
// выполним операцию
switch(Root->Operation){
      case '+':
        Result=x+y;
        break;
      case '-':
        Result=x-y;
        break;
      case '*':
        Result=x*y;
        break;
      case '/':
        Result=x/y;
        break;
}
return Result;
}
```

Голова дерева.

Дерево, как и линейный список, может иметь голову. В таких случаях, дерево, как правило, делают левым поддеревом головы.

При обратном обходе, повторный выход на голову означает завершение алгоритма.

Если дерево имеет голову, то каждый узел имеет отца, что позволяет избавиться от особенностей обработки корня.

Кроме того, как и в случае линейных списков, наличие головы дерева позволяет избавиться от проблем, связанных различением пустого и несуществующего дерева.

Прошитые деревья

В бинарном дереве, содержащем **N** узлов, на каждый узел, кроме корня указывает ровно одна связь. Всего связей **2*N**; непустых - **N-1**, следовательно, **N+1** связь пуста.

Пустые связи существуют только для того, чтобы обозначить, что дальше в этом направлении пути нет, для чего достаточно одного бита.

Возникает вопрос: а нельзя ли более рационально использовать пространство, занимаемое пустыми связями.

Прошитые деревья используют место, занимаемое пустыми связями для хранения указателей, упрощающих прохождение дерева. Эти дополнительные связи называют *нитями*, откуда и появился термин *прошитые*. Введем обозначения:

*P – предшественник узла P в обратном порядке,

P* – последователь узла P в обратном порядке,

$^{+}$P – предшественник в прямом порядке,

P^{+} – последователь в прямом порядке

Дерево может быть прошито для обхода в одном из порядков.

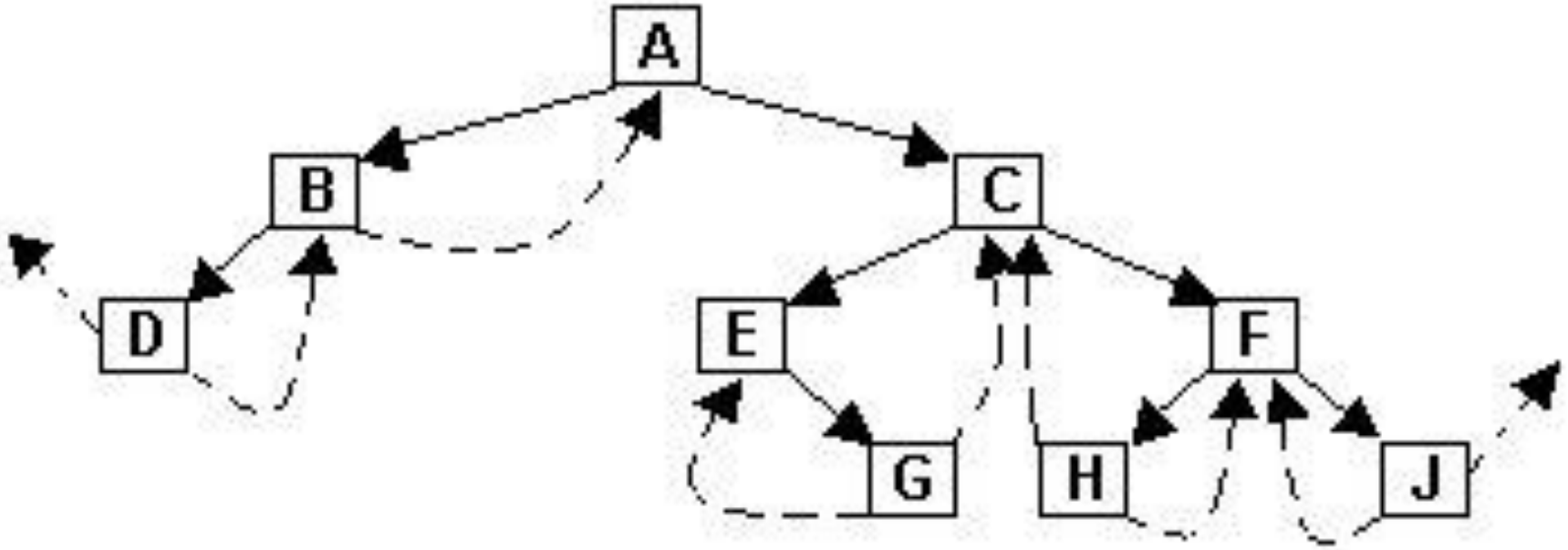
Рассмотрим дерево, прошитое для обхода в обратном порядке. Вместо пустых левых связей будем хранить указатель на предшественника в обратном порядке,

вместо пустых правых связей – указатель на последователя. Эти связи будем называть "нитями".

Для того, чтобы отличать основные связи от нитей, в каждом узле заведем два поля **L** и **R**, которые будут иметь значения **THREAD**, если связь – нить и **MAINLINK**, если связь – основная (**THREAD** и **MAINLINK –** константы). Таким образом, структура узла прошитого дерева имеет вид:

```
const int THREAD=0;
const int MAINLINK=1;
struct NODE{
   <поля данных>;
   NODE *Left, *Right;
   BYTE L,R;
};
```

На рис. изображено прошитое дерево. Пунктиром изображены нити.



Преимущество прошитых деревьев заключается в том, что упрощаются алгоритмы обхода. Ниже приведена функция, возвращающая указатель на последователя **p** в обратном порядке.

```
NODE *NextUzel(NODE *p){
NODE *q=p->Right;
if(p->R==THREAD) return q; // если это нить, то q- результат
// в противном случае спуститься до упора по левым связям
while(q->L==MAINLINK) q=q->Left;
return q;
}
```

При наличии алгоритма определения последователя отпадает необходимость в стеке (явном или порождаемом механизмом реализации рекурсии). Функция, выполняющая обход дерева в обратном порядке принимает вид:

```
void InverseBypass(NODE *Root){
NODE *q=Root;

// найдем первый в обратном порядке узел
// он находится в конце спуска по основным левым связям
while(q->L==MAINLINK)q=q->Left;
// проходим все узлы, используя функцию NextUzel
for(;q!=NULL;q=NextUzel(q)){
    Обработка(q);
}
}
```

Другие представления бинарных деревьев

Подходящий выбор представления дерева в первую очередь определяется видом операций, выполняемым над деревьями.

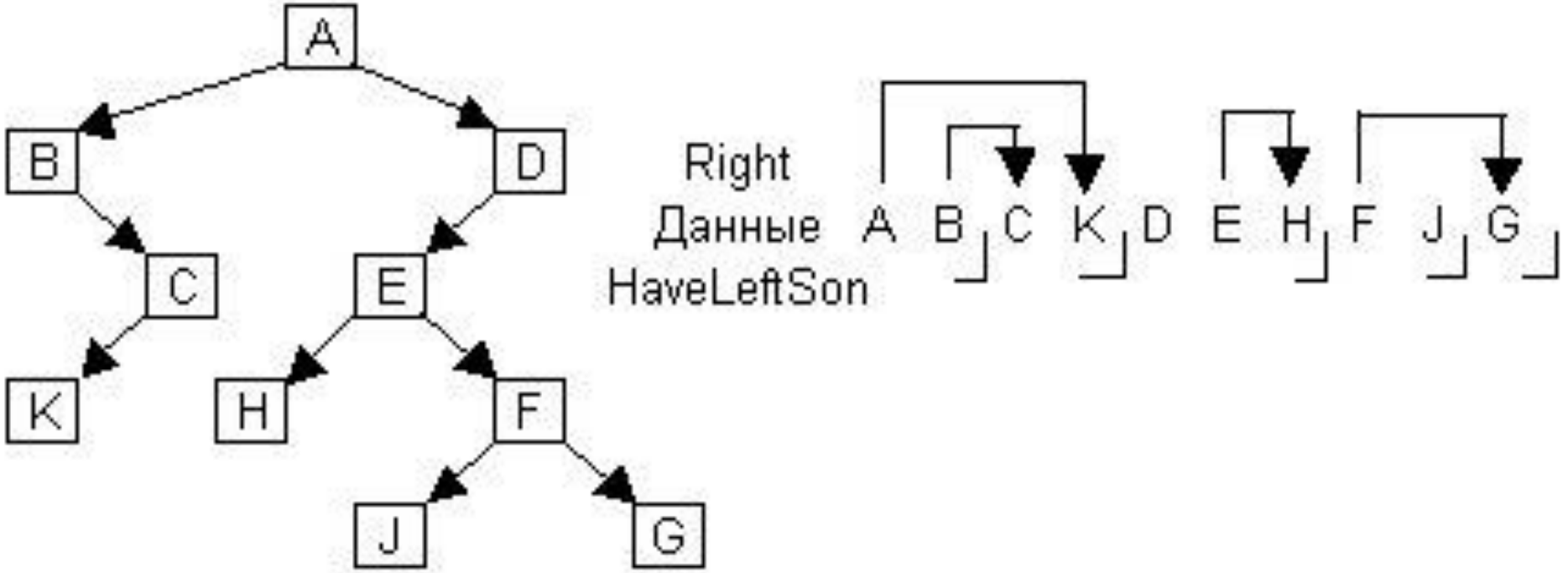
В частности, можно использовать методы последовательного распределения памяти, отображающие связи на физическое размещение данных. Такой способ годится, когда требуется компактное представление дерева, и оно не будет подвергаться радикальным динамическим изменениям в процессе работы программы.

Он заключается в том, что опускается поле **Left** или **Right**, а последовательное расположение узлов замещает опущенную связь. Структура узла имеет вид:

```
struct NODE{
   <поля данных>;
   NODE *Right;
   bool HaveLeftSon;
};
```

Хранится только правая связь. Левый сын узла, если он есть, расположен в памяти сразу за данным. Поле HaveLeftSon имеет значение true, если узел имеет левого сына. Узлы в памяти хранятся в порядке прямого обхода.

Следующую картинку с ошибкой переделать


A
B
D
C
E
K
H
F
J
G
Right
Данные
HaveLeftSon
A B C K D E H F J G

Представление деревьев общего вида

Рассмотрим два варианта представления дерева общего вида. В первом варианте для хранения указателей на сыновей используется массив фиксированной длины:

```
const  int MAXSON=10; //  максимально возможное число сыновей
struct NODE {
   <поля данных>;
   int nSon; // действительное число сыновей узла
   NODE *Sons[MAXSON];
};
```

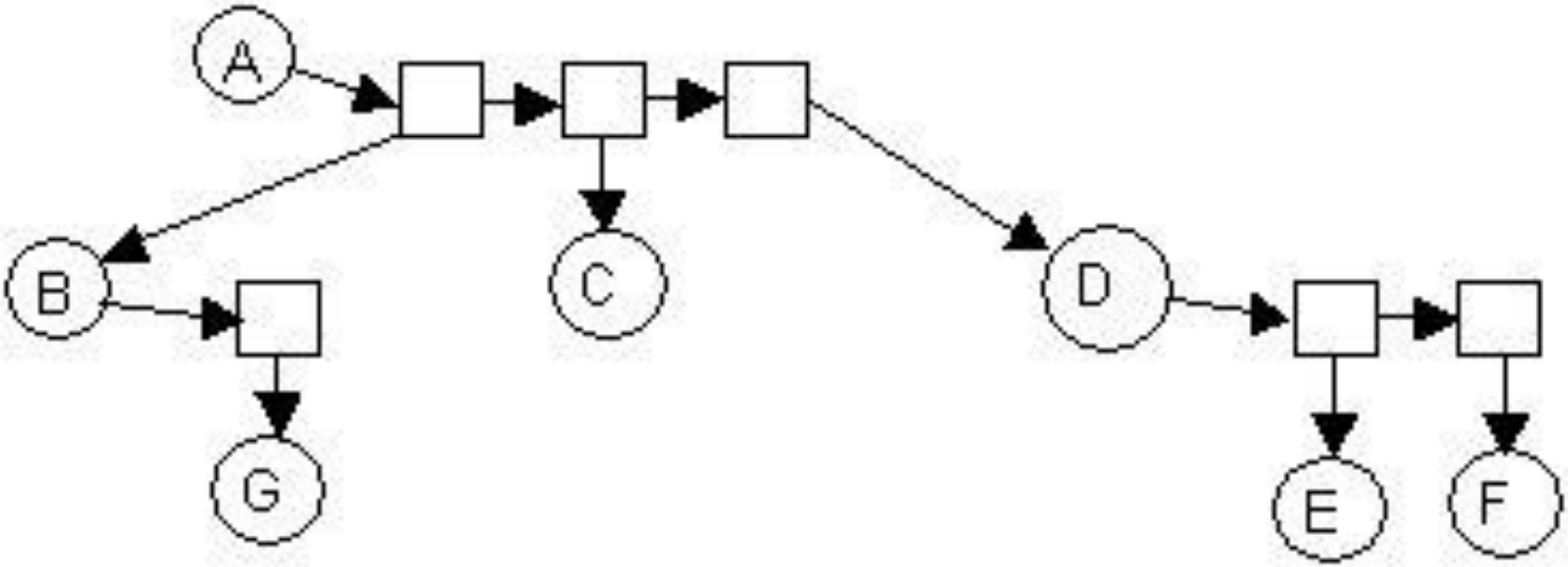
В некоторых узлах память оказывается недоиспользованной, а также возможна ситуация, когда число сыновей узла окажется больше максимально допустимого.

От этого недостатка свободно представление, когда указатели на сыновей находятся в линейном списке. Такая списочная структура содержит два типа узлов – узлы дерева и узлы линейного списка сыновей.

```
struct SON { // узел списка сыновей
    struct NODE *Son; // указатель на сына
    struct SON *Next; // указатель на следующий
    узел
                // списка сыновей
};

struct NODE { // узел дерева
    <поля данных>;
    SON *Sons; // указатель на начало списка
    сыновей
```

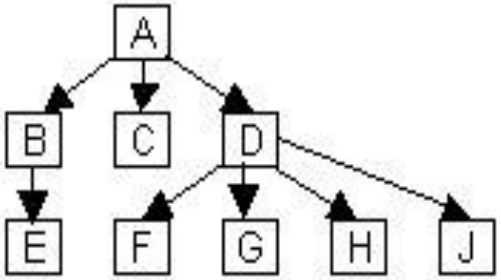
Пример такого дерева



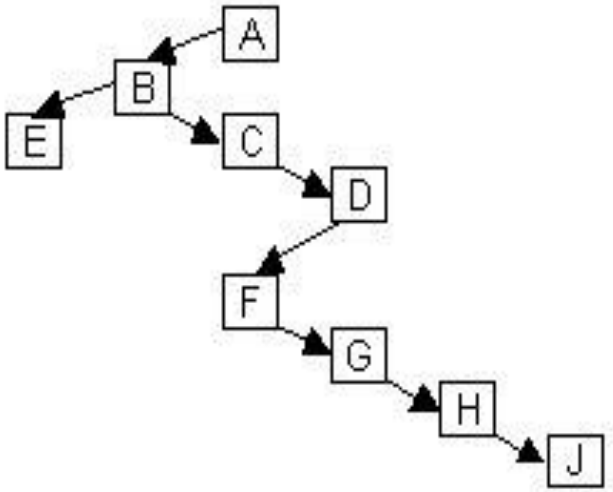
Представление деревьев общего вида бинарными деревьями

Всякое дерево можно представить в виде бинарного дерева. При преобразовании сохраняется связь отца с самым левым (старшим) сыном.

Они соединяются левой связью. Сыновья одного отца соединяются правыми связ э.



Дерево общего вида



Соответствующее бинарное дерево

В дальнейшем деревья ещё будут рассматриваться в связи с их использованием для представления таблиц.