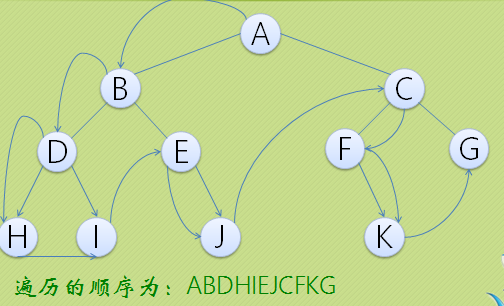
# 46二叉树的遍历

## 二叉树的遍历

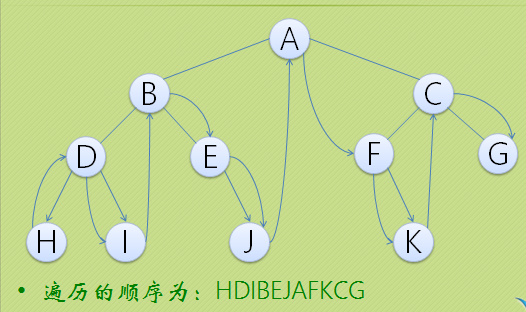
* 二叉树的遍历(traversing binary tree)是指从根结点出发，按照某种次序依次访问二叉树中所有结点，使得每个结点被访问一次且仅被访问一次。
* 这里有两个关键词小甲鱼给加红了：次序和访问
* 二叉树的遍历次序不同于线性结构，线性结构最多也就是分为顺序、循环、双向等简单的遍历方式。
* 树的结点之间不存在唯一的前驱和后继这样的关系，在访问一个结点后，下一个被访问的结点面临着不同的选择。这就像我们的人生，漫漫长途上一步踏错，满盘皆输！

## 二叉树的遍历方法

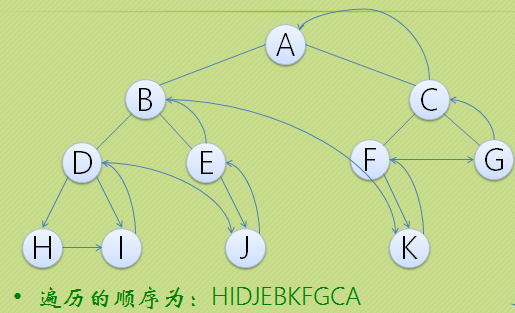
* 二叉树的遍历方式可以很多，如果我们限制了从左到右的习惯方式，那么主要就分为以下四种：
  + 前序遍历
  + 中序遍历
  + 后序遍历
  + 层序遍历
* 前序遍历：
  + 若二叉树为空，则空操作返回，否则先访问根结点，然后前序遍历左子树，再前序遍历右子树。



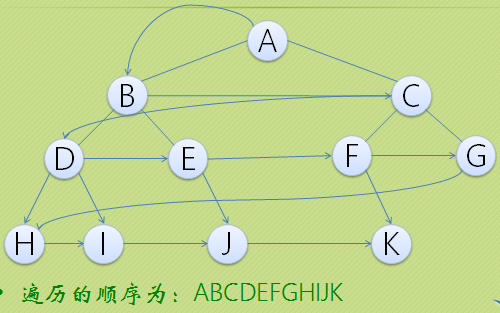
* 中序遍历：
  + 若树为空，则空操作返回，否则从根结点开始（注意并不是先访问根结点），中序遍历根结点的左子树，然后是访问根结点，最后中序遍历右子树。



* 后序遍历：
  + 若树为空，则空操作返回，否则从左到右先叶子后结点的方式遍历访问左右子树，最后访问根结点。



* 层序遍历：
  + 若树为空，则空操作返回，否则从树的第一层，也就是根结点开始访问，从上而下逐层遍历，在同一层中，按从左到右的顺序对结点逐个访问。



* 有童鞋会说，研究这么多遍历的方法干啥呢？
* 聪明的鱼油们怎么看？！

# 47二叉树的建立和遍历算法

## 二叉树的建立和遍历算法

有童鞋会说，我们上节课研究这么多遍历的方法干啥呢？聪明的鱼油们怎么看？！

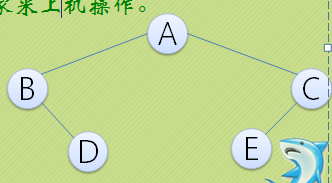
对于二叉树，思路方面我们已经谈得够多了，是时候由小甲鱼带大家来上机操作

题目要求：建立二叉树并输出每个字符所在的层数。如右图要求输出

A在第一层

B、C在第二层

D、E在第三层



前序接收输入并建立树，所以输入为：

A B 空格 D 空格 空格 C E 空格 空格 空格

#include <stdio.h>

#include <stdlib.h>

typedef char ElemType;

typedef struct BiTNode

{

ElemType data;

struct BiTNode \*lchild, \*rchild;

} BiTNode, \*BiTree;

// 创建一棵二叉树，约定用户遵照前序遍历的方式输入数据

CreateBiTree(BiTree \*T)

{

char c;

scanf("%c", &c);

if( ' ' == c )

{

\*T = NULL;

}

else

{

\*T = (BiTNode \*)malloc(sizeof(BiTNode));

(\*T)->data = c;

CreateBiTree(&(\*T)->lchild);//递归

CreateBiTree(&(\*T)->rchild);//递归

}

}

// 访问二叉树结点的具体操作，你想干嘛？！

visit(char c, int level)

{

printf("%c 位于第 %d 层\n", c, level);

}

// 前序遍历二叉树

PreOrderTraverse(BiTree T, int level)

{

if( T )

{

visit(T->data, level);

PreOrderTraverse(T->lchild, level+1);

PreOrderTraverse(T->rchild, level+1);

}

}

int main()

{

int level = 1;

BiTree T = NULL;

CreateBiTree(&T);

PreOrderTraverse(T, level);

return 0;

}

# 48线索二叉树

## 线索二叉树

为什么需要线索二叉树呢？

我想正如程序猿发觉单链表并不总能满足他们设计的程序某些要求的时候，发明了双向链表来弥补一样，线索二叉树也是在需求中被创造的！

那普通的二叉树到底有什么缺陷让我们发指呢？

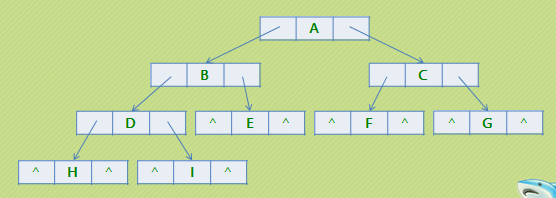
一，浪费空间 （指针空间）

二，浪费时间

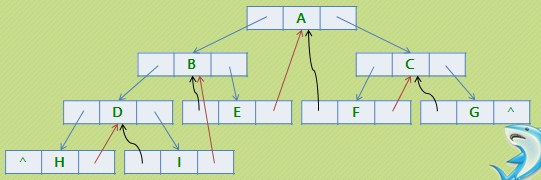
三，浪费青春

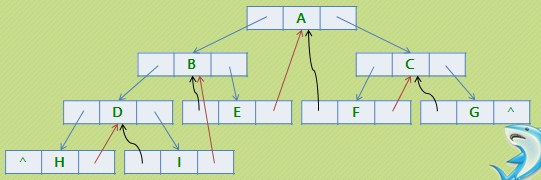
来，我们具体分析下如何个浪费法：

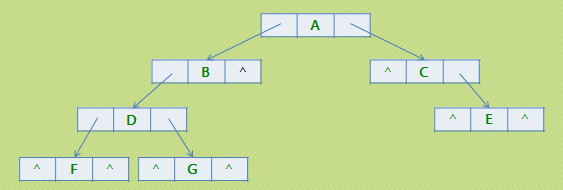
* 数序题：请问以下有多少个“^”？总共浪费了多少字节的空间？(32bit的机器,32位4个字节，指针占用4个字节)



* 脑筋急转弯：我们知道通过对二叉树的约定遍历方式，可以得到一个固定的遍历顺序，那么请问哪种遍历方式可以节省“^”所浪费的空间？（利用“^”记录该结点的前驱后继）







* 上图经过中序遍历后结果是：FDGBACE
* 黄色说明只有一个空闲的指针位置，如果是这样的话我们就面临一个问题：机器怎么识别到底是存放指针还是线索？
* 没错，她需要一丁点儿提示，为此我们将已经定义好的结构进行“扩容”：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **lchild** | **ltag** | **data** | **rtag** | **rchild** |

* + ltag为0时指向该结点的左孩子，为1时指向该结点的前驱（me：为了减少空间的浪费所有才没有另外的定义结构的新的指针数据域）。
  + rtag为0时指向该结点的右孩子，为1时指向该结点的后继。

让我们一起来敲代码吧！

???????????????????????//========201709234

# 49线索二叉树代码实现

BiThrTree.c

#include <stdio.h>

#include <stdlib.h>

typedef char ElemType;

// 线索存储标志位

// Link(0)：表示指向左右孩子的指针

// Thread(1)：表示指向前驱后继的线索

typedef enum {Link, Thread} PointerTag;

typedef struct BiThrNode

{

char data;

struct BiThrNode \*lchild, \*rchild;

PointerTag ltag;

PointerTag rtag;

} BiThrNode, \*BiThrTree;

// 全局变量，始终指向刚刚访问过的结点

BiThrTree pre;

// 创建一棵二叉树，约定用户遵照前序遍历的方式输入数据

void CreateBiThrTree( BiThrTree \*T )

{

char c;

scanf("%c", &c);

if( ' ' == c )

{

\*T = NULL;

}

else

{

\*T = (BiThrNode \*)malloc(sizeof(BiThrNode));

(\*T)->data = c;

(\*T)->ltag = Link;

(\*T)->rtag = Link;

CreateBiThrTree(&(\*T)->lchild);

CreateBiThrTree(&(\*T)->rchild);

}

}

// 中序遍历线索化

void InThreading(BiThrTree T)

{

if( T )

{

InThreading( T->lchild ); // 递归左孩子线索化

if( !T->lchild ) // 如果该结点没有左孩子，设置ltag为Thread，并把lchild指向刚刚访问的结点。

{

T->ltag = Thread;

T->lchild = pre;

}

if( !pre->rchild )

{

pre->rtag = Thread;

pre->rchild = T;

}

pre = T;

InThreading( T->rchild ); // 递归右孩子线索化

}

}

void InOrderThreading( BiThrTree \*p, BiThrTree T )

{

\*p = (BiThrTree)malloc(sizeof(BiThrNode));

(\*p)->ltag = Link;

(\*p)->rtag = Thread;

(\*p)->rchild = \*p;

if( !T )

{

(\*p)->lchild = \*p;

}

else

{

(\*p)->lchild = T;

pre = \*p;

InThreading(T);

pre->rchild = \*p;

pre->rtag = Thread;

(\*p)->rchild = pre;

}

}

void visit( char c )

{

printf("%c", c);

}

// 中序遍历二叉树，非递归

void InOrderTraverse( BiThrTree T )

{

BiThrTree p;

p = T->lchild;

while( p != T )

{

while( p->ltag == Link )

{

p = p->lchild;

}

visit(p->data);

while( p->rtag == Thread && p->rchild != T )

{

p = p->rchild;

visit(p->data);

}

p = p->rchild;

}

}

int main()

{

BiThrTree P, T = NULL;

CreateBiThrTree( &T );

InOrderThreading( &P, T );

printf("中序遍历输出结果为: ");

InOrderTraverse( P );

printf("\n");

return 0;

}

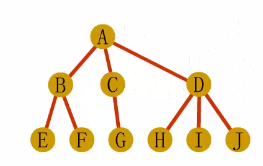
# 50树、森林及二叉树的相互转换

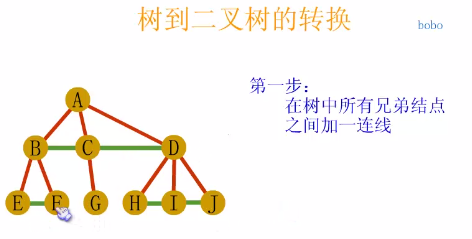
## 树、森林及二叉树的相互转换

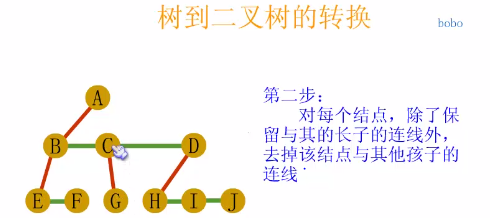
* 从一个屌丝逆袭高富帅的小故事说起。
* 在这一章节开始的时候我们是从一棵普通的树开始介绍，在满足树的条件下可以是任意形状，一个结点可以有任意多个孩子，这样对树的处理显然要复杂很多。
* 所以我们研究出了一些条条框框的限定，如：二叉树，完全二叉树，满二叉树等。
* 那么这时候你就会想，如果所有的树都像二叉树一样方便处理就好了。

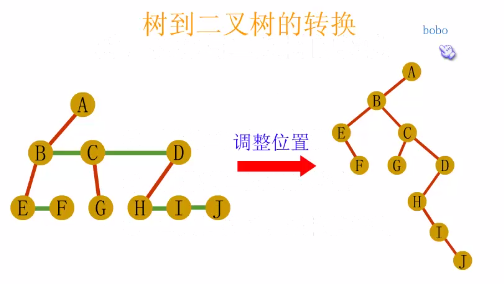
## 普通树转换为二叉树

* 步骤如下：
  + 加线，在所有兄弟结点之间加一条连线。
  + 去线，对树中每个结点，只保留它与第一孩子结点的连线，删除它与其他孩子结点之间的连线。
  + 层次调整，以树的根结点为轴心，将整棵树顺时针旋转一定的角度，使之结构层次分明。



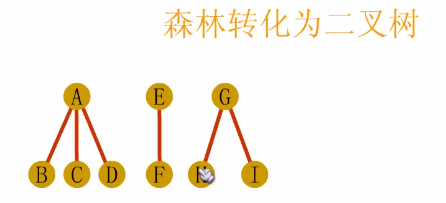


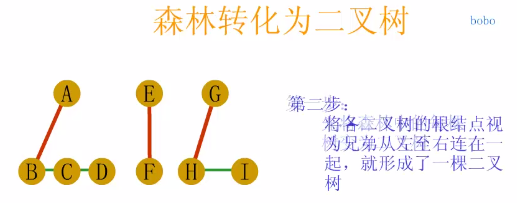


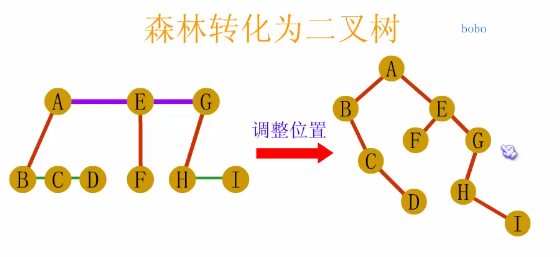


## 森林转换为二叉树

* 步骤如下：
  + 把每棵树转换为二叉树。
  + 第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树的根结点的右孩子，用线连接起来。

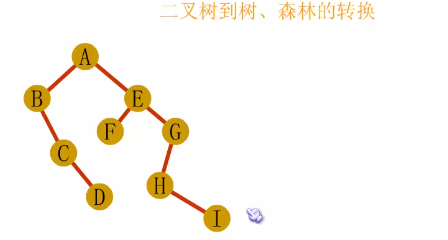


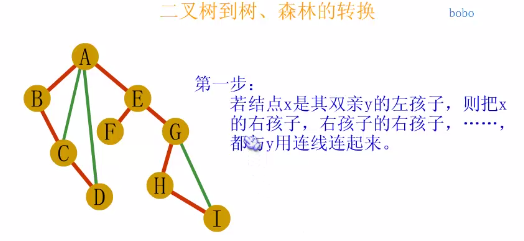


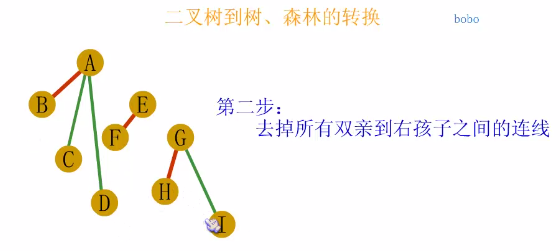


## 二叉树转换为树、森林

* 二叉树转换为普通树是刚才的逆过程，步骤也就是反过来做而已。
* 判断一棵二叉树能够转换成一棵树还是森林，标准很简单，那就是只要看这棵二叉树的根结点有没有右孩子，有的话就是森林，没有的话就是一棵树。







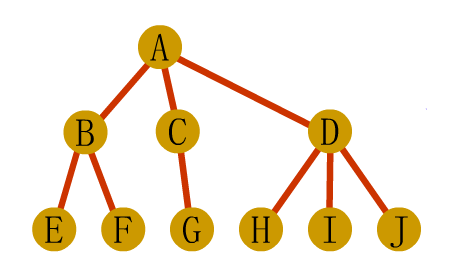
## 树与森林的遍历

树的遍历分为两种方式：一种是先根遍历（me：对应二叉树的前序遍历），另一种是后根遍历（me：对应二叉树的中序遍历）。

先根遍历：先访问树的根结点，然后再依次先根遍历根的每棵子树。

后根遍历：先依次遍历每棵子树，然后再访问根结点。

No pic you say a J8!



* 先根遍历结果：ABEFCGDHIJ
* 后根遍历结果：EFBGCHIJDA
* 森林的遍历也分为前序遍历和后序遍历，其实就是按照树的先根遍历和后根遍历依次访问森林的每一棵树。
* 我们的惊人发现：树、森林的前根（序）遍历和二叉树的前序遍历结果相同，树、森林的后根（序）遍历和二叉树的中序遍历结果相同！
* 这其实也就证实我们视频开头讲解的那个例子，我们找到了对树和森林遍历这种复杂问题的简单解决方案！