# 二叉树基本算法

先序创建二叉树

void Create(BiTree \*T) {

char ch;

scanf("%c", &ch);

if (ch == '#')

\*T = NULL;

else {

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

if (\*T == NULL) exit(-1);

(\*T)->data = ch;

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

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

}

}

遍历：对所有结点访问仅且访问一次。访问的含义很广，可以是打印，插入，查找等各种处理。

先序遍历：根结点，左子树，右子树。

中序遍历：左子树，右子树，根结点。

后序遍历：左子树，右子树，根结点。

先序遍历序列特点：每个结点的孩子结点都在其右边。

中序遍历序列特点：每个结点的左子树结点都在其左侧，右子树结点都在其右侧。

后序遍历序列特点：每个结点的孩子结点都在其左侧。

以上结论主要用于根据中序和先/后序遍历序列，还原一棵二叉树。

递归遍历复杂度分析

时间复杂度:O(n)\*\*每个结点访问一次\*\*

空间复杂度:辅助单元是树深+1.\*\*思考非递归遍历过程的模拟栈，即可得出此结论\*\*

先/中/后序遍历的visit(T)可以改变，实现按三种遍历次序对每个结点进行各种处理。

递归先序遍历二叉树

void DLR(BiTree T) {

if (T) {

visit(T);//可变操作，处理每个结点

DLR(T->lchild);

DLR(T->rchild);

}

}

递归中序遍历二叉树

void LDR(BiTree T)

{

if (T)

{

LDR(T->lchild);

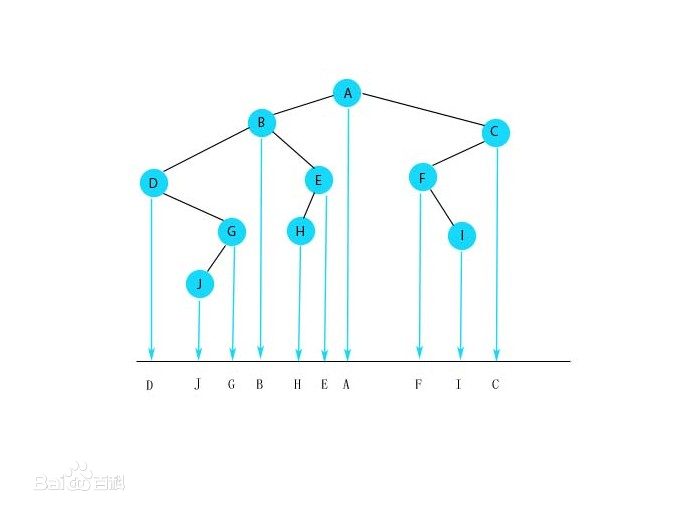
visit(T); //可变操作，处理每个结点

LDR(T->rchild);

}

}

特别说明中序遍历



由图可得以下结论：

一棵二叉树，根结点是T，则有

（1）中序遍历访问的第一个结点是T->lchild->lchild...即最左侧结点，该结点一定没有左孩子。

（2）中序遍历访问的最后一个结点是T->rchild->rchild...即最右侧结点，该结点一定没有右孩子。

（3）对树T的任一结点p，它的中序遍历前驱是以p为根结点的左子树中序遍历序列的最后一个结点，由（2）得，前驱是p->lchild->rchild->rchild->...。它的后继是以p为根结点的右子树中序遍历的第一个结点，由（1）得，后继是p->rchild->lchild->lchild->...。

以上结论主要用于线索二叉树。

二叉树非递归先/中序遍历

入栈时操作是先序，出栈时操作是中序。

void InOrderTravel(BiTree T) {

BiTree stack[100], p;

int top = -1;

if (T == NULL)

return;

stack[++top] = T;

while (top != -1) {

p = stack[top]->lchild;

while (p->lchild != NULL) {

stack[++top] = p;

p = p->lchild;

}//从根结点出发，一路向左到尽头

while (top != -1 && p->rchild == NULL) {

printf("%d ", p->data);

--top;

p = stack[top];

}//弹出栈中结点一直到碰到有右孩子的结点结束，把右孩子入栈

if (top == -1)

return;

--top;

printf("%d ", p->data);

stack[++top] = p->rchild;

}

}

由遍历序列恢复二叉树

从前面讨论的二叉树的遍历知道，任意一棵二叉树结点的先序序列和中序序列都是唯一的。反过来，若已知结点的先序序列和中序序列，能否确定这棵二叉树呢？这样确定的二叉树是否是唯一的呢？回答是肯定的。

根据定义，二叉树的先序遍历是先访问根结点，其次再按先序遍历方式遍历根结点的左子树，最后按先序遍历方式遍历根结点的右子树。这就是说，在先序序列中，第一个结点一定是二叉树的根结点。另一方面，中序遍历是先遍历左子树，然后访问根结点，最后再遍历右子树。这样，根结点在中序序列中必然将中序序列分割成两个子序列，前一个子序列是根结点的左子树的中序序列，而后一个子序列是根结点的右子树的中序序列。根据这两个子序列，在先序序列中找到对应的左子序列和右子序列。在先序序列中，左子序列的第一个结点是左子树的根结点，右子序列的第一个结点是右子树的根结点。这样，就确定了二叉树的三个结点。同时，左子树和右子树的根结点又可以分别把左子序列和右子序列划分成两个子序列，如此递归下去，当取尽先序序列中的结点时，便可以得到一棵二叉树。

同样的道理，由二叉树的后序序列和中序序列也可唯一地确定一棵二叉树。因为，依据后序遍历和中序遍历的定义，后序序列的最后一个结点，就如同先序序列的第一个结点一样，可将中序序列分成两个子序列，分别为这个结点的左子树的中序序列和右子树的中序序列，再拿出后序序列的倒数第二个结点，并继续分割中序序列，如此递归下去，当倒着取取尽后序序列中的结点时，便可以得到一棵二叉树。

下面通过一个例子，来给出右二叉树的先序序列和中序序列构造唯一的一棵二叉树的实现算法。

已知一棵二叉树的先序序列与中序序列分别为：

A B C D E F G H I

B C A E D G H F I

试恢复该二叉树。

首先，由先序序列可知，结点A是二叉树的根结点。其次，根据中序序列，在A之前的所有结点都是根结点左子树的结点，在A之后的所有结点都是根结点右子树的结点，由此得到图6.10 (a)所示的状态。然后，再对左子树进行分解，得知B是左子树的根结点，又从中序序列知道，B的左子树为空，B的右子树只有一个结点C。接着对A的右子树进行分解，得知A的右子树的根结点为D；而结点D把其余结点分成两部分，即左子树为E，右子树为F、G、H、I，如图6.10 (b)所示。接下去的工作就是按上述原则对D的右子树继续分解下去，最后得到如图6.10 (c)的整棵二叉树。

A

A

A

D

B

D

B

F

C

E

C

E

I

G

H

(a) (b) (c)

图6.10 一棵二叉树的恢复过程示意

上述过程是一个递归过程，其递归算法的思想是：先根据先序序列的第一个元素建立根结点；然后在中序序列中找到该元素，确定根结点的左、右子树的中序序列；再在先序序列中确定左、右子树的先序序列；最后由左子树的先序序列与中序序列建立左子树，由右子树的先序序列与中序序列建立右子树。

下面给出用C语言描述的该算法。假设二叉树的先序序列和中序序列分别存放在一维数组preod[ ]与inod[ ]中，并假设二叉树各结点的数据值均不相同。

void ReBiTree（char preod[ ],char inod[ ],int n,BiTree root）

/\*n为二叉树的结点个数，root为二叉树根结点的存储地址\*/

{ if (n≤0) root=NULL;

else PreInOd(preod,inod,1,n,1,n,&root);

}

算法 6.11

void PreInOd（char preod[ ],char inod[ ],int i,j,k,h,BiTree \*t）

{\* t=(BiTNode \*)malloc(sizeof(BiTNode));

\*t->data=preod[i];

m=k;

while (inod[m]!=preod[i]) m++;

if (m==k) \*t->lchild=NULL

else PreInOd(preod,inod,i+1,i+m-k,k,m-1,&t->lchild);

if (m==h) \*t->rchild=NULL

else PreInOd(preod,inod,i+m-k+1,j,m+1,h,&t->rchild);

}

算法 6.12

需要说明的是，数组preod和inod的元素类型可根据实际需要来设定，这里设为字符型。另外，如果只知道二叉树的先序序列和后序序列，则不能唯一地确定一棵二叉树。

根据先序和中序遍历序列还原二叉树源代码

BiTree Recover(int pre[], int in[], int length) {

int i;

if (length == 0)

return NULL;

BiTree root = (BiTree)malloc(sizeof(BiTNode));

if (!root) exit(-1);

root->data = pre[0];

for (i = 0; i <= length - 1; ++i) {

if (in[i] == pre[0])

break;

}

root->lchild = Recover(pre + 1, in, i);

root->rchild = Recover(pre + i + 1, in + i + 1, length - i - 1);

return root;

}

中序线索化二叉树

/\*

线索二叉树前言：

二叉树有性质，(1)n0=n2+1,一棵有n个结点的二叉树，(2)n = n0+n1+n2,

由（1）（2）得，2n0+n1= n+1，所以空指针数目是n+1.

用这n+1个指针，存放相应结点在某种遍历的前驱和后继，在以后的遍历中，

可以加快遍历速度，而且还未占用额外的空间。

如果一个结点的左指针是空，则放该结点的遍历前驱，如果一个结点的

右指针为空，则放该结点的遍历后继。但是这样有个副作用，无法知晓，

左右指针域放的是孩子，还是某种遍历的前驱后继。所以设置个标志位，

通过标志位，判断放的是谁。

步骤是，先把二叉树创建好，然后，按照某种遍历方法，操作每一个结点，

操作结点的顺序，刚好是遍历顺序，可以设置一个指针变量pre，始终保存，

上一个被处理的结点。--->>看中序线索二叉树的代码。

处理完毕的结果是，遍历序列的最后一个结点的左指针判断了，但是右指针没进行判断。

但是根据二叉树中序遍历的特点，我们可以“明知”右指针一定是个空指针。

把一棵二叉树线索化后，增设一个头结点，头结点的左孩子指向树根，头结点

的右孩子指向遍历序列的最后一个结点。这样可以达到，从任何一个结点都可以快速

的遍历整个二叉树。整体来看，头结点的左右就是找到中序遍历的最后一个结点，

与整个原线索二叉树无关。

如何找中序线索化二叉树某个结点p的后继？

如果右标志是Thread，说明右指针本来是NULL，现存放的是后继，而不是右孩子。

如果右指针存放的不是后继，那么该结点的后继是，以该结点右孩子为根结点的二叉树

中序遍历的第一个结点，即从p->rchild出发最左结点。

注意：最后一个元素没有后继结点！

如何找中序线索化二叉树某个结点p的前驱？

方法同找后继。

如何遍历中序线索化二叉树？

1，找到从树根出发的最左结点，该结点是中序遍历的第一个结点。

2，from 最左结点 to 中序遍历的最后一个结点（p->rchild==NULL）

注释：它是整个二叉树唯一一个空指针域！

3，不断调用求后继函数，以求后继函数做扫描指针。

\*/

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

typedef enum { Link, Thread } PointerTag;

typedef struct node {

char data;

struct node \*lchild, \*rchild;

PointerTag ltag, rtag;

}BiThrNode,\*BiThrTree;

BiThrTree pre = NULL;

/\*创建二叉树\*/

void Create(BiThrTree \*T) {

char ch;

scanf\_s("%c", &ch);

if (ch == '#')

(\*T) = NULL;

else {

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

if (!\*T) exit(-1);

(\*T)->data = ch;

(\*T)->ltag = Link;

(\*T)->rtag = Link;

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

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

}

}

/\*中序遍历二叉树\*/

void InOrder(BiThrTree T) {

if (T) {

InOrder(T->lchild);

printf("%c ", T->data);

InOrder(T->rchild);

}

}

/\*中序线索化一棵二叉树\*/

void InThread(BiThrTree T) {

if (T) {

InThread(T->lchild);

if (T->lchild == NULL) {

T->ltag = Thread;

T->lchild = pre;

}

/\*下面是一行very 重要的代码\*/

if (pre != NULL && pre->rchild == NULL) {

pre->rtag = Thread;

pre->rchild = T;

}

pre = T;

InThread(T->rchild);

}

}

/\*给线索化后的二叉树加上头结点\*/

void HeadInThread(BiThrTree \*head, BiThrTree T) {

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

if (!\*head) exit(-1);

if (T == NULL) {

\*head = NULL;

return;

}

InThread(T);

(\*head)->lchild = T;//指向根结点

(\*head)->rchild = pre;//指向中序遍历的最后一个结点

}

/\*求线索二叉树的中序遍历的后继结点\*/

BiThrTree LDRNext(BiThrTree p) {

if (p == NULL || p->rchild == NULL)

return NULL;

if (p->rtag == Thread)

return p->rchild;

else {

p = p->rchild;

while (p->ltag == Link) {

p = p->lchild;

}

return p;

}

}

/\*求线索二叉树的中序遍历的前驱结点\*/

BiThrTree LDRPre(BiThrTree p) {

if (p == NULL)//其实应该加个p不是中序遍历的第一个结点

return NULL;

if (p->ltag == Thread)

return p->lchild;

else {

p = p->lchild;

while (p->rtag == Link)

p = p->rchild;

}

}

/\*中序遍历线索二叉树\*/

void LDR(BiThrTree head) {

if (head == NULL)

return;

BiThrTree p = head;

while (p->lchild) {

p = p->lchild;

}

while (p) {

printf("%c ", p->data);

p = LDRNext(p);

}

printf("\n");

}

非递归后序遍历二叉树

算法流程：

本算法，每个结点只入栈一次，出栈一次。通过标志位确定何时出栈。出栈时即是访问时机。

每次入栈，标志位都先置为0。

从根结点出发，一路向左入栈。结束第一轮操作，以后操作的数据结构是栈，与本轮再无关系。

当栈不空时，

取栈顶元素，（这时还未出栈，判断后再决定是否出栈），如果栈顶元素的没有右孩子或者标志位是1，出栈。如果栈顶元素有右孩子且标志位是0，则（栈顶元素不栈）把栈顶的标志位置1，并把从栈顶元素的右孩子结点出发一路向左路径的结点全部入栈。结束。

核心步骤是，判断栈顶元素有无右孩子和标志位是0还是1，依据此条件执行两种不同的操作。

if(栈顶) 则 操作1；if(栈顶) 则 操作2.

/\*定义二叉树结点结构体\*/

typedef struct node {

char data;

struct node \*lchild, \*rchild;

}BiTNode, \*BiTree;

/\*定义栈中元素结构体\*/

typedef struct {

BiTree pTNode;//入栈结点指针

int flag;//入栈结点标志位

}StackElem;

void LRD(BiTree T) {

if (T == NULL)

return;

StackElem st;

StackElem stack[100];

int top = -1;

BiTree p = T;

while (p) {

top++;

stack[top].pTNode = p;

stack[top].flag = 0;

p = p->lchild;

}

while (top != -1) {//只要栈不空

st = stack[top];

if (((st.pTNode->rchild == NULL)) || st.flag == 1) {

/\*栈顶元素标志位是1或栈顶元素没有右孩子\*/

p = st.pTNode;

printf("%c ", p->data);//访问

top--;

}

else {

if (st.pTNode->rchild && st.flag == 0) {

stack[top].flag = 1;

/\*这行不能改成st.pTNode!!!!，tmd,这个bug调了5个小时，st是stack[top]拷贝，我们是要修改栈里面的

flag，而不是栈外st的flag。。。判断条件时，二者是等价的，写的时候，就

完全不一样了！！！！\*/

p = st.pTNode->rchild;//入栈右孩子及从右孩子出发的最左路径的所有结点

while (p) {

top++;

stack[top].pTNode = p;

stack[top].flag = 0;

p = p->lchild;

}

}

}

}

}

求树的深度

1,根是NULL，返回0

2，左子树和右子树的深度谁大+1返回

int DepthOfTree(BiTree T)

{

if (T == NULL)

return 0;

int i = tree\_depth(T->lchild);

int j = tree\_depth(T->rchild);

return i >= j ? i + 1 : j + 1;

}

层次遍历

流程：把根结点放进队列，队头结点出队，若出队结点的孩子结点存在，孩子结点入队。

继续出队执行相同操作，直到队列为空。

拓展：这样能实现所有结点按层次都入队和出队一次仅且一次，每次出队对结点执行操作。

出队一个结点的同时，入队0或1或2个结点。

任意含有n个结点的二叉树，辅助队列的最大容量是 n/2 + 2 即可。

Status LevelTravel(BiTree T)

{

TNode \*p;

TNode\* Queue[100];

int front = 0, rear = 0;

if (T == NULL) return error;

Queue[rear] = T;

rear = (rear + 1 + 100) % 100;

while (front != rear)

{

p = Queue[front];

printf("%d ", p->data);

front = (front + 1 + 100) % 100;

if (p->lchild != NULL)

{

Queue[rear] = p->lchild;

rear = (rear + 1 + 100) % 100;

}

if (p->rchild != NULL)

{

Queue[rear] = p->rchild;

rear = (rear + 1 + 100) % 100;

}

}

}

求结点e的双亲结点

流程：利用队列出口层次遍历所有结点，某一结点的孩子结点是e，则该结点是e的双亲结点.

求e的左兄弟

流程：找到e的双亲结点，

if（parents->lchild&&parents->rchild&&parent->rchild->data == e）

return parent->lchild; else return NULL;

/\*销毁一棵树\*/

void destory\_tree(BiTree \*T)

{

if (\*T == NULL) return;

else

{

if ((\*T)->lchild)

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

if ((\*T)->rchild)

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

free(\*T);

\*T = NULL;

}

}

删除一棵树

LR=0，删除p所指结点的左子树，LR=1，删除p所指结点的右子树

Status delete\_child(BiTree p, int LR)

{

if (p == NULL)

return error;

else

{

if (LR == 0)

destory\_tree(&p->lchild);

if (LR == 1)

destory\_tree(&p->rchild);

}

}

求两个结点的最低公共祖先

如果二叉树是搜索二叉树

如果两个结点一个大于根结点，一个小于根结点，此时根结点就是它俩的最低公共祖先。

如果两个结点都小于根结点，就在左子树查找最低公共祖先。

如果两个结点都大于根结点，就在右子树查找最低公共祖先。

Node\* GetAncestor(Node\* root, Node\* x1, Node\* x2)//1.该二叉树为搜索二叉树

{

assert(x1 && x2);

if (x1->\_data <= root->\_data && x2->\_data <= root->\_data)

{

return GetAncestor(root->\_left, x1, x2);//两个节都小于根节点，最近公共祖先在左子树中

}

else if (x1->\_data > root->\_data && x2->\_data > root->\_data)

{

return GetAncestor(root->\_right, x1, x2);//两个节都大于根节点，最近公共祖先在左子树中

}

else

return root; //一个在左子树，一个在右子树，找到公共祖先

}

一般二叉树

路径最好是放结点的地址而不是值，因为可能多个结点的值相等。

1. 找到从根到node1的路径，并存储在一个向量或数组中。
2. 找到从根到node2的路径，并存储在一个向量或数组中。
3. 遍历这两条路径，直到遇到一个不同的节点，则前面的那个即为最低公共祖先.

时间复杂度是O（n）

两个结点的最远距离

求出每个结点的左右子树深度，左右子树深度之和就是最远距离。

判断一棵树是完全二叉树

层次遍历二叉树，在出对口判断结点，找到的第一个不是度为2的结点后，剩下的结点全是叶子结点就是完全二叉树，否则不是完全二叉树。

判断二叉树是否对称

求一个二叉树的镜像

1，找出二叉树中最远结点的距离

2，判断一棵二叉树是否是平衡二叉树

两个程序的母本程序是求深度的递归程序。

该程序能按照后序遍历，计算所有结点的左右子树的深度关系，并存储在全部变量中。

二叉树非递归算法集锦

（1）求二叉树指定结点的路径？

非递归后序遍历二叉树，出站口判断出结点是指定结点，则栈内剩余元素就是路径，正序遍历栈是根到指定结点的路径，逆序遍历栈是指定结点到根的路径。

时间复杂度是O（n），空间复杂度取决于栈的最大容量，最大容量是二叉树的深度，即最长路径。

（2）求二叉树所有叶子结点到根的逆序？

非递归后序遍历二叉树，在出栈口，判断出是叶子结点，此时，遍历输出栈，这样，有多少叶子结点就会打印多少路径。

此题还可以转换成，打印所有度为1为2，或者结点值是奇数，偶数的所有结点的路径。

时间复杂度O(n),空间复杂度仍旧取决于栈最大容量，即二叉树的深度。

（3）求二叉树最长路径？

最长路径一定是某个叶子结点到根的路径，所以非递归后序遍历二叉树，碰到叶子结点，就统计一下此时路径长度，更新保存最长路径参数的变量值。

此题还可以用来求二叉树的深度，最长路径长度就是树的深度。

（4）求二叉树指定的两个节点的最低公共祖先？

求出根到两个结点的路径，从头比较两条路径，第一次出现的不相等的路径结点的上一个结点，就是最低公共祖先。

求路径时间复杂度是O（n），比较路径时间复杂度也是O（n），空间复杂度就是保存两条路径的空间，也是O（n）。

（5）输出路径结点的和为给定值的所有路径？

注释：这里的路径只是指叶子结点到根的路径，其他非叶子结点不考虑。

非递归后序遍历二叉树，出栈口碰到叶子结点就统计此时栈中所有元素的和sum，如果sum等于指定值，就输出路径。

统计sum时，要遍历栈，栈的最大容量是二叉树的深度，设为k，则结点总数目n是2^k，时间复杂度变成叶子结点的数目\*叶子结点的路径长 = k\*2^k,把k转换成n可以得到最坏时间复杂度是n\*log2(n).

修改一下程序：设置一个全局变量SUM = 0，每次进栈，就把进栈元素加到SUM，每次出栈就用SUM减去出栈元素，这样SUM始终保存的是栈内元素的和，出栈口统计叶子结点路径和时，直接读取SUM，而不必遍历栈，时间复杂度变成O(n)。

计算每个结点的左右子树之和

/\*可改装成计算每个结点的左右子树深度之和的最大值\*/

#include<stdio.h>

#include<stdlib.h>

typedef struct node {

struct node \*lchild, \*rchild;

char data;

}BiTNode, \*BiTree;

typedef struct {

BiTree pNode;

int flag;

}StackNode;

void Printf(char \*arry, int len) {

printf(" ");

for (int i = 0; i < len; ++i) {

printf("%c", arry[i]);

}

printf(" ");

}

void Create(BiTree \*T) {

char ch;

scanf("%c", &ch);

if (ch == '#')

\*T = NULL;

else {

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

if (\*T == NULL) exit(-1);

(\*T)->data = ch;

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

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

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*王者程序的上边界\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*计算每个结点的左右子树深度之和\*/

int TwoDepthSum(BiTree T, int \*distance,int \*top)

{

if (T == NULL) {

return 0;

}

int left = TwoDepthSum(T->lchild, distance,top);

int right = TwoDepthSum(T->rchild, distance,top);

distance[(\*top)++] = right + left;

return left > right ? left + 1 : right + 1;

}

/\*封装函数\*/

void AllTwoDepthSum(BiTree T) {

int distance[100];

int top = 0;

TwoDepthSum(T,distance,&top);

for(int i = 0;i <= top -1;++i)

{

printf("%d ",distance[i]);

}

printf("\n");

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*王者程序的下边界\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*abd##e#f##c##\*/

int main() {

BiTree T;

Create(&T);

int distance[100];

int top = 0;

int a;

a = TwoDepthSum(T,distance,&top);

printf("二叉树的深度：%d\n",a);

printf("所有结点左右子树深度之和:\n");

AllTwoDepthSum(T);

return 0;

}

判断是否是平衡二叉树

#include<stdio.h>

#include<stdlib.h>

typedef struct node {

struct node \*lchild, \*rchild;

char data;

}BiTNode, \*BiTree;

typedef struct {

BiTree pNode;

int flag;

}StackNode;

void Create(BiTree \*T) {

char ch;

scanf("%c", &ch);

if (ch == '#')

\*T = NULL;

else {

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

if (\*T == NULL) exit(-1);

(\*T)->data = ch;

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

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

}

}

/\*算法流程

递归计算二叉树的深度的时间复杂度是O（n）

先写出计算计算二叉树深度的代码。

后序遍历每个结点，判断左右深度的关系

设置全局变量标志位。

\*/

/\*判断二叉树是否是平衡树\*/

int Depth(BiTree T,bool \*status,bool \*flag) {

if(T == NULL)

return 0;

int left = Depth(T->lchild,status,flag);

int right = Depth(T->rchild,status,flag);

if(left - right > 1 || left - right < -1) {

\*status = false;//某个结点不是平衡树，记录下来

if(\*status == false)//可以用全局变量数组记录，然后遍历数组

\*flag = false;//有false成员就不是平衡树。但可以用一个变量记录，

}//但是由于要遍历所有结点，会覆盖status，所有又加一个全局变量flag.

return left > right ? left + 1 : right + 1;

}

bool IsPingHeng(BiTree T) {

if(T == NULL)

return true;

bool status = true;

bool flag = true;

Depth(T,&status,&flag);

return flag ? true : false;

}

int main() {

BiTree T;

Create(&T);

bool haha,hehe;

bool status;

status = IsPingHeng(T);

if(status)

printf("平衡树\n");

else

printf("不是平衡树\n");

printf("二叉树的深度是: %d\n",Depth(T,&haha,&hehe));

return 0;

}

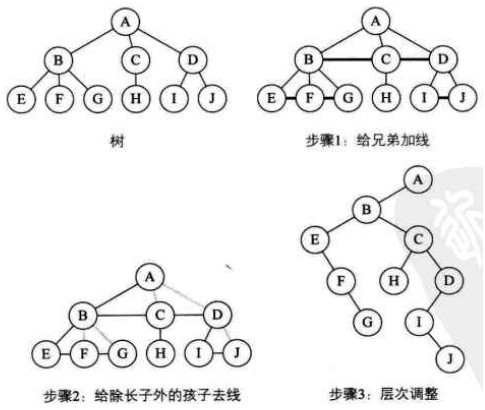
森林 树 二叉树的转换

**树转换为二叉树**

（1）加线。在所有兄弟结点之间加一条连线。

（2）去线。树中的每个结点，只保留它与第一个孩子结点的连线，删除它与其它孩子结点之间的连线。

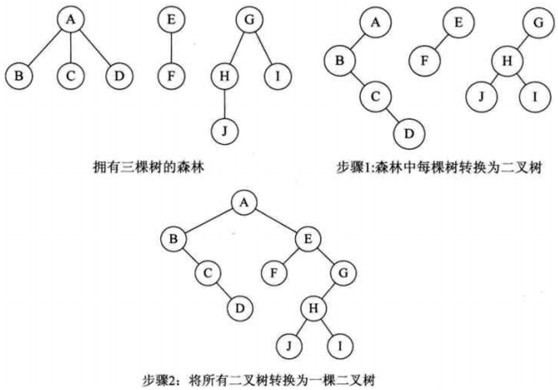
（3）层次调整。以树的根节点为轴心，将整棵树顺时针旋转一定角度，使之结构层次分明。（注意第一个孩子是结点的左孩子，兄弟转换过来的孩子是结点的右孩子）



**森林转换为二叉树**

（1）把每棵树转换为二叉树。

（2）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。



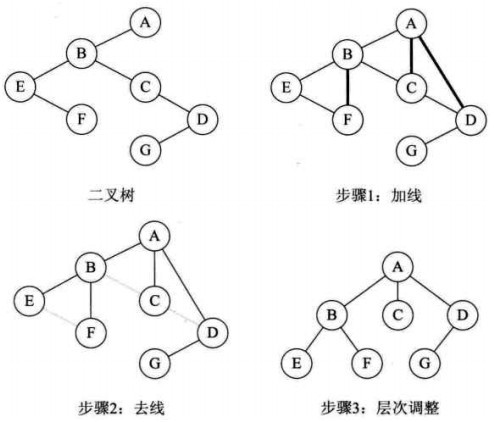
**二叉树转换为树**

是树转换为二叉树的逆过程。

（1）加线。若某结点X的左孩子结点存在，则将这个左孩子的右孩子结点、右孩子的右孩子结点、右孩子的右孩子的右孩子结点…，都作为结点X的孩子。将结点X与这些右孩子结点用线连接起来。

（2）去线。删除原二叉树中所有结点与其右孩子结点的连线。

（3）层次调整。

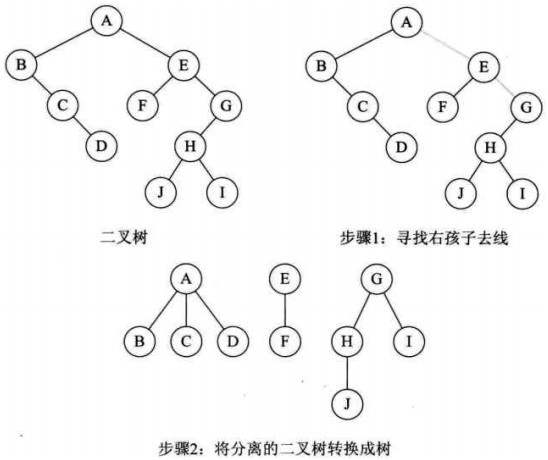


**二叉树转换为森林**

假如一棵二叉树的根节点有右孩子，则这棵二叉树能够转换为森林，否则将转换为一棵树。

（1）从根节点开始，若右孩子存在，则把与右孩子结点的连线删除。再查看分离后的二叉树，若其根节点的右孩子存在，则连线删除…。直到所有这些根节点与右孩子的连线都删除为止。

（2）将每棵分离后的二叉树转换为树。



完全二叉树

|  |  |  |
| --- | --- | --- |
|  | n是偶数 | n是奇数 |
| i是叶子结点（i）无孩子 | i>n/2 | i>n/2 |
| i有两个孩子 | i<n/2 | i<=n/2 |
| i有1个孩子（且是左孩子） | i=n/2 | 无 |
| i有左孩子 | i<=n/2 | i<=n/2 |
| i有右孩子 | i<n/2 | i<=n/2 |
| i有左兄弟 |  |  |
| i有右兄弟 |  |  |

结点总数是n，最后一个有孩子的结点编号是：n/2，非叶子结点一定是n/2,叶子结点的个数是n – n/2.

深度为h的二叉树，除h层，1 to h-1 各层结点数目都达到最大，h层的结点都连续集中在最左边。这样的二叉树叫完全二叉树。

任意一个结点的左子树高度是L，它的右子树一定是L或L-1.

1 to h-2层的所有结点度都是2，h层的所有结点度都为0，h-1层的结点度可能为0,1,2。

叶子结点只可能出现在h和h-1层，只可能有1个或没有度为1的结点，若有，该结点的左孩子是二叉树最后一个结点。

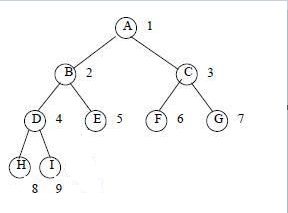
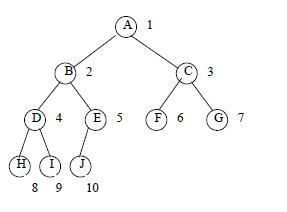
二叉树要么没有孩子，个数是n-n/2，要么有两个孩子，个数是n/2或n/2-1,要么只有一个左孩子，且这样的结点仅有一个，如果一个结点有右孩子，必有左孩子。

n是偶数，有度为1的结点，n是奇数无度为1的结点。

i是奇数，必是右孩子，左兄弟是i-1,

i是偶数，必是左孩子，若n是奇数，必有右兄弟，是i+1；若n是偶数且i!=n,必有右兄弟，是i+1。

完全二叉树的两种形态



n是偶数

n是奇数

具有n个结点的[完全二叉树](https://baike.baidu.com/item/%E5%AE%8C%E5%85%A8%E4%BA%8C%E5%8F%89%E6%A0%91)的深度为k = [log2 n] +1 (自己推一遍)

深度为h的完全二叉树最多有 https://gss2.bdstatic.com/-fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D38/sign=8cafdaa318d5ad6eaef962e280cb7442/279759ee3d6d55fbec4f97416e224f4a21a4ddc8.jpg 个结点(h>=1)， 最少有2^(h-1) – 1 + 1个结点；

二叉树结点的个数

现提供3种方法：

第一种方法，是递归的。

统计左子树结点，统计右子树结点，两棵子树结点数目相加再加根结点。

unsigned int CountNode(BiTree T) {

if (T == NULL)

return 0;

return CountNode(T->lchild) + CountNode(T->rchild) + 1;

}

第二种方法也是递归的。

设置一个全局变量count作为计数器，用count++代替递归遍历的printf行。

unsigned int count = 0;

void DLR(BiTree T) {

if (T) {

count++;//可变操作，处理每个结点

DLR(T->lchild);

DLR(T->rchild);

}

}

这种带有全局变量的函数调用的时候，需要使用全局变量，使用不便，可以改进，改进方法如下：

定义一个新函数，全局变量作为新函数的函数内局部变量，调用上述函数，向函数传入新函数的局部变量作为上述函数的全局变量参数，最后把局部变量返回给新函数，这样可以直接调用新函数，直接使用函数运行的结果。

unsigned int Count(BiTree T) {

unsigned int count = 0;

DLR(T);

return count;

}

第三种方法是非递归的。

层次遍历，在队头统计，出队一个统计一个，直到队空。代码略。

在二叉树中查找

查找二叉树中第一个值为value的结点,返回其地址，无值等于value的结点返回空

三种方法：

第一种方法是递归的。如果根结点等于value，返回根结点的地址，否则在左子树中查找，找到返回其地址，找不到在右子树找。

BiTree SearchValue(BiTree T,char value) {

if(T == NULL)

return NULL;

if(T->data == value)

return T;

if(SearchValue(T->lchild,value))

return SearchValue(T->lchild,value);

return SearchValue(T->rchild,value);

}

第二种方法也是递归的。利用递归遍历设置全局变量。

BiTree p = NULL；

void DLR(BiTree T) {

if (T) {

if (T->data == 'c') {

p = T;

return;

}

DLR(T->lchild);

DLR(T->rchild);

}

}

第三种方法是非递归的。层次遍历，在队头出来一个结点比较一次。代码略。

二叉树的宽度

/\*定义栈内元素的数据类型\*/

typedef struct {

BiTree pNode;//二叉树结点的指针

int level;//二叉树结点所在的层

}NewBiTree;

/\*求二叉树最大宽度\*/

void MaxWidth(BiTree T) {

int levelOfMaxWidth = 1;//始终保存结点数目最多的层

int nodeCountOfLevel = 1;//始终保存结点数目最多的层的结点数目

int i = 1, j = 0;//统计当前层和当前层的结点数目

NewBiTree p;//保存出队元素

/\*创建队列\*/

NewBiTree Queue[300];

int front = 0;

int rear = 0;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (T == NULL) {

printf("空树！\n");

return;

}

/\*\*\*\*\*\*根结点入队\*\*\*\*\*\*\*/

Queue[rear].pNode = T;

Queue[rear].level = 1;

rear++;

/\*\*\*\*\*层次遍历二叉树\*\*\*\*\*\*/

while (!(front == rear)) {

p = Queue[front];

front++;

/\*如果当前出队结点和上一出队结点是同一层，数目累加\*/

if (p.level == i)

j++;

/\*如果不属于同一层，说明已经遍历完一层，比较该层结点数目

与最宽层数目，更新最宽层数据\*/

else {

if (j > nodeCountOfLevel) {

levelOfMaxWidth = i;

nodeCountOfLevel = j;

}

/\*统计下一层，数目初始化\*/

i = p.level;

j = 1;

}

/\*出队结点的左右孩子入队\*/

if (p.pNode->lchild) {

Queue[rear].pNode = p.pNode->lchild;

Queue[rear].level = p.level + 1;

rear++;

}

if (p.pNode->rchild) {

Queue[rear].pNode = p.pNode->rchild;

Queue[rear].level = p.level + 1;

rear++;

}

}

/\*最后一层更新最宽层数据\*/

if (j > levelOfMaxWidth) {

levelOfMaxWidth = i;

nodeCountOfLevel = j;

}

printf("结点最多的层是%d,结点的数目是%d.\n", levelOfMaxWidth,nodeCountOfLevel);

}

生成哈夫曼树

#include<stdio.h>

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<math.h>

#include<io.h>

typedef int Status;

#define ok 1

#define false 0

#define ok 1

#define error 0

typedef struct

{

unsigned int weight;

unsigned int parents;

unsigned int lchild;

unsigned int rchild;

}HTNode,\*HuffmanTree;

//选出权值最小的两个结点

void Select(HuffmanTree \*HT,int n,int \*s1,int \*s2)

{

int i,tmp;

for(i=1;i<=n;++i)

{

if((\*HT)[i].parents==0)

{

\*s1 = i;break;

}

}//s1,第一个符合要求的任意权值的结点

for(i++;i<=n;++i)

{

if((\*HT)[i].parents==0)

{

\*s2 = i;break;

}

}//s1,第二个符合要求的任意权值的结点

if((\*HT)[\*s1].weight>(\*HT)[\*s2].weight)

{

tmp = \*s1;

\*s1 = \*s2;

\*s2 = tmp;

}//保证s1存储的是最小权值的元素的索引

for(i++;i<=n;++i)

{

if((\*HT)[i].weight<=(\*HT)[\*s1].weight&&(\*HT)[i].parents==0)

{

\*s2 = \*s1;

\*s1 = i;

}

}

}//遍历寻找

//搭建哈夫曼树并生成哈夫曼码

Status HuffmanCoding(HuffmanTree \*HT,int \*w,int n)

{

int m = 2\*n - 1;

int i,j,s1,s2;

if(n<=1)

return error;

\*HT = (HuffmanTree)malloc(sizeof(HTNode)\*(m+1));

if(\*HT==NULL)

exit(-1);

for(i=1;i<=n;++i)

{

(\*HT)[i].weight = w[i];

(\*HT)[i].parents = 0;

(\*HT)[i].lchild = 0;

(\*HT)[i].rchild = 0;

}

for(;i<=m;++i)

{

(\*HT)[i].weight = 0;

(\*HT)[i].parents = 0;

(\*HT)[i].lchild = 0;

(\*HT)[i].rchild = 0;

}

puts("\n哈夫曼树的构造过程如下：");

printf("HT初态：\n结点 weight parents lchild rchild\n");

for(i=1;i<=n;++i)

{

printf("%2d%8d%8d%8d%8d\n",i,(\*HT)[i].weight,(\*HT)[i].parents,(\*HT)[i].lchild,(\*HT)[i].rchild);

}

for(i=n+1;i<=m;++i)

{

Select(HT,i-1,&s1,&s2);

(\*HT)[s1].parents = i;

(\*HT)[s2].parents = i;

(\*HT)[i].weight = (\*HT)[s1].weight+(\*HT)[s2].weight;

(\*HT)[i].lchild = s1;(\*HT)[i].rchild = s2;

printf("Select: s1 = %d s2 = %d 新生成的第%d个结点:\n",s1,s2,i);

printf("%d(w) %d(p) %d(l) %d(r)\n",(\*HT)[i].weight,(\*HT)[i].parents,(\*HT)[i].lchild,(\*HT)[i].rchild);

}

puts("哈夫曼树的最终状态如下：");

printf("结点 weight parents lchild rchild\n");

for(i=1;i<=m;++i)

{

printf("%2d%8d%8d%8d%8d\n",i,(\*HT)[i].weight,(\*HT)[i].parents,(\*HT)[i].lchild,(\*HT)[i].rchild);

}

}

int main()

{

int w[6] = {0,3,4,5,6};//n = 5

HuffmanTree HT;

HuffmanCoding(&HT,w,4);

system("pause");

return 0;

}

生成哈夫曼码

解哈夫曼码

二叉树的镜像

如何生成三叉链表

求叶子结点的个数

左子树叶子数加右子树叶子数或者用遍历统计。