数据结构与算法笔记

**单链表**

1.2012年966

已知一个带有表头结点的单链表，结点结构为（data,link）.假设该链表只给出了头指针list。在不改变链表的前提下，请设计一个尽可能高效的算法，查找出链表倒数第k(k为正整数)个位置上的结点。若查找成功，算法输出该结点的data域的值，并返回1；否则，只返回0。

/\*定义两个指针变量p和q,初始时均指向头结点的下一个结点。p指针

沿链表移动，当p指针移动到第k个结点时，q指针开始与p指针同步

移动；当p指针移动到表尾结点时，q指针所指元素为倒数第k个结点。\*/

typedef struct LNode{

int data;

struct LNode \*link;

}\*LinkList;

int Searchk(LinkList list, int k){

LinkList p, q;

int count = 0;

p = q = List→link;

while(p != NULL){

if(count<k)

count++;

else

q = q→link;

p = p→link;

}

if(count < 0)

return(0);

else{

print("%d", q→data);

return(1);

}

}

2.2015年 941

给定两个单链表（为简化，假设两个链表均不含有环）的头指针分别为head1和head2,请设计一个算法判断这两个单链表是否相交，如果相交，则返回第一个交点，要求算法的时间复杂度为O(length1+length2),其中length1和length2分别为两个链表的长度。

方法一.红皮书答案

/\*求两个链表的长度，若两个链表相交，则两链表从某一结点

开始往后必一致，所以长链表比短链表长的部分不可能存在交点，求

出两链表的差值n,长链表从第n+1个结点，短链表从第一个结点同时开

始遍历，若当前结点不是交点，两链表用来遍历的指针同时后移，依次

判断，直到找到交点，若遍历完两个链表仍无交点，则说明两链表不相交。\*/

int len(LinkList L){

int i = 0;

LNode \*p = L;

while(p!=NULL){

i++;

p = p→next;

}

return i;

}

LNode Search(LinkList head1, LinkList head2){

int length1 = len(head1);

int length2 = len(head2);

int n;

LNode \*q = head1,\*s = head2;

if(length1 > length2){

n = length1 - length2;

for(int j = 0; j < n; j++)

q = q→next;

}

if(length1 < length2){

n = length2 - length1;

for(int j = 0; j < n; j++)

s = s→next;

}

while(q != NULL && q→data != s→data){

q = q→next;

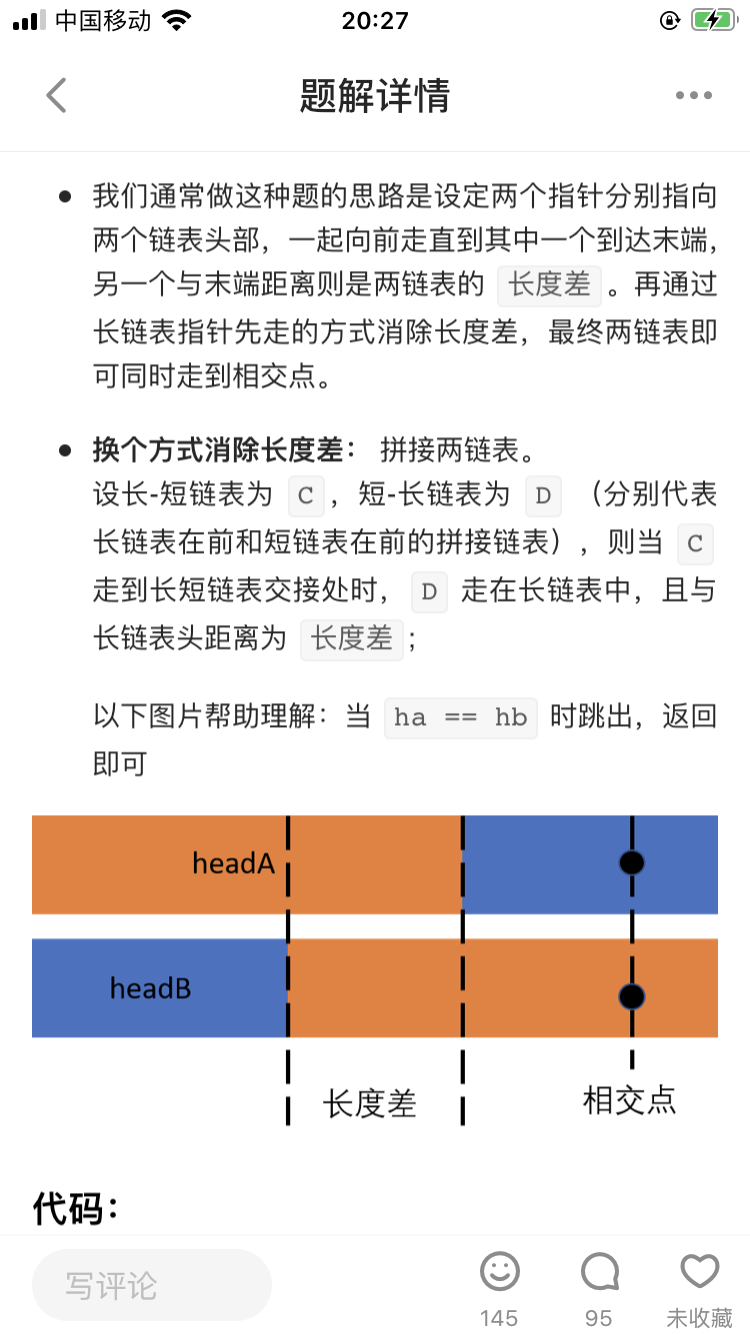
s = s→next;

}

return q;

}

（补充 双指针法）方法2. 拼接两链表。设长-短链表长度为C,短-长链表为D，则当C走到长短链表交接处时，D走在长链表中，且与长链表表头距离为高度差，当headA ==headB时跳出。**当已知相交时可以求交点，不相交不能用**



struct ListNode\*getIntersectionNode(struct ListNode \*headA, struct ListNode \*headB){

struct ListNode \*L1 = headA, \*L2 = headB;

while(L1 != L2){

L1 = ((L1 == NULL) ? headB : L1→next);

L2 = ((L2 == NULL) ? headA : L2→next);

}

return L1;

}

（补充 快满指针法）方法3.将L2的最后一个结点指向L2的头结点，构成环，判断L1链表是否有环，若有环则两个单链表相交，若无环则不相交，构环后需要解环。**只给出核心代码，判断是否有环代码，构环尾指针指向头结点即可。**

bool hasCycle(struct ListNode \*head){

if(NULL == head)

return false;

struct ListNode \*fast = head;

struct ListNode \*slow = head;

while(slow && fast && fast→next){

slow = slow→next;

fast = fast→next→next;

if (slow == fast)

return true;

}

return false;

}

3.2013.年966

类似算法

1.设计一个算法，在带头结点的单链表list中确定元素值最大的结点

2.设计一个算法，在带头结点的单链表list中删除一个具有最小值的结点，并从返回值中取得被删除结点的值。

3.设非空单链表list带有头结点，设计一个算法，将链表中具有最小值的结点移动到链表的最前面。

4.已知非空线性链表第一个结点的指针为list,写一个算法，找出链表中数据域最小的那个结点，并将其链接到链表的最前面。(不含头结点)

第一个

/\*算法思想,设置一个指针pmax,记忆当前找到的具有最大元素值的结点。

算法用指针p检查链表，如果找到值比所指示结点的元素值还要大的结点，让pmax记忆它。当链表所有结点都检测完，pmax所指示结点即为所求。\*/

LinkNode \*Max(LinkList &L){

if(L→next == NULL)

return NULL;

LinkList pmax = L→next, p;

for (p = pmax→next; p != NULL; P = P→next)

if(p→data > pmax→data)

pmax = p;

return pmax;

}

第二三个类似，区别的写在注释里了

/\*算法思想，在从单链表中删除一个结点时，为使结点删除后不出现"断链"，应知道被删除结点的前驱结点。而“最小元素值结点”是在遍历整个链表才知道的。\*/

bool Remove\_min(LinkList &L, DataType &x){

if(L→next = NULL)

return false;

LinkList p, pre = L, pmin = L→next;

for(p = L→next; p→next != NULL; p = p→next)

if(p→next→data < pmin→data){

pre = p; pmin = p→next;

}

pre→next = pmin→next;

x = pmin→data;free(pmin); //第二个

pmin→next = L.next; L.next = pmin; //第三个

return true;

}

第四个

void adjustLinkList(LNode \*\*list)

{

LNode \*min\_node = \*list, \*min\_node\_pre = NULL, \*p = \*list;

LNode\*pre; //找最小值结点的前驱

while(p !=NULL){

pre = p;//pre为p的前驱

p = p→next;

if(min\_node→data > p→data){

min\_node = p;

min\_node\_pre = pre; //最小值结点的前驱

}

}

if(min\_node == list)

return;

min\_node\_pre→next = min\_node→next;

min\_node→next = \*list;

\*list = min\_node;

}

4. 2018. 979（类似2001年p指针指向结点与后继交换）

在Head为头指针的单链表中查找结点DATA域为k的结点，并将该结点与其前驱结点交换位置。

红皮书答案

/\*算法思想：设置两个指针pre和p,让p指向Head的下一个结点的下一个结点，而pre指向Head,若p非空，则p和pre同时后移，当p找到值域为k的结点，先令pre的next的next指向p的next;再令p的next指向pre的next,最后令pre的next指向p,这就完成了要求。\*/

bool Exchange(Linklist Head){

LNode \*p = Head→next, \*pre = head;

if(p != NULL && p→data ==x){

pre→next = p→next;

p→next = pre;

return true;

}

p = p→next;

while(p !=NULL){

if(p→data ==x){

pre→next→next = p→next;

p→next = pre→next;

pre→next = p;

return true;

}

else{

p = p→next;

pre = pre→next;

}

}

return false;

}

5. 2000.

单链表直接插入排序

待排序文件以单链表方式表示，指针变量FIRST指向表头结点，表中结点结构为KEY LINK,其中key为结点关键词域，LINK为链接域，试写出该线性表的直接插入排序算法INSERT2(FIRST),要求算法是稳定的，并说明算法的时间复杂度。

struct ListNode\* insertionSortList(struct ListNode\* head){

if(head == NULL ||head -> next == NULL)

return head;

struct ListNode\* dummyhead = (struct ListNode\*)malloc(sizeof(struct ListNode));

dummyhead -> next = head;

struct ListNode \*p = head -> next, \*pre = dummyhead;

head -> next = NULL;

while(p != NULL)

{

struct ListNode\* temp = p -> next; //保存p的后继

while(pre -> next != NULL && pre -> next -> val < p -> val)

pre = pre -> next;

p -> next = pre -> next;

pre -> next = p;

p = temp;

pre = dummyhead;

}

return dummyhead -> next;

}

6.2013.967

归并排序 （不带头结点，返回第一个结点）

算法思想：  
1.遍历一次链表求出链表的长度len，初始化虚拟头结点，其后继为head  
2.初始化步数index，从长度1开始，每次自增2倍，将链表划分为长度为index的子链表，对每两个index的子链表进行合并。  
3.结束返回虚拟头结点的后继

struct ListNode\* Merge(struct ListNode\* head1, struct ListNode\* head2)

{

struct ListNode\* dummyhead = (struct ListNode\*)malloc(sizeof(struct ListNode));

struct ListNode\* current = dummyhead;

while(head1 != NULL && head2 != NULL) //合并排序

{

if(head1 -> val < head2 -> val)

{

current -> next = head1;

current = head1;

head1 = head1 -> next;

}

else

{

current -> next = head2;

current = head2;

head2 = head2 -> next;

}

}

if(head1 != NULL) current -> next = head1;

else current -> next = head2;

return dummyhead -> next;

}

struct ListNode\* cuts(struct ListNode\* head, int index)

{

struct ListNode\* dummyhead = (struct ListNode\*)malloc(sizeof(struct ListNode));

dummyhead -> next = head;

struct ListNode\* current = dummyhead;

int i = 0;

while(current -> next != NULL && i < index) //向后遍历index个单位，若链表长度不足index，则将返回NULL

{

i++;

current = current -> next;

}

struct ListNode\* u = current -> next; //用于返回下一个链表的头指针

current -> next = NULL; //将当前长为index的链表断链

return u; //返回下一个链表的头指针

}

struct ListNode\* sortList(struct ListNode\* head){

int len = 0;

struct ListNode\* p = head;

while(p != NULL)

{

len++;

p = p -> next;

}

struct ListNode\* dummyhead = (struct ListNode\*)malloc(sizeof(struct ListNode));

dummyhead -> next = head;

for(int index = 1; index <= len; index \*= 2)

{

struct ListNode\* left = dummyhead -> next; //left指向第一个链表

struct ListNode\* tail = dummyhead; //tail指向第一个链表的前一个结点

while(left != NULL)

{

struct ListNode\* right = cuts(left,index); //此时链表right返回的是等待比较的头结点

struct ListNode\* nextleft = cuts(right, index); //nextleft返回的是下一个left的位置

tail ->next = Merge(left,right); //tail的后继指向已经排序好的链表

while(tail -> next != NULL) //将tail继续向后遍历，找到下一个需要合并的位置的前一个结点

tail = tail -> next;

left = nextleft; //将left更新为nextleft，用于继续遍历

}

}

head = dummyhead -> next;

return head;

}

**双链表**

17年979

给定一个带表头结点的双向链表L,每个结点有4个数据成员：前驱结点的指针LLink,后继结点的指针RLink,数据的成员Data,和访问频度Freq;且已知双向链表L中结点一直按访问频度递减的顺序排列，即频繁访问的结点总是靠近表头；初始状态L中所有结点的Freq都是0，对双链表L的locate(x)操作，每操作一次，将数据值为x的结点访问频度Freq加1.

请设计一个算法实现对双链表L的Locate(x)操作，要求操作后L中结点仍按照访问频度的递减顺序排列。

红皮书答案

/\*设置两个指针p和q,p指针从头结点向后找到值域为x的结点，然后使其频度加1，q此时从p向前遍历，直到找到一个频度大于等于p所指结点频度的结点，

然后令p换到当前q的前面。\*/

bool Locate(DLinklist &L, int x){

DNode \*p, \*q;

p = L;

while(p != NULL && p→data != x)

p = p→rlink;

if(p != NULL)

q = p;

p→freq = p→freq + 1;

while(q→llink != L && q→llink→freq < p→freq)

q = q→llink;

p→llink→rlink = p→rlink;

if(p→rlink != NULL)

p→rlink→llink = p→llink;

p→rlink = q;

p→llink = q→llink;

p→llink→rlink = p;

q→llink = q;

return true;

}

return false;

}

其他参考书答案

/\*在查找到data域为x结点\*p时，将其freq域增1.再找到\*p结点的前驱结点\*pre,若\*pre不是头结点，且满足pre→freq < p→freq,则\*p结点和\*pre结点的地data与freq域交换，让p指向\*pre结点，pre再指向\*p的前驱结点，如此这样下去，从而使双链表中的结点按freq递减排序。\*/

int locateNode(DLinkList \*L, ElemType x){

DLinkList \*p = L→next, \*pre;

ElemType temp;

int freq1;

while(p != NULL && p→data != x)

p = p→next;

if(p ==NULL)

return 0;

else{

p→freq++;

pre = p→llink;

if(pre != L){

while(pre != L && pre→freq < p→freq){

temp = p→data;

p→data = pre→data;

pre→data = temp;

freq1 = p→freq;

p→freq = pre→freq;

pre→freq = freq1;

p = pre;

pre = p→llink;

}

}

return 1;

}

}

**二叉树**

1. 已知一颗二叉树的先序和中序序列（后序和中序），恢复二叉树。
2. 判断一颗二叉树是否是完全二叉树，平衡二叉树，二叉排序树。
3. 求两个结点的最近公共祖先结点。
4. 根到某结点的路径（后序非递归遍历）/打印值为x结点的所有祖先，假设值为x的结点不多与一个。这两个一样的，表述不一样而已
5. 自上而下，自左向右 /自下而上，自左向右 /自下而上，自右向左的遍历。
6. 求一颗二叉树的宽度，高度，直径
7. 求二叉树中独生叶结点的个数（每一层独生叶结点的个数），叶子结点总数，双分支结点个数
8. 先根次序遍历二叉树，输出所有的结点的关键字及层次。/先序遍历序列第k个结点值。
9. n个节点的完全二叉树存放在数组A[n]中，建立一棵用二叉链表表示的二叉树根用tree指向。并遍历这棵二叉树
10. 根据给定序列创建二叉树。
11. （期末）判断两棵树是否相同/相似。
12. 判断一棵二叉树是否对称。/翻转一棵二叉树，每个结点的左右结点交换。
13. 删除每个元素值为x的结点，删去以它为根的子树，并释放相应的空间。
14. 打印二叉树中左子树非空右子树为空的所有结点。
15. 找出二叉树先根序列的最后一个结点，不使用栈，不使用递归。
16. （期末）输出该二叉树中第一条最长的路径长度，并输出此路径上各结点的值。
17. （期末题）一棵二叉树以链接形式存储，结点结构为(Left, Data, Right), 设计一算法，求二叉树中从根结点到叶结点的一条路径长度等于树的高度的路径，若有这样的路径存在多条，则输出路径终点（叶结点）在“最左”的一条。
18. （期末题）设计一个算法求结点x在二叉树中的双亲。
19. （期末题）一棵二叉树以链接形式存储，结点结构为（Left, Data, Right）,求其指定的某一层k(k>0)上的叶子结点个数。
20. 其他期末简单题

二叉排序树

1. 二叉搜索树中第K小的元素

1. 删除二叉排序树指定结点
2. 将KEY域之值大于等于给定值x的结点全部删除。
3. （期末题）求结点在二叉排序树中层次的算法。
4. 二叉搜索树的最近公共祖先。
5. 已知先序和中序序列，构造一颗二叉树

算法思想：递归

1.从先序preorder中遍历选择，创建节点p，值为val，然后在inorder中寻找第一个值为val的元素，下标为loc，此时loc左边元素的值为p的左子树，loc节点右边的值为p的右子树

2.此时 新的区间长度 mid = loc – inleft，对于新的左子树，构造二叉树先序区间为[preleft + 1, preleft + mid]， 中序区间为[inleft, loc - 1]，对于新的右子树，构造二叉树先序区间为[preleft + mid + 1, preright]，中序区间为[loc + 1, inright]

3.递归调用函数 root -> left = create(preorder,preleft + 1,preleft + mid,inorder,inleft,loc - 1)，root -> right = create(preorder,preleft + mid + 1, preright,inorder,loc + 1, inright)

递归边界： preleft > preright || inleft > inright 即为区间错误，返回NULL

struct TreeNode\* create(int\* preorder, int preleft, int preright, int\* inorder, int inleft, int inright)

{

if(preleft > preright || inleft > inright)

return NULL;

struct TreeNode\* root = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

root -> val = preorder[preleft];

int loc;

for(int i = inleft; i <= inright; i++)

if(inorder[i] == root -> val)

{

loc = i;

break;

}

int mid = loc - inleft;

root -> left = create(preorder,preleft + 1,preleft + mid,inorder,inleft,loc - 1);

root -> right =create(preorder,preleft + mid + 1, preright,inorder,loc + 1, inright);

return root;

}

struct TreeNode\* buildTree(int\* preorder, int preorderSize, int\* inorder, int inorderSize){

return create(preorder,0,preorderSize-1,inorder,0,inorderSize-1);

}

2. 已知后序和中序序列,构造一颗二叉树

算法思想：递归

1.从最后一个元素开始向前遍历后序遍历postorder,值为val，并在中序遍历inorder寻找值为val的元素loc

2.设置区间长度为mid = loc - inleft，创建结点，值为postorder[postright]

左子树的后序遍历区间范围为[postleft, postleft + mid - 1]，中序遍历区间范围为[inleft, loc - 1]，右子树的后序遍历区间范围为[postleft + mid, postright - 1]，中序遍历区间范围为[loc + 1, inright]，递归调用root -> left = create(postorder,postleft, postleft + mid - 1,inorder,inleft, loc - 1); root -> right = create(postorder,postleft + mid, postright - 1,inorder,loc + 1, inright);

3.边界条件，当postleft > postright || inleft > inright 时说明超出范围，返回NULL

struct TreeNode\* create(int\* postorder,int postleft, int postright, int\* inorder, int inleft, int inright)

{

if(postleft > postright || inleft > inright)

return NULL;

struct TreeNode\* root = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

root -> val = postorder[postright];

int loc;

for(int i = inleft; i <= inright; i++)

if(inorder[i] == root -> val)

{

loc = i;

break;

}

int mid = loc - inleft;

root -> left = create(postorder,postleft, postleft + mid - 1,inorder,inleft, loc - 1);

root -> right = create(postorder,postleft + mid, postright - 1,inorder,loc + 1, inright);

return root;

}

struct TreeNode\* buildTree(int\* inorder, int inorderSize, int\* postorder, int postorderSize){

return create(postorder,0,postorderSize - 1,inorder,0,inorderSize - 1);

}

1. 判断是否是完全二叉树

根据完全二叉树的定义，具有n个结点的完全二叉树与满二叉树中编号1-n的结点对应。采用层次遍历算法，将所有结点加入队列，遇到空结点时，看其后是否有非空结点。若有，则不是完全二叉树

bool ImComplete(BiTree T){

InitQueue(Q);

if(!T)

return 1;

EnQueue(Q, T);

while(!IsEmpty(Q)){

DeQueue(Q, p);

if(p){ //结点非空，将其左右孩子加入队列

EnQueue(Q, p->lchild);

EnQueue(Q, p->rchild);

}

else

while(!IsEmpty(Q)){

DeQueue(Q, p);

if(p)

return 0;

}

}

return 1;

}

1. 判断是否是二叉平衡树

设置二叉树的平衡标记balance,标记返回二叉树bt是否是平衡二叉树，若为平衡二叉树，则返回1，否则返回0；h为二叉树bt的高度，采用后序遍历的递归算法。

1.若bt为空，则高度为0，balance = 1.

2.若bt仅有根结点，则高度为1，balance = 1

3.否则，对bt的左右子树进行递归，返回左右子树高度和平衡标记，bt的高度为最高子树的高度加1.若左右子树的高度差大于1，则balance = 0;若左右子树的高度差小于等于1，且左右子树都平衡时，balance = 1,否则balance = 0.

void Judge\_AVL(BiTree bt, int &balance, int &h){

int bl = 0, br = 0, hl = 0, hr = 0; //左右子树的平衡标记和高度

if(bt == NULL){

h = 0;

balance = 1;

}

else if(bt->lchild == NULL && bt->rchild == NULL){ //仅有根节点

h = 1;

balance = 1;

}

else{

Judge\_AVL(bt->lchild, bl, hl) //递归判断左子树

Judge\_AVL(bt->rchild, br, hr) //递归判断右子树

h = (hl>hr?hl:hr)+1;

if(abs(hl-hr)<2)

balance = bl&&br; //都平衡时平衡

else

balance = 0;

}

}

1. 判断一棵树是否二叉排序树

采用中序遍历递归二叉树，初始化最小值节点为INT\_MIN，对遍历到的节点判定条件如下

（一）若该节点的值小于左子树或者大于右子树，则直接返回false

（二）若该节点小于min，则直接返回false

每次执行结束递归后，更新min为当前节点的值

void Inorder (struct TreeNode\* root, int\* min, bool\* flag)

{

if(root == NULL)

return;

Inorder(root -> left, min, flag);

if(root -> val <= \*min)

{

\*flag = false;

return;

}

\*min = root -> val;

Inorder(root -> right, min, flag);

}

bool isValidBST(struct TreeNode\* root){

if(root == NULL || (root -> left == NULL && root -> right == NULL))

return true;

int min = -32767;

bool flag = true;

Inorder(root, &min, &flag);

return flag;

}

非递归写法

bool isValidBST(struct TreeNode\* root){

struct TreeNode\* stack[5000];

int top = -1;

long pre = LONG\_MIN;

struct TreeNode\* p = root;

while(p != NULL || top != -1)

{

if(p != NULL)

{

stack[++top] = p;

p = p -> left;

}

else

{

p = stack[top--];

if(p -> val <= pre)

return false;

pre = p -> val;

p = p -> right;

}

}

return true;

}

1. 求两个结点的最近公共祖先结点

p与q的位置关系分为三种情况

p是q的祖先、q是p的祖先、p和q存在共同祖先

递归自底向上寻找

当遍历到p或q时，则直接返回该节点

当遍历到空节点时，直接向上返回空

若遍历到的节点非p,q，同时也非空，那么判断左右两边是否为空，

若左子树未找到p q，则返回右子树

若右子树未找到p q，则返回左子树

若左右子树均存在，则直接返回该节点（说明分别在这个节点的左右子树中）

struct TreeNode\* lowestCommonAncestor(struct TreeNode\* root, struct TreeNode\* p, struct TreeNode\* q) {

if(root == NULL || root == p || root == q)

return root;

struct TreeNode\* lefttree = lowestCommonAncestor(root->left,p,q); //在左子树中寻找

struct TreeNode\* righttree = lowestCommonAncestor(root->right,p,q); //在右子树中寻找

if(lefttree == NULL)

return righttree;

else if(righttree == NULL)

return lefttree;

return root;

}

非递归算法

后序遍历最后访问根结点，即在递归算法中，根是压在栈底的。设p在q的左边。采用后序非递归算法，栈中存放二叉树结点的指针，当访问到某结点时，栈中所有元素均为该结点的祖先。后序遍历必然先遍历到结点p,栈中元素均为p的祖先，先将栈复制到另一辅助栈中。继续遍历到结点q,将栈中元素从栈顶开始逐个到辅助栈中去匹配，第一个匹配的元素就是p和q的最近公共祖先。

typedef struct{

BiTree t;

int tag; //tag = 0表示左子女被访问，tag=1表示右子女被访问过

}stack;

stack s[], s1[]; //容量足够大的栈

BiTree Ancestor(BiTree ROOT, BiTNode \*p,BiTNode \*q){

top = 0;

bt = ROOT;

while(bt != NULL || top>0){

while(bt != NULL && bt!=p && bt != q) //结点入栈

while(bt != NULL){

s[++top].t = bt;

s[top].tag = 0;

bt = bt->lchild;

}

while(top != 0 && s[top].tag == 1){

//假定p在q的左侧，遇到p时，栈中元素均为p的祖先

if(s[top].t == p){

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

s1[i] = s[i]

top1 = top;

} //将栈s的元素转入辅助栈s1保存

if(s[top].t == q) //找到q结点

for(i = top; i>0; i--){ //将栈中元素的树节点到s1中去匹配

for(j = top1; j>0; j--)

if(s1[j].t == s[i].t)

return s[i].t; //p和q的最近公共祖先已找到

}

top--;

}

if(top != 0){

s[top].tag = 1;

bt = s[top].t->rchild; //沿右分支向下遍历

}

}

return NULL; //无祖先

}

1. 输出root到p的路径

采用后序非递归遍历树，引入一个栈辅助，当出栈元素为p时，栈中元素恰为其所有祖宗结点，即所求路径。

void Searchroad(BiTree root, BiTree \*p){

BiTree Stack[MaxSize];

int top = -1;

BiTree \*q, \*r;

q = root;

while(q != NULL || top != -1){

if(q != NULL){

Stack[++top] = q;

q = q->lchild;

}

else{

q = Stack[top];

if(q->rchild != NULL && q ->rchild != r){

q = q->rchild;

Stack = [++top] = p;

q = q->lchild;

}

else{

q = Stack[top--];

if(q == p){

Stack[++top] = p;

return;

}

r = q;

q = NULL;

}

}

}

1. 在二叉树中查找值为x的结点，编写算法打印值为x的结点的所有祖先，假设值为x的结点不多于1个。

采用非递归后序遍历，最后访问根结点，访问到值为x的结点时，栈中所有元素均为该结点的祖先，依次出栈打印即可。

typedef struct{

BiTree t;

int tag;

}stack; //tag=0表示左子女被访问，tag=1表示右子女被访问

void Search(BiTree bt, ElemType x){

stack s[];

top = 0;

while(bt != NULL || top>0){

while(bt!=NULL && bt->data !=x){

s[++top].t = bt;

s[top].tag = 0;

bt = bt->lchild;

}

if(bt->data == x){

print("")

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

printf("%d", s[i].t->data);

exit(1);

}

while(top != 0 && s [top].tag == 1)

top--;

if(top != 0){

s[top].tag = 1;

bt = s[top].t->rchild;

}

}

}

1. 自上而下，自左向右遍历（原始层序遍历）

void LevelOrder(BiTree){

BiTree Queue[maxsize];

int front = -1, rear = -1;

BiTree p;

Queue[++rear] =T;

while(front != rear){

p = Queue[++front];

Visit(p);

if(p->lchild != NULL)

Queue[++rear] = p->lchild;

if(p->rchild != NULL)

Queue[++rear] = p->rchild;

}

}

1. 自下而上，自右向左遍历。

利用原有的层次遍历算法，出队的同时将各结点指针入栈，在所有结点入栈后在从栈顶开始依次访问即得所求的算法。

void InvertLevel(BiTree bt){

Stack s;

Queue Q;

if(!bt = NULL){

InitStack(s);

InitQueue(Q);

EnQueue(Q, bt);

while(!IsEmpty(Q) == false){

DeQueue(Q, p);

Push(s, p);

if(p->lchild)

EnQueue(Q, p->lchild);

if(p->rchild)

EnQueue(Q, p->rchild);

}

while(IsEmpty(s) == false){

Pop(s, p);

visit(p->data);

}

}

}

1. 自下而上，自左向右遍历。

与上个代码的差别是先右孩子入队，再左孩子入队

void InvertLevel(BiTree bt){

Stack s;

Queue Q;

if(!bt = NULL){

InitStack(s);

InitQueue(Q);

EnQueue(Q, bt);

while(!IsEmpty(Q) == false){

DeQueue(Q, p);

Push(s, p);

if(p->rchild)

EnQueue(Q, p->rchild);

if(p->lchild)

EnQueue(Q, p->lchild);

}

while(IsEmpty(s) == false){

Pop(s, p);

visit(p->data);

}

}

}

1. 求二叉树的宽度

int LevelOrder(BiTree T){

BiTree Queue[maxsize];

int front = -1, rear = -1, width = 0, last = 0, max = 0;

BiTree p;

Queue[++rear] = T;

while(front != rear){

p = Queue[++front]

width++;

if(p→lchild != NULL)

Queue[++rear] = p→lchild;

if(p→rchild != NULL)

Queue[++rear] = p→rchild;

if(front == last){

if(max<width)

max = width;

width == 0;

last = rear;

}

}

return max;

}

递归算法

void GetWidth(BTNode \*p, int cnt[], int \*w, int i) //i为当前结点p所在的层次cnt[i]为第i层含有的结点个数，w保存二叉树的宽度。

{

if(p != NULL){

cnt[i]++;

if(w < cnt[i])

\*w = cnt[i];

GetWidth(p->lchild, cnt, w, i+1);

GetWidth(p->rchild, cnt, w, i+1);

}

}

1. 求二叉树的高度

红皮书答案

int LeverOrder(Bitree T){

BiTree Queue[maxsize];

int front = -1, rear = -1, level = 0, last = 0;

BiTree p;

Queue[++rear] = T;

while(front != rear){

p = Queue[++front];

if(p→lchild != NULL)

Queue[++rear] = p→lchild;

if(p→rchild != NULL)

Queue[++rear] = p→rchild;

if(front == last){

level++;

last = rear;

}

}

return level;

}

计算深度及宽度

void level(BTNode \*p){

int front,rear,level = 0,length = 0;

int count = 0,width = 1,cursor; //这里width为宽度，level为深度

BTNode \*q;

if(p != NULL){

rear = (reat+1)%maxsize;

que[rear] = p;

cursor = rear;

while(front != rear){

front = (front+1)%maxsize;

count ++;

q = que[front];

if(q->lchild != NULL){

rear = (rear+1)%maxsize;

que[rear] = q->lchild;

}

if(q->rchild != NULL){

rear = (rear+1)%maxsize;

que[rear] = q->rchild;

}

if(front == cursor){ //是否扫描完一层

cursor = rear; //将下一层的最后一个节点赋值给扫描器

level ++; //层数+1

width = count > width ? count : width; //取最大宽度

count = 0; //重新计算下一层

}

}

}

}

1. 二叉树的最小高度（补充的，真题没有）

void dfs(struct TreeNode\* root, int\* min, int num)

{

if(root == NULL)

return;

if(root -> left == NULL && root -> right == NULL)

{

if(num < \*min)

\*min = num;

}

dfs(root -> left, min, num + 1);

dfs(root -> right, min, num + 1);

}

int minDepth(struct TreeNode\* root){

if(root == NULL)

return 0;

int min = INT\_MAX;

dfs(root,&min,1);

return min;

}

1. 二叉树的直径（任意两个结点路径的最大值）

递归

1.先序遍历每个结点，递归的求出每个结点的直径，进行比较

2.直径 = 左子树高度 + 右子树高度

其中 高度等于（左子树高度，右子树高度）+ 1

int max (int a, int b)

{

return a > b ? a : b;

}

int Treehigh(struct TreeNode\* root)

{

if(root == NULL)

return 0;

return max(Treehigh(root -> left), Treehigh(root -> right))+ 1;

}

void dfs(struct TreeNode\* root, int\* max)

{

if(root == NULL)

return;

int temp = Treehigh(root -> left) + Treehigh(root -> right);

if(temp > \*max)

\*max = temp;

dfs(root -> left, max);

dfs(root -> right, max);

}

int diameterOfBinaryTree(struct TreeNode\* root){

int max = 0;

dfs(root, &max);

return max;

}

1. 二叉树的独生叶结点个数

int LeverOrder(BiTree T){

BiTree Queue[maxsize];

int front = -1, rear = -1;

int num = 0;

BiTree p;

Queue[++rear] = T;

while(front != rear){

p = Queue[++front];

if(p→lchild != NULL && p→rchild == NULL)

if(p→lchild→lchild == NULL && p→lchild→rchild == NULL)

num++;

if(p→lchild == NULL && p→rchild != NULL)

if(p→rchild→lchild == NULL && p→rchild→rchild == NULL)

num++;

if(p→lchild != NULL){

Queue[++rear] = p→lchild;

}

if(p→rchild != NULL){

Queue[++rear] = p→rchild;

}

}

if(T→lchild == NULL && T→rchild == NULL)

num++;

return num;

}

1. 每一层独生叶结点个数(默认根结点为第1层)

叶子结点，要求其无后代，而独生结点，要求其无兄弟。算法思想：引入一个队列辅助，基本操作为层序遍历，引入一个计数值count,记录当前层下一层独生叶子结点的个数，每当一个结点入队，判断是否只有一个孩子，若是进而判断是否是叶子结点，若是，计数加一。用一个标志last实时随队尾更新，若出队结点时的front=last,说明当前遍历的是某一层最后一个结点，然后输出某层的下一层的独生叶子结点个数及层号。

void LeafNoBrother\_InEachlevel(BiTree root){

BiTree[MaxSize];

int rear = -1, front = -1;

int level = 1, count = 0, last = 0;

BiTree p;

if(root != NULL){

if(root->lchild == NULL && root->rchild ==NULL){

printf(“%d”, 0);

printf(“%d”, 1);

return;

}

}

Queue[++rear] = root;

while(front != rear){

p = Queue[++front];

if(p->lchild != NULL && p->rchild == NULL)

if(p->lchild->lchild == NULL && p->lchild->rchild ==NULL)

count++;

if(p->lchild == NULL && p->rchild != NULL)

if(p->rchild->lchild == NULL && p->lchild->rchild ==NULL)

count++;

if(p->lchild != NULL)

Queue[++rear] = p->lchild;

if(p->rchild != NULL)

Queue[++rear] = p->rchild;

if(front == last){

level++;

printf(“%d”, level);

printf(“%d”, count);

count = 0;

last = rear;

}

}

}

1. 二叉树叶子结点总数

int LevelOrder(BiTree T){

BiTree Queue[maxsize];

int front = -1, rear = -1, count = 0;

BiTree p;

Queue[++rear] = T;

while(front != rear){

p = Queue[++front];

if(p→lchild == NULL && p→rchild == NULL)

count++;

if(p→lchild != NULL)

Queue[++rear] = p→lchild;

if(p→rchild != NULL)

Queue[++rear] = p→rchild;

}

return count;

}

1. 计算一棵给定二叉树的所有双分支结点个数

int DSonNodes(BiTree b){

if(b == NULL)

return 0;

else if(b->lchild != NULL && b->rchild != NULL) //双分支结点

return DSonNodes(b->lchild) + DSonNodes(b->rchild)+1;

else //其他情况，单分支或者叶子结点

return DSonNodes(b->lchild) + DSonNodes(b->rchild);

}

也可以设置一个全局变量Num,每遍历到一个结点时，判断每个结点的是否是双分支结点，Num++.

1. 先根次序遍历二叉树，输出所有的结点的关键字及层次。

void nodePrint(BiTNode \*t, int i){

if(t! = NULL){

printf(“结点的值为%d, 结点的层次为%d”, t->data, i)

nodePrint(t->lchild, i+1);

nodePrint(t->rchild, i+1);

}

}

1. 假设二叉树采用二叉树存储存储，设计一个算法，求先序遍历序列中第k个结点的值。（王道的）

设置一个全局变量i记录已访问过的结点的序号，其初值是根结点在先序序列中的序号即1，当二叉树b为空时返回特殊字符'#',当i == k时，表示找到了满足条件的结点，返回b->data;当i不等于k时，递归地在左子树中查找，若找到则返回该值，否则继续递归地在右子树中查找，返回其结果。

int i = 1；

ElemType PreNode(BiTree b, int k){

//本算法查找二叉树先序遍历序列中的第k个结点的值

if(b == NULL)

return '#';

if(i == k)

return b->data;

i++;

ch = PreNode(b->lchild, k);

if(ch != '#')

return ch;

ch = PreNode(b->rchild, k);

if(ch != '#')

return ch;

}

1. （补充的）n个节点的完全二叉树存放在数组A[n]中，建立一棵用二叉链表表示的二叉树根用tree指向。

BTNode \*Create(ElementType a[], int i){

BTNode \* tree;

if(i <= n){

tree = (BTNode) malloc(sizeof(BTNode));

tree->data = A[i];

if(2\*i > n)

tree->lchild = NULL;

else

tree->lchild = Create(A, 2\*i);

if(2\*i+1>n)

tree->rchild = NULL;

else

tree->rchild = Create(A, 2\*i+1);

}

return tree;

}

遍历创建的这棵二叉树

void preorder(ElemType bt[], int n){

int i = 1,top = 0,s[];

while(i<= n || top > 0){

while(i<n){

visit(bt[i]); //访问根节点

if(2\*i+1 <= n)

s[++top] = 2\*i+1; //右子树入栈

i = 2\*i; //不断访问左子树

}

if(top>0)

i = s[top--]; //开始访问右子树

}

}

1. 根据给定序列创建二叉树（此代码会反向处理，正向处理需要先将数组逆序）

一个由0，1，2组成的字符序列，称之为二叉树

序列S; S=0:如果r没有子结点，S=1S1:如果r有一个子结点；S=2S1S2：如果r有两个子结点，S1、S2分别表示其两个子树的二叉树序列

Bitree CreateBitree[int a[i], int n]{

Bitree Stack[n];

int top = -1;

for(int i = n-1;i >= 0; i--){

if(a[i] == 0){

Bitree bt = (Bitree)malloc(sizeof(Bitree));

bt→lchild = NULL;

bt→rchild = NULL;

Stack[++top] = bt;

}

else if(a[i] ==1){

Bitree bt = (Bitree)malloc(sizeof(Bitree));

Bitree p = Stack[top--];

bt→rchild = NULL;

Stack[++top] = bt;

}

else{

Bitree bt = (Bitree)malloc(sizeof(Bitree));

Bitree p = Stack[top--];

bt→lchild = p;

Bitree q = Stack[top--];

bt→rchild = q;

Stack[++top] = bt;

}

}

return Stack[0];

}

1. 判断两棵树是否相同

递归

需要执行的步骤为判断此时的两个结点是否同时为空，若是则返回true，

若不同时为空或值不同则返回false

bool isSameTree(struct TreeNode\* p, struct TreeNode\* q){

if(p == NULL && q == NULL)

return true;

if(p == NULL || q == NULL || p -> val != q -> val)

return false;

return isSameTree(p->left,q->left) && isSameTree(p -> right, q -> right);

}

非递归先序遍历

1.初始化两个结点分别指向两颗树的根节点，创建两个栈保存两棵树的结点，从根节点开始先序遍历

2.若两结点的值不同，那么则树不同，返回false，

每当遇到一个为空，一个不空的情况时，返回false

每当遇到空结点时，两个栈同时出栈，然后向右走一个单位

#define MAXSIZE 100

bool isSameTree(struct TreeNode\* root1, struct TreeNode\* root2){

if(root1 == NULL && root2 == NULL) //若两棵树均为空

return true;

if(root1 == NULL || root2 == NULL) //只有一棵树为空

return false;

//创建两个堆栈，用于保存先序遍历的结点

struct TreeNode\*\* stack1 = (struct TreeNode\*\*)malloc(sizeof(struct TreeNode\*) \* MAXSIZE);

struct TreeNode\*\* stack2 = (struct TreeNode\*\*)malloc(sizeof(struct TreeNode\*) \* MAXSIZE);

int top = -1;

struct TreeNode \*p = root1, \*q = root2;

while((p != NULL && q != NULL) || top != -1)

{

if(p != NULL && q != NULL)

{

if(p -> val != q -> val)

return false;

top++;

stack1[top] = p;

stack2[top] = q;

p = p -> left;

q = q -> left;

}

else if((p != NULL && q == NULL) || (p == NULL && q != NULL))

return false;

else

{

p = stack1[top];

q = stack2[top];

top--;

p = p -> right;

q = q -> right;

}

}

return true;

}

1. 判断两棵树是否相似

要求树的形状一样，值可以不同

bool similar(BiTree T1 && BiTree T2){

if(T1 == NULL && T2 == NULL)

return true;

else if(T1 == NULL && T2 == NULL)

return false;

else if(!similar(T1->lchild, T2->rchild))

return false;

else return similar(T1->rchild, T2->rchild);

}

1. 判断一棵树是否对称

递归

bool issymmery (struct TreeNode\* lchild, struct TreeNode\* rchild)

{

if(lchild == NULL && rchild == NULL)

return true;

if(lchild == NULL || rchild == NULL)

return false;

if(lchild -> val != rchild -> val)

return false;

return issymmery(lchild->left, rchild -> right) && issymmery(lchild->right, rchild -> left);

}

bool isSymmetric(struct TreeNode\* root){

if(root == NULL)

return true;

return issymmery(root -> left, root -> right); //左树和右树开始对称比较

}

非递归

bool isSymmetric(struct TreeNode\* root){

if(root == NULL)

return true;

struct TreeNode\* queue[1500];

int front = -1, rear = -1;

queue[++rear] = root -> left; //将根的左右子树进队

queue[++rear] = root -> right;

while(front != rear)

{

struct TreeNode \*p = queue[++front];

struct TreeNode \*q = queue[++front];

if(p == NULL && q == NULL) //若同时为空则继续出队

continue;

if(p == NULL || q == NULL) //一方为空，另一方不为空则返回false

return false;

if(p -> val != q -> val) //值不同返回false

return false;

queue[++rear] = p -> left; //按照对称顺序依次进队

queue[++rear] = q -> right;

queue[++rear] = p -> right;

queue[++rear] = q -> left;

}

return true;

}

1. 将一棵二叉树每个结点的左右子树翻转（不能使用中序遍历，其他都行）

递归实现（先序遍历框架）

struct TreeNode\* invertTree(struct TreeNode\* root){

if(root == NULL)

return NULL;

struct TreeNode\* temp = root -> left; //交换结点

root -> left = root -> right;

root -> right = temp;

invertTree(root -> left);

invertTree(root -> right);

return root;

}

非递归（层次遍历）

1.判断根节点是否为空，若为空则直接返回，否则将根节点的左右子树入队

2.将结点出队，同时交换该结点的左右子树，若左右子树存在时，将左右子树入队

3.反复执行过程2，直至队空

struct TreeNode\* invertTree(struct TreeNode\* root){

if(root == NULL)

return root;

struct TreeNode\* queue[500];

int front = -1, rear = -1;

queue[++rear] = root;

while(front != rear)

{

struct TreeNode\* p = queue[++front];

struct TreeNode\* temp = p -> left;

p -> left = p -> right;

p -> right = temp;

if(p -> left != NULL)

queue[++rear] = p -> left;

if(p -> right != NULL)

queue[++rear] = p -> right;

}

return root;

}

1. （王道）在二叉树上查找所有以x为元素值的结点，并删除以其为根的子树。

删除值为x的结点，意味着应将其父结点的左（右）孩子指针置为空。用层序遍历易于找到某结点的父亲，本题要求删除每个值为x的结点的子树，因此要完全遍历二叉树。

void DeleteXTree(BiTree bt){ //删除以bt为根的子树

if(bt){

DeleteXTree(bt->lchild);

DeleteXTree(bt->rchild);

free(bt);

}

}

void Search(BiTree bt, ElemType x){

BiTree Q[]; //Q存放二叉树结点指针的队列，容量足够大。

if(bt){

if(bt->data == x){

DeleteXTree(bt);

exit(0);

}

Init Queue(Q);

EnQueue(Q, bt);

while(!IsEmpty){

DeQueue(Q,p);

if(p->lchild) //若左孩子非空

if(p->lchild == x){ //左子树符合则删除左子树

DeleteXTree(p->lchild);

p->lchild = NULL;

}

else

EnQueue(Q, p->lchild);

if(p->rchild) //若右孩子非空

if(p->rchild->data == x){ //右孩子符合要求则删除

DeleteXTree(p->rchild);

p->rchild = NULL;

}

else

EnQueue(Q,p->rchild);

}

}

}

1. 打印二叉树左子树非空，右子树为空的所有结点

中序遍历(右中左)

void inOrderNonRecurseTraverse(BTNode \*p){

if(p == NULL)

return; //空树不操作

initStack(S); //初始化一个栈

while(p || !Empty(S)){

push(S, p);非空结点进栈

p = p->rchild;

}

//遇到空结点就得出栈

p = pop(S); //p接收出栈元素

if(p->lchild && !p->rchild) //打印左子树非空右子树为空的结点。

printf(“%d\n”, p->data);

p = p->lchild;

}

}

1. 找出二叉树先根序列的最后一个结点，不使用栈，不使用递归。

红皮书答案

算法思想：一定是最靠右的叶子结点，尽量往右，往下。

BiTree FindLast(BiTree root){

BiTree \*p = root;

while(p->right != NULL || p->left != NULL){

if(p ->right){

while(p->right != NULL)

p = p->right;

}

else

p = p->left;

}

return p;

}

也可以使用层序遍历，最后一个结点一定是先根序列的最后一个结点。

1. （期末）输出该二叉树中第一条最长的路径长度，并输出此路径上各结点的值。与4小题的根到某结点的路径对比

二叉树的最长路径一定是从根结点到某个叶子结点的路径，用形参maxpath[]数组存放最长路径，maxpathlen存放最长路径长度

递归

void Maxpath1(BTNode \*b, ElemType path[], int pathlen, ElemType maxpath[], int &maxpathlen) //pathlen和maxpathlen的初值均为0

int i;

if (b == NULL){

if(pathlen>maxpathlen){ //通过比较求最长路径

for(i = pathlen-1; i>=0; i--)

maxpath[i] = path[i];

maxpathlen = pathlen;

}

}

else{

path[pathlen] = b->data; //将当前结点放入路径中

pathlen++; //路径长度增1

MaxPath1(b-lchild, path, pathlen, maxpath, maxpathlen); //递归扫描左子树

MaxPath1(b-rchild, path, pathlen, maxpath, maxpathlen); //递归扫描右子树

}

}

非递归层次遍历（广度优先遍历）

void MaxPath(BTNode \*b, ElemType maxpath[], int &maxpathlen){

//pathlen和maxpathlen的初值均为0

struct snode{

BtNode \*node; //存放当前指针

int parent; 存放双亲结点在队列中的位置

}Qu[MaxSize]; //定义非循环队列

ElemType path[MaxSize]; //存放一条路径

int pathlen; //存放一条路径长度

int front, rear, p, i; //定义队头队尾指针

front = rear = -1; //置队列为空

rear++;

Qu[rear].node = b; //根结点入队

Qu[rear].parent = -1; //根结点没有双亲

while(front<rear){

front++;

b = Qu[front].node;

if (b->lchild ==NULL && b->rchild ==NULL){ //\*b为叶子结点

p = front;

pathlen = 0;

while(Qu[p].parent != -1){

path[pathlen] = Qu[p].node->data;

pathlen ++;

p = Qu[p].parent;

}

path[pathlen] = Qu[p].node->data;

pathlen++;

if(pathlen>maxpathlen){

for(i = pathlen-1; i>=0; i--)

maxpath[i] = path[i];

maxpathlen = pathlen;

}

}

if (b->lchild != NULL){

rear++;

Qu[rear].node = b->lchild;

Qu[rear].parent = front;

}

if (b->rchild != NULL){

rear++;

Qu[rear].node = b->rchild;

Qu[rear].parent = front;

}

}

1. （期末题）一棵二叉树以链接形式存储，结点结构为(Left, Data, Right), 设计一算法，求二叉树中从根结点到叶结点的一条路径长度等于树的高度的路径，若有这样的路径存在多条，则输出路径终点（叶结点）在“最左”的一条。

递归求二叉树的高度

int Btdepth(BiTree T){

if(T == NULL)

return 0;

int ldep = Btdepth(T→lchild);

int rdep = Btdepth(T→rchild);

if(ldep > rdep)

return ldep + 1;

else

return rdep + 1;

}

方法1.层次遍历，遍历到第k(为高度)层，寻找第一个叶子结点，返回。

方法2先序递归遍历

void Pathlen\_is\_depth(BTNode \*b, ElemType path[], int pathlen, k){

//初始调用时path为空，pathlen为0, k为树的高度

if (b != NULL){

if(b->lchild == NULL && b->rchild == NULL && pathlen==k)

printf(“%c到根结点的路径：%c”, b->data, b->data);

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

printf(“%c”, path[i]);

printf(‘’\n”)

}

else{

path[pathlen] = b->data;

pathlen++;

Pathlen\_is\_depth(b->lchild, path, pathlen, k);

Pathlen\_is\_depth(b->rchild, path, pathlen, k);

}

}

}

1. （期末题）设计一个算法求结点x在二叉树中的双亲。

红皮书题目官方答案

typedef struct node {

datatype data;

struct node \*lchild,\*rchild;

}bitree;

bitree \*q[20];

int r = 0, f= 0, flag = 0;

void preorder(bitree \*bt, char x)

{

if(bt != 0 && flag == 0)

if(bt->data == x){

flag = 1;

return;

}

else{

r = (r+1)%20;

q[r] = bt;

preorder(bt->lchild, x);

preorder(bt->rchild, x);

}

}

void parent(bitree \*bt, char x){

int i;

preorder(bt, x);

for(i = f + 1; i<=r; i++)

if(q[i]->lchild->data == x || q[i]->rchild->data)

break;

if(flag == 0)

printf(“not found x\n”);

else if(I <=r)

printf(“%c”, bt->data);

else

printf(“not found x\n”);

}

递归：如果\*t的左孩子或右孩子是\*p,则\*p的双亲结点是\*t,否则递归到\*t的左子树或右子树中寻找。此代码没有定义结构体，与上面一样，判断改为判断结点数据即可。

BiTNode \*getParent(BiTNode \*t, BiTNode \*P){

if(t == NULL)

return NULL;

if(t == p)

return NULL;

if(t->lchild == p || t->rchild == p)

return t;

BiTNode \*s = getParent(t->lchild, p);

if (s != NULL)

return s;

else

return getParent(t->rchild, p);

}

1. （期末题）一棵二叉树以链接形式存储，结点结构为（Left, Data, Right）,求其指定的某一层k(k>0)上的叶子结点个数。

用队列Qu进行层次遍历，若一层的结点入队后，rear指向该层中的最右结点，将其赋给last(对于第一层，last=1),在出队时，若front = last,表示这一层处理完毕，表示层号的level加1.

int LeafKLevel(BTNode \*b, int k){

BTNode \*Qu[MaxSize]; //定义循环队列

int front, rear; //定义队首，队尾指针

int leaf = 0; //leaf累计叶子结点个数

int last;

int level;

front = rear = 0;

if(b == NULL || K<=1)

return(0);

rear++;

last = rear;

level = 1;

Qu[rear]=b;

while(rear!=front){

front = (front+1)%MaxSize;

b = Qu[front];

if(level==k && b->lchild==NULL && b->rchild==NULL)

leaf++;

if(b->lchild != NULL){

rear = (rear+1)%MaxSize;

Qu[rear]=b->lchild;

}

if(b->lchild != NULL){

rear = (rear+1)%MaxSize;

Qu[rear]=b->lchild;

}

if(front==last){

level++;

last = rear;

}

if(level>k)

return(leaf);

}

}

1. 其他期末题

计算二叉树中所有结点值之和的算法

int FindSum(BTNode \*b){

if(b ==NULL)

return 0;

else

return(b->data + FindSum(b->lchild) + FindSum(b->rchild));

}

计算一棵二叉树所有结点数

int Nodes(BTNode \*b){

int num1, num2;

if(b==NULL)

return 0;

else{

num1 = Nodes(b->lchild);

num2 = Nodes(b->rchild);

return (num1 + num2 +1);

}

}

计算一棵给定二叉树所有叶子结点的个数

int LeafNodes(BTNode \*b){

int num1, num2;

if(b == NULL)

return 0;

else if (b->lchild == NULL && b->rchild ==NULL)

return 1;

else{

num1 = LeafNodes(b->lchild);

num2 = LeafNodes(b->rchild);

return(num1+num2);

}

}

**二叉排序树**

1. 二叉搜索树中第K小的元素

中序递归遍历，设置计数器等于0，每次递归执行则加1

当count = k时，直接返回

void Inorder(struct TreeNode\* root, int k, int\* num, int\* Nodenum)

{

if(root == NULL)

return;

Inorder(root -> left,k,num,Nodenum);

(\*num)++;

if(\*num == k)

{

\*Nodenum = root -> val;

return;

}

Inorder(root -> right,k,num,Nodenum);

}

int kthSmallest(struct TreeNode\* root, int k){

if(root == NULL)

return 0;

int Nodenum, num = 0;

Inorder(root,k,&num,&Nodenum);

return Nodenum;

}

非递归算法

int kthSmallest(struct TreeNode\* root, int k){

struct TreeNode\* stack[500];

int top = -1, count = 0;

struct TreeNode\* p = root;

while(p != NULL || top != -1)

{

if(p != NULL)

{

stack[++top] = p;

p = p -> left;

}

else

{

p = stack[top--];

count++;

if(count == k)

return p -> val;

p = p -> right;

}

}

return -1;

}

1. 删除二叉排序树指定结点

算法思想：递归

1.从根节点开始递归向下寻找值为key的节点，

（1）若root -> val > key时说明待删除节点在左子树，需要对左子树进行修改

root -> left = deleteNode(root -> left, key)

（2）若root -> val < key时说明待删除节点在右子树，需要对右子树进行修改

root -> right = deleteNode(root -> right, key)

（3）若root -> val == key则说明该节点为待删除节点

2.判断待删除节点的情况

（1）若为叶子节点，直接将该节点置为空

（2）若右子树不为空，则在右子树中寻找最小值节点，同时交换root与最小值节点的val，同时再递归删除右子树中的最小值节点

（3）若右子树为空，则在左子树中寻找最大值节点，同时交换root与最大值节点的val，同时再递归删除左子树中的最大值节点

int successor(struct TreeNode\* root) //寻找后继节点

{

if(root -> left != NULL)

return successor(root -> left);

return root -> val;

}

int predecessor(struct TreeNode\* root) //寻找前驱节点

{

if(root -> right != NULL)

return predecessor(root -> right);

return root -> val;

}

struct TreeNode\* deleteNode(struct TreeNode\* root, int key){

if(root == NULL) //递归边界

return NULL;

if(root -> val > key) //需要对左子树进行修改

root -> left = deleteNode(root -> left, key);

if(root -> val < key) //需要对右子树进行修改

root -> right = deleteNode(root -> right, key);

if(root -> val == key) //需要修改该节点

{

if(root -> left == NULL && root -> right == NULL) //叶子节点

root = NULL;

else if(root -> right != NULL) //存在右子树

{

root -> val = successor(root -> right);

root -> right = deleteNode(root -> right, root -> val);

}

else //存在左子树

{

root -> val = predecessor(root -> left);

root -> left = deleteNode(root -> left, root -> val);

}

}

return root;

}

1. 将KEY域之值大于等于给定值x的结点全部删除。

红皮书答案

从根结点开始判断其值域，向右子树方向延伸，并记录每个结点的前驱，当某结点大于x,让其入队，并且从该结点开始层次遍历，当出队结点值大于给定x值，让其左孩子入队，递归删除其右子树，若小于等于，则让其右孩子入队。

void Del(BiTree T){ //递归删除树算法

if(T != NULL){

Del(T ->left);

Del(T ->right);

free(T);

}

}

void Delx(BiTree &root, int x){

BiTree Queue[MaxSize];

int rear = -1, front = -1;

BiTree \*p = root, \*pre = NULL;

while(p->right != NULL && p->key<=x){

pre = p;

p = p->right;

}

if(p->key <= x)

return;

Queue[++rear] = p;

while(rear != front){

p = Queue[++front];

if(p->key>x){

Queue[++rear] = p->left

pre->right = p->left;

p->left = NULL;

Del(p);

}

else

Queue[++rear] = p->right;

}

}

1. （期末题）求结点在二叉排序树中层次的算法。

在二叉排序树中，查找一层就下降一层。因此，查找该结点所用次数就是该节点在二叉排序树中的层次。采用二叉排序树非递归查找算法，用n保存查找层次，每查找一次，n就加一，直到找到相应的结点。

int level(BiTree bt, BSTNode \*p){

int n = 0; //统计查找次数

BiTree t = bt;

if(bt != NULL){

n++;

while(t->data != p->data){

if(p->data<t-data)

t = t->lchild;

else

t = t->rchild;

n++;

}

}

return n;

}

1. 二叉排序树的最近公共祖先

算法思想：二分搜索

1.从根节点开始比较val

若当前节点的值大于p,q，则说明公共祖先在左子树

若当前节点的值小于p,q，则说明公共祖先在右子树

若当前节点的值在p与q之间，那么说明该节点就是第一个祖先节点（因为是从最上面向下来的）

struct TreeNode\* lowestCommonAncestor(struct TreeNode\* root, struct TreeNode\* p, struct TreeNode\* q) {

if(root == NULL)

return NULL;

while(1)

{

if(root -> val > p -> val && root -> val > q -> val)

root = root -> left;

else if(root -> val < p -> val && root -> val < q -> val)

root = root -> right;

else

return root;

}

}

**图**

1. 图的DFS和BFS遍历算法。
2. 采用邻接表存储结构，编写一个判断无向图中任意给定的两个顶点之间是否存在一条长度为k的简单路径的算法
3. 输出从顶点Vi到Vj的所有简单路径。
4. 基于深度优先遍历和广度优先遍历判别有向图中是否存在由顶点Vi到顶点Vj的路径。
5. 无权最短路径问题（不带权值）
6. （绿皮书）设计一个算法，求不带权无向连通图G中距离顶点v最远的一个顶点
7. 正权单源最短路径Dijkstra（带权值）
8. 判断无向图是否是一颗树。
9. 邻接表生成相应的逆邻接表。
10. 图从邻接表转换成邻接矩阵。
11. 判断图是否有回路
12. 统计图中连通分量的个数
13. （期末题）最小边数最短路最短路径可能有多条，边数最少的最短路径称为最小边数最短路，设计高效算法，在非负权图上求顶点v到顶点w的最小边数最短路
14. 给定连通G和G中的一个结点V，求G的支撑树（T为根，T的层次遍历次序恰好是以V为起点的G的某个广度优先遍历）
15. 设有一个正权有向图G = (V, E), w是G的一个顶点，w的偏心距定义为max{从u到w的最短路径长度，u属于V},路径长度指的是路径上各边权值之和，将G中的偏心距最小的顶点称为G的中心，设计一个函数返回图G的中心。
16. 判断图G是否有根
17. 自由树（即无环连通图）T = (V, E)的直径是树中所有顶点之间最短路径的最大值，试设计一个时间复杂度尽可能低的算法求T的直径
18. 假设图采用邻接表存储，设计一个算法，判断一个图是否连通，若连通则返回1，否则返回0.
19. (期末题)假设图G采用邻接表存储，设计一个算法，输出图G中顶点u到v的长度为k的所有简单路径
20. （期末题）已知有n个顶点的有向图的邻接表，求出度
21. （期末题）求出无向无权连通图中，距离顶点v的最短路径长度为k的所有顶点，路径长度以边数为单位计算。
22. （绿皮书，真题考过）设计一个算法，判断以邻接矩阵结构存储的有向图中是否存在一个简单回路，若存在，则以顶点序列的方式输出该回路。
23. （绿皮书）假设图采用邻接表存储，编写一个算法，利用深度优先遍历求出无向图中通过定点v的简单回路。
24. 图的DFS和BFS遍历算法。

邻接矩阵的结构体

typedef struct{

char data;

char info;

} vertexType;

typedef struct {

int edges[maxsize][maxsize]

int vnums,enums;

vertexType vex[maxsize]

}mGraph

基于邻接矩阵的深度优先遍历

void DFS\_Nonrecursion(mGraph \*G, int visit[], int stackVT){

int stack[maxsize];

int top = -1;

int i,j,curVT;

stack[++top] = stackVT;

while(top != -1){

curVT = stack[top--];

cout<<G->vex[curVT].data;

for( i = G->vnums-1; i>0; i--){

if(G->edges[curVT][i] == 1 && visit[i] == 0){

stack[++top] = i;

visit[i] = 1;

}

}

}

}

基于邻接矩阵的广度优先遍历

void BFS(mGraph \*G, int visit[], int startVT){

int i,queue[maxsize],front=0,rear = 0,vertex;

queue[++rear] = startVT;

while(front != rear){

front = (front+1)%maxsize;

vertex = queue[front];

cout<<G->vex[vertex].data;

for(i = 0; i<G->nums;i++){

if(G->edges[vertex][i] == 1 && visit[i] == 0){

rear = (rear+1) % maxsize;

queue[rear] = i;

visit[i] = 1;

}

}

}

}

基于邻接矩阵的深度递归优先遍历

void DFS(mGraph \*G, int visit[], int startVT){

int j;

cout<<G->vex[startVT].data;

for(j = 0; j<G->vnums; j++){

if(G->edges[starVT][j] == 1 && visit[j] == 0){

visit[j] = 1;

DFS(G,visit,j);

}

}

}

邻接表的结构体

typedef struct ArcNode{

int adjvex;

int info; //权值

struct ArcNode \*next;

}ArcNode;

//顶点表结点

typedef struct VNode{

int data;

struct ArcNode \*firstarc;

}VNode;

typedef struct{

VNode adjlist[maxnum];

int n,e;

}AGraph;

深度优先遍历

递归方法

void dfs(AGraph \*g, int v, int visited[])

{

visited[v] = 1;

printf("%d ", v);

ArcNode\* p = g->adjlist[v].firstarc;

while(p != NULL)

{

if(visited[p->adjvex] == 0)

dfs(g,p->adjvex,visited);

p = p -> next;

}

}

void dfsmain(AGraph \*g, int v)

{

int visited[g->n];

for(int i = 0; i < g->n; i++)

visited[i] = 0;

dfs(g,v,visited);

}

非递归深度优先搜索

算法思想：

1.初始化堆栈Stack和访问数组visited，将起始顶点v入栈，将其标记为已经访问

2.栈非空时执行循环，设置k为出栈顶点，p为其邻接表上的首个边表，将边表上所有未访问的顶点全部入栈，并将其标记为已经访问

3.反复执行过程2，直至栈空

void DFS(AGraph \*g, int v)

{

int stack[g->n], visited[g->n] = {0};

int top = -1;

visited[v] = 1; //标记起始顶点为已经访问

stack[++top] = v; //入栈

while(top != -1) //栈非空执行

{

int k = stack[top--]; //出栈

printf("%d",k);

ArcNode \*p = g->adjlist[k].firstarc; //p为其邻接表上的首个边表

while(p != NULL)

{

if(visited[p -> adjvex] == 0) //若未访问过，则入栈

{

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

visited[p -> adjvex] = 1;

}

p = p -> next;

}

}

}

BFS广度优先遍历

1.将所有顶点的visited值置为0，访问初始顶点v0,置visited[0] = 1,v0入队。

2.检测队列是否为空，若队列为空，则迭代结束。

3.从队头取出一个顶点v,检测其每个邻接顶点w:如果w未被访问过，则

a. 访问w.

b. 将visited[w]值更新为1.

c. 将w入队。

4.执行步骤2

void BFS(AGraph \*g, int v)

{

int visited[g->n];

for(int i = 0; i < g->n; i++)

visited[i] = 0;

int queue[g->n];

int front = -1, rear = -1;

visited[v] = 1;

queue[++rear] = v;

while(front != rear)

{

int k = queue[++front]; //出队顶点

printf("%d ",k);

ArcNode\* p = g->adjlist[k].firstarc;

while(p != NULL)

{

if(visited[p -> adjvex] == 0)

{

queue[++rear] = p -> adjvex;

visited[p -> adjvex] = 1;

}

p = p -> next;

}

}

}

1. 采用邻接表存储结构，编写一个判断无向图中任意给定的两个顶点之间是否存在一条长度为k的简单路径的算法

dfs深搜

bool isHavePath(AGraph \*g, int u, int v, int k, int visited[])

{

if(k == 0 && u = v) //若已经找到

return true;

else if(k <= 0) //递归出口

return false;

visited[v] = 1; //将其标志为已访问

ArcNode \*p = g->adjlist[v].firstarc;

while(p != NULL)

{

if(visited[p->adjvex] == 0)

{

if(isHavePath(g,p->adjvex,v,k-1,visited))

return true;

else

visited[p->adjvex] = 0; //回溯到未访问状态

}

p = p -> next;

}

return false;

}

3．输出从顶点Vi到Vj的所有简单路径。

算法思想：

1.采用回溯的深度优先搜索算法，初始化结果数组为result，路径条数为size，每个路径的边数保存在数组column中，

初始化标记访问数组visited，保存路径的数组path，从顶点v开始DFS

2.将当前顶点加入路径，判断当前访问的顶点是否为顶点v，若是，则将当前路径保存在数组result中

若不是，则对当前顶点的边表进行回溯的深度优先搜索遍历， 在执行DFS过程后，将visited重新标记为未访问，继续寻找其他路径

3.输出result数组中的元素

void dfs(AGraph \*g, int u, int v, int\*\* result, int\* size, int\*\* column, int\* path, int\* visited, int index)

{

path[index] = u; //保存路径到path中

if(u == v) //若访问到顶点v，则将其加入结果数组

{

result[\*size] = (int\*)malloc(sizeof(int) \* (index + 1)); //开辟子路径

(\*column)[\*size] = index + 1; //子路径的空间

for(int i = 0; i <= index; i++) //复制路径

result[\*size][i] = path[i];

(\*size)++; //更新子路径个数

return;

}

ArcNode \*p = g->adjlist[u].firstarc;

while(p != NULL) //遍历整个边表

{

if(visited[p->adjvex] == 0)

{

visited[p->adjvex] = 1; //选中该结点作为路径进行搜索

dfs(g,p->adjvex,v,result,size,column,path,visited,index+1);

visited[p->adjvex] = 0; //回溯为未选中

}

p = p -> next;

}

}

int\*\* FindAllPath(AGraph \*g, int u, int v, int\* size, int\*\* column)

{

int\*\* result = (int\*\*)malloc(sizeof(int\*) \* maxnum); //结果数组

\*column = (int\*)malloc(sizeof(int) \* maxnum); //结果数组每一列的大小

int visited[g->n] = {0}, path[maxnum]; //初始化访问数组以及路径数组

\*size = 0;

visited[u] = 1;

dfs(g,u,v,result,size,column,path,visited,0);

return result;

}

写法2输出图G中从顶点u到v的所有简单路径

采用深度优先遍历的方法，在DFS算法的基础上改为G,u,u,path和d共5个参数。其中path数组存放路径初始为空，d表示路径长度初始为-1。

void FindPath(AGraph \*G, int u, int v, int path[], int d){

//d是到当前为止已走过的路径长度，调用时初值为-1；

int w, i;

ArcNode \*p;

d++;

path[d] = u;

visited[u] = 1;

if(u==v){

for(i = 0; i<=d; i++ )

printf(“%2d”, path[i]);

printf(“\n”);

}

p = G->adjlist[u].firstarc;

while(p != NULL){

w = p->adjvex; //w为顶点u的相邻顶点

if(visited[w] == 0)

FindPath(G, w, v, path, d);

p = p->nextarc; //p指向v的下一个相邻点

}

visited[u] = 0; //恢复环境，使该顶点可重新使用

}

1. 基于深度优先遍历和广度优先遍历判别有向图中是否存在由顶点Vi到顶点Vj的路径。

深度优先遍历

算法思想：

采用深度优先搜索，判断当前顶点是否为目标顶点，若是，则返回true

否则遍历该顶点的整个边表，对未访问的顶点进行递归判断

若在递归过程找到则返回true，否则递归结束返回false

bool dfs(AGraph \*g, int u, int v, int visited[])

{

if(u == v)

return true;

visited[u] = 1;

ArcNode\* p = g->adjlist[u].firstarc;

while(p != NULL)

{

if(visited[p->adjvex] == 0 && dfs(g,p->adjvex,v,visited))

return true;

p = p -> next;

}

return false;

}

void dfsmain(AGraph \*g, int u, int v)

{

int visited[g->n];

for(int i = 0; i < g->n; i++)

visited[i] = 0;

if(dfs(g,u,v,visited))

printf("有这条路径");

else

printf("无此路径");

}

广度优先遍历

算法思想：

1.采用广度优先搜索，将起始顶点u入队，将该顶点标记为已访问

2.不断执行出队，判断出队结点是否等于v，若等于则返回true，

否则继续将所有未访问的顶点入队，反复执行该过程，直至队空仍未返回时，则返回false

bool BFS(AGraph \*g, int u, int v)

{

int visited[g->n];

for(int i = 0; i < g->n; i++)

visited[i] = 0;

int queue[g->n];

int front = -1, rear = -1;

visited[u] = 1;

queue[++rear] = u;

while(front != rear)

{

int k = queue[++front]; //出队顶点

if(k == v)

return true;

ArcNode\* p = g->adjlist[k].firstarc;

while(p != NULL)

{

if(visited[p -> adjvex] == 0)

{

queue[++rear] = p -> adjvex;

visited[p -> adjvex] = 1;

}

p = p -> next;

}

}

return false;

}

1. 无权最短路径问题（不带权值）

算法思想：

1.初始化数组dist[i]用于记录顶点v到顶点i的最短路径长度，path[i]用于记录顶点v到顶点i的最短路径上顶点i的前驱顶点

2.初始化dist[v] = 0,其余均赋值为-1，创建队列，将顶点v入队

3.不断执行出队，遍历出队顶点u的边表，将所有未访问过的顶点入队，更新其路径长度为dist[u] + 1, 前驱顶点为u，反复执行该过程，直至队空

void ShortestPath(AGraph \*g, int v, int dist[], int path[])

{

int queue[maxnum]; //初始化队列

int front = -1, rear = -1;

for(int i = 0; i < g.n; i++) //初始化路径长度和前驱结点数组

{

dist[i] = -1;

path[i] = -1;

}

dist[v] = 0; //初始化路径长度为0

queue[++rear] = v; //顶点v入队

while(front != rear)

{

int u = queue[++front];

ArcNode\* p = g->adjlist[u].firstarc; //将u的所有未访问邻接顶点入队，修改path和dist值

while(p != NULL)

{

if(dist[p -> adjvex] == -1)

{

dist[p -> adjvex] = dist[u] + 1;

path[p -> adjvex] = u;

}

p = p -> next;

}

}

}

1. 设计一个算法，求不带权无向连通图G中距离顶点v最远的一个顶点

假设图G采用邻接表存储结构，采用广度优先遍历算法，从v出发，进行广度优先遍历时，最后一层的顶点距离v最远。遍历时利用队列逐层暂存各个顶点，队列中的最后一个顶点k一定在最后一层，因此，只要将该顶点作为结果即可。

int maxdist(AGraph \*G, int v){

ArcNode \*p;

int Qu[MAXV], front = 0, rear = 0;

int visited[MAXV], I, j, k;

for(i = 0; i<G->n; i++) //初始化访问标志数组

visited[i] = 0;

rear++; Qu[rear] = v; //顶点v进队

visited[v] = 1; //标记v已访问

while(rear !=front){

front = (front+1)%MAXV;

k = Qu[front]

p = G->adjlist[k].first; //找第一个邻接点

while(p!= NULL){ //所有未访问过的邻接点进队

j = p->adjvex;

if(visited[j] == 0){

visited[j] = 1;

rear = (rear+1)%MAXV;

}

p = p->nextarc; //找下一个邻接点

}

}

return k;

}

1. 正权单源最短路径Dijkstra（带权值）

每次选取待访问顶点中D值最小的顶点访问。另外，还增设了一个辅助数组s[],数组元素的初始值均为0，一旦顶点i被访问，则s[i] = 1

**伪代码理解**

***DShortestPath(Head, n, v. dist, path)***

***/\*计算v到其他各顶点的最短路径。n表示顶点个数，v表示起始顶点\*/***

***DSPath1.[初始化]***

***FOR i = TO n DO***

***(Path[i]←-1.***

***dist[i]←max.***

***s[i]←0.) //数组s[i]记录i是否被访问过***

***dis[v]←0.***

***s[v]←1.***

***p←adjacent(Head[v]).***

***u←v. //u为即将访问的顶点***

***DSPath2.[求从初始顶点v到其他各顶点的最短路径]***

***FOR j = 1 TO n-1 DO //循环（1）：求从初始顶点v到其他各顶点的最短路径***

***(WHILE p ≠ Λ DO //循环（2）：修改u邻接顶点的s[]值、path[]值和dist[]值***

***(k←VerAdj(p).***

***IF(s[k] ≠ 1AND dist[u] + cost(p)<dist[k]) THEN***

***(dist[k]←dist[u] + cost(p). path[k]←u.)***

***p←link(p).)***

***//循环（3）：确定即将被访问的顶点u***

***ldist←max.***

***FOR i = 1 TO n DO***

***IF(dist[i]<ldist AND s[i] = 0) THEN***

***(ldist←dist[i].u←i.)***

***s[u]←1. //访问u***

***p←adjacent(Head[u]).) //p为u的边链表的头指针▐***

**C语言代码**

void DshortestPath(AGraph \*g, int v)

{

int s[g->n], dist[g->n], path[g->n]; //初始化访问数组S[]，存储距离数组dist，路径数组path

for(int i = 0; i < g->n; i++) //初始化

{

path[i] = -1;

dist[i] = maxnum;

s[i] = 0;

}

s[v] = 1; //将顶点v设置为已经访问，起始距离为0

dist[v] = 0;

ArcNode\* p = g->adjlist[v].firstarc; //p为v的首个邻接边表

int u = v;

for(int i = 0; i < g->n-1; i++) //一共需要n-1次

{

while(p != NULL) //对边表进行遍历，更新path和dist

{

int k = p -> adjvex;

if(s[k] == 0 && dist[u] + p -> info < dist[k])

{

dist[k] = dist[u] + p -> info;

path[k] = u;

}

p = p -> next;

}

int ldist = maxnum;

for(int j = 0; j < g->n; j++)

if(dist[j] < ldist && s[j] == 0) //选出未访问过的最小值

{

ldist = dist[j];

u = j;

}

s[u] = 1; //将其选中

p = g -> adjlist[u].firstarc; //更新顶点所有邻接顶点的值

}

for(int i = 0; i < g -> n; i++)

{

printf("%d到%d的最短距离为%d",v,g->adjlist[i].data,dist[i]);

printf("\n");

}

}

1. 判断无向图是否是一颗树。

算法思想：

1.若无向图是一棵树，则该无向图是连通图，且顶点数 - 1 等于边数，一次访问就能访问到所有的顶点，初始化边数和顶点数为0

2.采用深度优先搜索，只执行一次遍历，若遍历到顶点时，则对顶点数进行更新，若遍历到边时，则对边进行更新

3.由于为无向图，因此同一条边会遍历两次，所以结束时的判断条件为访问到的顶点数是否等于图的顶点数，访问到的边数是否等于顶点数减一的二倍 若是，则返回true，若不是，则返回false

void DFS(AGraph \*g, int v,int \*countvnum,int \*countenum, int visited[])

{

visited[v] = 1; //将顶点v标记为已经访问

ArcNode\* p = g->adjlist[v].firstarc; //p为边表第一个结点

(\*countvnum)++; //访问到一个顶点，则更新顶点数

while(p != NULL)

{

(\*countenum)++; //每访问到一条边，就将边数加一，因为是无向图，所以会访问两遍同一条边

if(visited[p->adjvex] == 0)

DFS(g,p->adjvex,countvnum,countenum,visited);

p = p -> next;

}

}

//判断无向图是否是一棵树

bool isTree(AGraph \*g)

{

int visited[g->n]; //初始化标记访问数组visited

int countvnum = 0, countenum = 0; //初始化访问到的边数和顶点数为0

for(int i = 0; i < g->n; i++)

visited[i] = 0;

DFS(g,0,&countvnum,&countenum,visited);

if(countvnum == g->n && countenum == (g -> n - 1) \* 2) //判断一次遍历，访问到的顶点数是否等于图的顶点数，访问到的边数是否等于顶点数减一的二倍

return true;

else

return false;

}

1. 邻接表生成相应的逆邻接表。

算法思想：

1.初始化逆邻接表的顶点信息和边表信息

2.遍历邻接表，邻接表的顶点信息成为逆邻接表的边表信息，边表信息成为顶点信息，将其头插至逆邻接表

void getReverseAdjlist(AGraph \*g, AGraph \*r) //g为邻接表，r为逆邻接表

{

r.n = g.n; //初始化逆邻接表顶点数目

r.e = g.e; //初始化逆邻接表边数目

for(int i = 0; i < r.n; i++){

r.adjlist[i].data = g.adjlist[i].data; //构造逆邻接表顶点向量

r.adjlist[i].firstarc = NULL;

}

for(int i = 0; i < r.n; i++)

{

ArcNode \*p;

p = g.adjlist[i].firstarc; //邻接表的第一个顶点

while(p != NULL)

{

int j = p -> adjvex; //j为边表结点信息

ArcNode\* s = (ArcNode\*)malloc(sizeof(ArcNode)); //创建新的边表结点

s -> adjvex = i; //由于是逆邻接表，顶点信息与边表信息互换

s -> next = r.adjlist[j].firstarc; //将其头插至逆邻接表

r.adjlist[j].firstarc = s;

p = p -> next;

}

}

}

1. 图从邻接表转换成邻接矩阵。

void AdjmatrixToAdjList(AdjMatrix gm, AdjList gl){

for(i = 0; i<n; i++){

cin>>gl[i].vertex;

gl[i].firstarc = NULL;

} //初始化

for(i = 0;i<n;i++){

for(j = 0;j<n;j++){

if(gm[i][j] == 1){

p = (ArcNode\*)malloc(sizeof(ArcNode));

p->adjvex = j;

p->next = g[i].firstarc;

g[i].firstarc = p;

}

}

}

}

1. 判断图是否有回路

拓扑排序，求邻接表的入度。

1.统计入度，并将所有入度为0的顶点入栈，设置以及拓扑排序的顶点个数为count，初始化为0

2.执行出栈，访问出栈顶点，更新count，将出栈顶点的边表上所有顶点的入度减一，并将入度为0的入栈，循环该过程，直至栈空

3.判断是否访问到所有顶点，若是，则返回true，否则返回false

统计入度

void indegree(AGraph \*g, int d[])

{

for(int i = 0; i < g->n; i++) //初始化统计入度的数组d

d[i] = 0;

for(int i = 0; i < g->n; i++)

{

ArcNode\* p = g->adjlist[i].firstarc;

while(p != NULL) //遍历边表，为边表上的顶点信息入度加一

{

d[p->adjvex]++;

p = p -> next;

}

}

}

拓扑排序

bool TopologicalSort(AGraph \*g, int d[])

{

int stack[g->n]; //初始化堆栈

int top = -1;

int count = 0; //用于计数以及输出的顶点个数

for(int i = 0; i < g->n; i++)

if(d[i] == 0)

stack[++top] = i;

while(top != -1)

{

int k = stack[top--]; //出栈顶点

printf("%d",k); //输出该顶点

count++; //输出顶点加一

ArcNode \*p = g->adjlist[k].firstarc;

while(p != NULL) //遍历整个边表，更新入度

{

d[p->adjvex]--;

if(d[p->adjvex] == 0)

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

p = p -> next;

}

}

if(count == g->n)

return true;

else //若失败，则说明有向图中存在回路

return false;

}

深度优先搜索的拓扑排序

设u是v的祖先，则调用DFS访问u的过程中，必然会在这个过程结束之前递归地对v进行调用DFS访问，即v的DFS函数结束时间先于u的DFS结束时间。

从而可以考虑在DFS调用过程设置一个时间标记，在DFS调用结束时，对各定点计时，因此，祖先的结束时间必然大于子孙的结束时间。

1.采用深度优先搜索，初始化标记是否访问的数组visited，初始化表示结束时间的数组finishtime，设置变量time表示该顶点结束的时间，初始化为0

2.对边表结点进行深度优先搜索遍历，遍历结束后，对总时间进行更新，同时保存当前遍历顶点的时间

3.按照结束时间从大到小，得到一个拓扑序列

void dfs(AGraph \*g, int v, int visited[], int\* time, int finishtime[])

{

visited[v] = 1; //将当前顶点标记为已经访问

ArcNode\* p = g->adjlist[v].firstarc;

while(p != NULL)

{

if(visited[p->adjvex] == 0) //递归深度优先搜索边表顶点

{

dfs(g,p->adjvex,visited,time,finishtime);

}

p = p -> next;

}

\*time = \*time + 1; //当所有的边表顶点结束后，才会对时间进行更新

finishtime[v] = \*time; //更新当前遍历顶点的时间

}

void dfsmain(AGraph \*g)

{

int visited[g->n] = {0}, finishtime[g->n]; //visited用于标记是否访问，finishtime用于保存顶点的结束时间

int time = 0;

for(int i = 0; i < g->n; i++) //未防止不是连通图

if(visited[i] == 0)

dfs(g,i,visited,&time,finishtime);

int max = 0; //用于按照顶点大小进行输出

for(int i = 0; i < g->n; i++)

{

for(int j = 0; j < g->n; j++)

if(finishtime[j] >= finishtime[max])//选出最大值

max = j;

printf("%d",max);

finishtime[max] = -1; //输出后，将其标记为已经输出

}

}

1. 统计图中连通分量的个数

算法思想:

1.设置time表示执行深度优先搜索的次数，从第一个顶点开始DFS遍历，访问所有能够遍历到的顶点，将遍历到的顶点标记为已经访问

2.对每个顶点未访问过的顶点再进行一次深度优先搜索，其中每执行一次深度优先搜索，则连通个数加一

void dfs(AGraph \*g, int v, int visited[])

{

visited[v] = 1; //标记为已经访问

ArcNode\* p = g->adjlist[v].firstarc;

while(p != NULL) //遍历整个边表，递归访问

{

if(visited[p->adjvex] == 0)

dfs(g,p->adjvex,visited);

p = p -> next;

}

}

int CountConnected(AGraph \*g)

{

int num = 0, visited[g->n];

for(int i = 0; i < g->n; i++)

visited[i] = 0;

for(int i = 0; i < g->n; i++) //对每个顶点进行一次深度优先搜索

{

if(visited[i] == 0)

{

dfs(g,i,visited);

num++; //每执行一次深度优先搜索，则连通个数加一

}

}

return num;

}

写法2

方法一

采用深度优先搜索方法

void DFS(AGraph \*G, int v){

ArcNode \*p;

visited[v] = 1; //置已访问标记

printf(“%d”, v);

p = G->adjlist[v].firstarc; //p指向顶点p->adjvex顶点未访问，递归调用它

while(p != NULL){

if(visited[p->adjvex] == 0)

DFS(G, p->adjvex);

p = p->nextarc; //p指向顶点v的下一个相邻点

}

}

int ConnNum1(AGraph \*G){ //求图G连通分量

int i, num=0;

for(i=0; i<G->n; i++)

if(visited[i] == 0){

DFS(G, i); //调用DFS算法

num++;

}

return(num);

}

方法2

采用广度优先遍历

void BFS(AGraph \*G, int v)

ArcNode \*p;

int Qu[MAXV], front = 0, rear = 0;

int w, i;

for (i = 0; i<G-n; i++)

visited[i] = 0;

printf(“%d”, v);

visited[v] = 1;

rear = (rear+1)%MAXV;

Qu[rear] = v;

while(front != rear){

front = (front+1)%MAXV;

w = Qu[front];

p = G->Adjlist[w].firstarc; //找与顶点w邻接的第一个顶点

while(p != NULL){

if(visited[p->adjvex]==0){ //若当前邻接顶点未被访问

printf(“%2d”, p->adjvex);

visited[p->adjvex]=1;

rear = (rear+1)%MAXV;

Qu[rear] = p->adjvex; //该顶点进队

}

p = p->nextarc; //找下一个邻接顶点

}

}

printf(“\n”);

}

int ConnNum2(AGraph \*G){ //求图G的连通分量

int i, num = 0;

for (i = 0; i<G->n; i++)

visited[i] = 0;

for(i=0; i<G->n; i++)

if(visited[i] == 0){

BFS(G, i);

num++;

}

return (num);

}

1. （期末题）最小边数最短路最短路径可能有多条，边数最少的最短路径称为最小边数最短路，设计高效算法，在非负权图上求顶点v到顶点w的最小边数最短路。

算法思想：

1.采用回溯的深度优先搜索，设置path保存路径，min保存最短路径的边数，当前路径的边数为d

2.当前遍历的顶点v = w时，判断是否d < min，若是，则对min进行更新

3.对边表顶点进行回溯的深度优先搜索，以便反复访问同一结点，求出最短路径

void dfs(AGraph \*g, int v, int w, int visited[], int\* mind, int\* mincost, int path[], int d, int cost, int\* result)

{ path[d] = v;

if(v == w && cost < \*mincost && d < \*mind) //若找到了顶点w，且是边数更小的最短路径

{

\*mincost = cost; //更新最短路径和最短边数

\*mind = d;

for(int i = 0; i <= \*mind; i++) //复制最短路径到结果数组中

result[i] = path[i];

return;

}

ArcNode\* p = g->adjlist[v].firstarc;

while(p != NULL)

{

if(visited[p -> adjvex] == 0) //遍历边表

{

visited[p->adjvex] = 1;

dfs(g,p->adjvex,w,visited,mind,mincost,path,d+1,cost+p->info,result); //递归调用dfs，更新边数和路径长度

visited[p -> adjvex] = 0;

}

p = p -> next;

}

}

void dfsmain(AGraph \*g, int v, int w)

{

int path[g->n], visited[g->n] = {0}, result[g->n]; //path用于保存路径，visited用于标记是否访问

int mind = maxnum, mincost = maxnum; //初始化最小边数和最短路径长度

dfs(g,v,w,visited,&mind,&mincost,path,0,0,result);

for(int i = 0; i <= mind; i++) //输出最短路径

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

}

1. 给定连通G和G中的一个结点V，求G的支撑树（T为根，T的层次遍历次序恰好是以V为起点的G的某个广度优先遍历）

算法思想：

1.以N叉树的形式作为树的结构，创建树的根结点，值为v，创建N叉树结构的队列，将根结点入队，将根标记为已经访问

2.将出队树结点的值作为广度优先搜索出队顶点的值，执行出队，遍历该顶点的边表，将所有邻接的未访问顶点入队，将其创建为出队顶点的孩子结点

3.反复执行过程2，直至队空

struct TreeNode

{

int data;

int numChildren; //孩子数目

struct TreeNode\* children[maxnum]; //保存孩子的顶点数组

};

struct TreeNode\* BFSCreateTree(AGraph \*g, int v)

{

int visited[g->n] = {0};

visited[v] = 1;

struct TreeNode\* root = (struct TreeNode\*)malloc(sizeof(struct TreeNode));

//建立支撑树的根节点

root -> data = v;

struct TreeNode\* queue[maxnum];

int front = -1, rear = -1;

queue[++rear] = root;

while(front != rear)

{

struct TreeNode\* t = queue[++front]; //出队顶点

ArcNode\* p = g->adjlist[t->data].firstarc;

t -> numChildren = 0;

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

while(p != NULL)

{

if(visited[p -> adjvex] == 0)

{

t->children[t->numChildren]=(struct TreeNode\*)malloc(sizeof(struct TreeNode));

t -> children[t -> numChildren] -> data = p -> adjvex;

queue[++rear] = t -> children[t -> numChildren];

(t -> numChildren)++;

visited[p -> adjvex] = 1;

}

p = p -> next;

}

}

return root;

}

1. 设有一个正权有向图G = (V, E), w是G的一个顶点，w的偏心距定义为max{从u到w的最短路径长度，u属于V},路径长度指的是路径上各边权值之和，将G中的偏心距最小的顶点称为G的中心，设计一个函数返回图G的中心。

算法思想：设C是有向图G的邻接矩阵，求最小偏心度的顶点的步骤如下：

利用弗洛伊德算法求出每对顶点之间的最短路径矩阵A

对矩阵A求出每列i的最大值，得到顶点i的偏心度

在这n个顶点的偏心度中，求出最小偏心度的顶点k,便是图G的中心点。

int Centerp(MGraph g){

int [MAXV][MAXV],B[MAXV];

int i, j, k, m;

for(i=0; i<g.n; i++) //将邻接矩阵赋值给A

for(j = 0; j<g.n; j++)

A[i][j] = g.edges[i][j];

for(k=0; k<g.n; i++) //实现1的功能

for(i = 0; i<g.n; j++)

for(j = 0; g<g.n; j++)

if (A[i][k]+A[k][j]<A[i][j])

A[i][j] = A[i][k]+A[k][j];

for (j = 0; j<g.n; j++){ //实现2的功能

B[j] = A[0][j];

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

if(B[j]<[i][j])

B[j] = A[i][j];

}

k = 0; m = B[k]; //实现3的功能

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

if(B[i]<m){

m = b[i];

k = i;

}

return k; //返回k值

}

1. 判断图G是否有根

在有向图G中，如果r到G中的每个结点都有路径可达，则称结点r为G的根结点。编写一个算法判断有向图G是否有根，若有，则打印出所有根结点的值。

**基于邻接表的有向图**

算法思想:

1.对每个顶点进行一次深度优先搜索，每递归遍历一次，则访问的顶点个数加一

2.判断能否一次遍历到所有顶点，若能，则输出该根顶点的值，若不能，则将visited访问数组重置

void dfs(AGraph \*g, int v, int visited[], int\* num)

{

visited[v] = 1; //标记为已经访问

(\*num)++; //访问的顶点个数加一

ArcNode\* p = g->adjlist[v].firstarc;

while(p != NULL) //遍历整个边表，递归访问

{

if(visited[p->adjvex] == 0)

dfs(g,p->adjvex,visited,num);

p = p -> next;

}

}

void RootInGraph(AGraph \*g)

{

int num, visited[g->n];

for(int i = 0; i < g->n; i++) //对每个顶点进行一次深度优先搜索

{

num = 0;

for(int j = 0; j < g->n; j++) //每次深度优先搜索结束后，都需要对visited数组进行重置

visited[j] = 0;

dfs(g, i, visited, &num);

if(num == g->n) //若一次能访问所有顶点

printf("%d",i);

}

}

**基于邻接矩阵的有向同**

方法1 深度优先遍历

void MDFS(MGraph g, int v){} //基于邻接矩阵的深度优先遍历

int j;

visited[v] = 1;

for(j = 0; j<g.n; j++)

if(g.edges[v][j] != 0 && g.edges[v][j] != INF && visited[j] == 0)

MDFS(g, j); //与顶点i邻接的未访问的顶点j

}

int DGRoot1(MGraph g){

int I, j, k, n;

for (i=0; i<g.n; i++)

for(j = 0; j<g.n; i++)

visited[j] = 0;

MDFS(g, i);

n = 0;

for (k = 0; k <g.n; k++) // 累计从顶点i出发访问到的顶点个数

if(visited[k] ==1)

n++;

if(n == g.n) //若访问所有顶点，则顶点i为根

return(i);

}

return (-1);

}

方法2 广度优先遍历

void MBFS(MGraph g, int v){ //基于邻接矩阵的广度优先遍历

int Qu[MAXV], front = 0, rear = 0;

int i, j;

visited[v] = 1;

rear = (rear+1)%MAXV;

Qu[rear] = v;

while(front != rear){

front = (front+1)%MAXV;

i = Qu[front]; //出队顶点i

for(j=0; j<g.n; j++)

if(g.edges[i][j] != 0 && g.edges[i][j] != IFN) //找邻接点

if(visited[j] == 0){ //若当前邻接点未被访问

visited[j] = 1; //置该顶点已被访问的标志

rear = (rear+1)%MAXV;

Qu[rear] = j; //该顶点进队

}

}

printf(“\n”);

}

int DGRoot1(MGraph g){

int I, j, k, n;

for (i=0; i<g.n; i++)

for(j = 0; j<g.n; i++)

visited[j] = 0;

MBFS(g, i);

n = 0;

for (k = 0; k <g.n; k++) // 累计从顶点i出发访问到的顶点个数

if(visited[k] ==1)

n++;

if(n == g.n) //若访问所有顶点，则顶点i为根

return(i);

}

return (-1);

1. 自由树（即无环连通图）T = (V, E)的直径是树中所有顶点之间最短路径的最大值，试设计一个时间复杂度尽可能低的算法求T的直径

算法思想：

已知定理：在一个连通无向无环图中，以任意结点出发所能到达的最远结点，一定是该图直径的端点之一

1.第一次从任意顶点开始BFS，求出距离该顶点最远的一个顶点s，顶点s必然是自由树的直径的端点

2.从顶点s再次开始BFS，寻找到的距离该顶点最远的顶点即为直径，返回该直径即可

时间复杂度O(n + e)

int bfs(AGraph \*g, int v) //本算法用于求出直径

{

int queue[maxnum], dist[maxnum]; //初始化队列

int front = -1, rear = -1;

for(int i = 0; i < g->n; i++) //初始化路径长度

dist[i] = -1;

dist[v] = 0; //初始化路径长度为0

queue[++rear] = v; //顶点v入队

while(front != rear)

{

int u = queue[++front];

ArcNode\* p = g->adjlist[u].firstarc; //将u的所有未访问邻接顶点入队

while(p != NULL)

{

if(dist[p -> adjvex] == -1)

{

dist[p -> adjvex] = dist[u] + 1;

queue[++rear] = p -> adjvex;

}

p = p -> next;

}

}

int s = 0; //s为距离直径的起始顶点

for(int i = 0; i < g->n; i++)

if(dist[s] < dist[i])

s = i; //此时找出距离最远的顶点

for(int i = 0; i < g->n; i++) //再次初始化路径长度

dist[i] = -1;

dist[s] = 0; //初始化从顶点s开始的距离

queue[++rear] = s; //s入队

while(front != rear)

{

int u = queue[++front];

ArcNode\* p = g->adjlist[u].firstarc; //将u的所有未访问邻接顶点入队，修改path和dist值

while(p != NULL)

{

if(dist[p -> adjvex] == -1){

dist[p -> adjvex] = dist[u] + 1;

queue[++rear] = p -> adjvex;

}

p = p -> next;

}

}

int t = 0; //s为距离直径的起始顶点

for(int i = 0; i < g->n; i++)

if(dist[t] < dist[i]) //寻找距离最远的顶点

t = i;

return dist[t]; //此时s与t直径的距离即为直径

}

1. 假设图采用邻接表存储，设计一个算法，判断一个图是否连通，若连通则返回1，否则返回0.

方法1.

采用深度优先遍历，先给visit[]数组置初值0，然后从0顶点开始遍历该图。在一次遍历之后，所有顶点i的visit[i]均为1，则该图是连通的；否则不连通。

void DFS(AGraph \*G, int v){

ArcNode \*p;

visited[v] = 1;

printf(“%d”, v);

p = G->adjlist[v].firstarc; //p指向顶点v的第一条边的终结点

while(p != NULL){

if(visited[p->adjvex] == 0) //若p-adjvex顶点未访问，递归访问它

DFS(G, p->adjvex);

p = p->nextarc; //p指向顶点v的下一条边的终节点

}

}

int Gconnect1(AGraph \*G){ //判断无向图G的连通性

int i, flag = 1;

int visited[MAXV];

for (I = 0; i<G->n; i++)

if (visited[i] == 0){

flag = 0;

break;

}

return flag;

}

方法2.

采用广度优先遍历，先给visited[]数组置初值0，然后从0顶点开始遍历该图，在遍历一次之后，若所有顶点i的visited[i]均为1，则该图是连通的；否则不连通。

void BFS(AGraph \*G, int v){

ArcNode \*p;

int Qu[MAXV], front = 0, rear = 0; //定义循环队列

int w, I;

for (i = 0; i<G->n; i++) //访问标志数组初始化

visited[i] = 0;

printf(“%2d”, v); //输出被访问的顶点的编号

visited[v] = 1;

rear = (rear+1)%MAXV;

Qu[rear] = v;

while(front != rear){

front = (front+1)%MAXV;

w = Qu[front];

p = G->adjlist[w].firstarc; //找与顶点w邻接的第一个顶点

while(p != NULL){

if (visited[p->adjvex] == 0){ //若当前邻接顶点未被访问

printf(“%2d”, p->adjvex);

visited[p->adjvex] = 1; //置该顶点已被访问的标志

rear = (rear+1)%MAXV;

Qu[rear] = p->adjvex; //该顶点进队

}

p = p->nextarc; //找下一个邻接点

}

}

printf(“\n”);

}

int Gconnect2(AGraph \*G){ //判断无向图G的连通性

int i, flag;

int visited[MAXV];

for(I = 0; i<G->n; i++)

visited[i] =0;

BFS(G, 0);

for(I = 0; i<G->n; i++)

if(visited[i] == 0){

flag = 0;

break;

}

return flag;

}

1. (期末题，绿皮书)假设图G采用邻接表存储，设计一个算法，输出图G中顶点u到v的长度为k的所有简单路径

void PathALL(AGraph \*G, int u, int v, int k, int path[], int d){}

//d是当前为止已走过的路径长度，调用时初值为-1

int w, i;

ArcNode \*p;

visited[u] = 1;

d++;

path[d] = u;

if(u == v && d ==k){

for(i = 0; i<=d; i++)

printf(“%2d”, path[i]);

printf(“\n”);

}

p = G->adjlist[u].first[u].firstarc;

while(p != NULL){

w = p->adjvex; //w为顶点u的相邻顶点

if(visited[w] == 0) //若该顶点未标记访问，则递归访问之

PathAll(G, w, v, k, path, d);

p = p->nextarc;

}

visited[u] = 0; //恢复环境，使该顶点可重新使用

}

1. （期末题）已知有n个顶点的有向图的邻接表，求出度

1.设计算法计算图中出度为0的顶点个数。

2.设计在有向图中插入一条有向边<vi, vj>的算法

计算每个结点出度算法

扫描邻接表，对于顶点I, 记录各顶点的边数n，并输出

void Out Ds(AGraph \*G){

int I, n;

ArcNode \*p;

printf(“各顶点的出度：\n”);

for(i = 0; i<G->n; i++){

n = 0;

p = G->adjlist[i].firstarc;

while(p != NULL){

n++;

p = p->nextarc;

}

printf(“顶点%d: %d\n”, i, n);

}

}

1. （绿皮书，真题考过）求出无向无权连通图中，距离顶点v的最短路径长度为k的所有顶点，路径长度以边数为单位计算。

解法1.从顶点v出发，广度优先遍历的层次特性来求解，访问顶点时要知道一个顶点相对于v的层数，而每个顶点的层数是由其前驱决定的。

解法2可以从顶点v开始，每访问到一个顶点就根据其前驱的层数计算该顶点的层数，并将层数值与顶点编号一起入队和出队，算法可以使用两个队列，一个记录入队的顶点编号，另一个记录该顶点的层数，并保持两者同步。

红皮书答案

typedef struct VNode{

int lno; //距离初始顶点v0的距离

int v; //顶点编号

}Node; //邻接表的顶点类型

void bfsPrintNode(ALGraph \*g, int v0, int k){

Node Q[MaxSize]; //定义一个顶点类型的队列

int front = 0, rear = 0; //队头队尾

flag[v0] = 1; //访问后将其置为1

Q[rear++] = {0, v0}; //初始顶点v0入队

while(!isEmpty(Q)){

Node node = Q[front++]; //接收出队元素

for(ArcNode \*p = g.vexset[node.v].firstarc; p!= NULL; p=p->nextarc){

if(flag[p->adjvex] == 0){

visit(p->adjvex); //若边p对应的邻接顶点没被访问，则访问它

flag[p->adjvex] = 1;

Q[rear++] = {node.lno+1, p->adjvex}; //邻接顶点入队

}

}  
 }

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

if(Q[I].lno == k) //打印出距离顶点距离为k的所有顶点

printf(“%d”, Q[i].v);

}

}

1. （绿皮书，真题考过）设计一个算法，判断以邻接矩阵结构存储的有向图中是否存在一个简单回路，若存在，则以顶点序列的方式输出该回路。

上面给出了拓扑排序和基于dfs的拓扑排序。

这里是只利用深度优先遍历的思想，判断回路

采用红皮书写法

bool dfs(ALGraph \*g, int v){ //单次调用这个函数不一定能判断是否存在回路，得考虑非连通图

flag[v] = 1; //访问顶点v后标记

for(ArcNode \*p = g.vexset[v].firstarc; p != NULL; p = p->nextarc){

if(flag[p->adjvex] == 1) //若p->adjvex 访问过，则存在环

return 1;

if(flag[p->adjvex] == 0)

return dfs(g, p->adjvex); //递归的访问边p对应的邻接顶点

}

return false;

}

//考虑到非连通图

bool isHaveRing(ALGraph g){

for(int i = 0; i<g.vexnum; ++i){

for(int j=0; j<g.vexnum; ++j)

flag[j] = 0; //每次从新的顶点进行遍历的时候，都得重新将flag数组清0

if(dfs(g, i) == true) //只有一个子图存在环，则图g存在环

return true;

}

return false;

}

1. （绿皮书）假设图采用邻接表存储，编写一个算法，利用深度优先遍历求出无向图中通过定点v的简单回路。采用深度优先遍历的方法，只需将起始顶点和终止顶点均设为v，输出path的条件为 w==v&&d>0

void CyclePath(AGraph \*G, int u, int v, int path[], int d){

int w, I;

ArcNode \*p;

visited[u] = 1;

d++; //路径长度增1

path[u] = u; //将当前顶点添加到路径中

p = G->adjlist[u].firstarc;

while(p != NULL){

w = p->adjvex; //w为顶点u的相邻顶点

if(visited[w] == 0)

CyclePath(G, w, v, path, d);

else if(w ==v && d>0){ //找到一条路径则输出

for(i = 0; i<=d; i++)

printf(“%2d”, path[i]);

printf(“%2d\n”, v);

}

p = p->nextarc;

visited[u] = 0; //恢复环境，使该顶点可重新使用

}

void CycleAll(AGraph \*G, int v){ //经过顶点v的所有回路

int path[MAXV];

printf(“经过%d顶点的所有回路如下：\n”, v)

CyclePath(G,v, v, path, -1);

}