**关于单链表**

设计一个算法void reverse(LinkList \*&L)，将一个带头节点的循环单链表L（至少含有两个以上节点）中所有节点逆置。

先构造一个空的循环单链表，用p扫描所有数据节点，循环将\*p节点采用头插法插入到L中。算法如下：

void reverse(LinkList \*&L)

{ LinkList \*p=L->next,\*q;

L->next=L;

while (p!=L)

{ q=p->next;

p->next=L->next;

L->next=p;

p=q;

}

}

设A和B是两个单链表（带头结点），其中元素递增有序。设计一个算法由A和B中公共元素产生单链表C，要求不破坏A、B的结点。

本题采用尾插法建立单链表C，由于算法要求不破坏A、B的结点，所以需要通过比较复制的方式产生单链表C。对应的算法如下：

void common(LinkList \*A,LinkList \*B,LinkList \*&C)

{ LinkList \*p=A->next,\*q=B->next,\*r,\*s;

C=(LinkList \*)malloc(sizeof(LinkList));

r=C; //让r始终指向C的尾结点

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

{ if (p->data<q->data)

p=p->next;

else if (p->data>q->data)

q=q->next;

else //p->data==q->data：为公共元素

{ s=(LinkList \*)malloc(sizeof(LinkList));

s->data=p->data; //复制产生结点\*s

r->next=s;r=s; //将\*s链接到C上

p=p->next;

q=q->next;

}

}r->next=NULL;}

设计一个算法delminnode(LinkList \*&L)，在带头结点的单链表L中删除所有结点值最小的结点（可能有多个结点值最小的结点）。

用p从头至尾扫描单链表，pre指向\*p结点的前驱，用minp保存值最小的结点指针，minpre指向\*minp结点的前驱。一面扫描，一面比较，将最小值的结点放到\*minp中。算法如下：

void delminnode(LinkList \*&L)

{

LinkList \*pre=L,\*p=pre->next,\*minp=p,\*minpre=pre;

ElemType mindata=p->data;

while (p!=NULL && p->data<mindata)

{ mindata=p->data;

p=p->next;

}

p=pre->next;

while (p!=NULL)

{

if (p->data==mindata)

{ pre->next=p->next;

free(p);

}

pre=pre->next;

p=pre->next;

}

}

有3个带头结点并且结点值递增的单链表h1、h2和h3，它们的结点个数分别为m、n和k，单链表的结点类型如下：

typedef struct node

{ int data;

struct node \*next;

} LinkList;

设计一个算法：

void merge(LinkList \*&h, LinkList \*h1,LinkList \*h2,LinkList \*h3)

其功能是将h1、h2和h3的所有结点归并成一个新的递增单链表h，要求空间复杂度为O(1)，时间复杂度为O(m+n+k)。

对应的算法如下：

void merge1(LinkList \*&h, LinkList \*h1,LinkList \*h2)

{ /\*两个单链表归并\*/

LinkList \*p1=h1->next,\*p2=h2->next,\*t;

h=h1;t=h;

while (p1!=NULL && p2!=NULL)

{

if (p1->data<p2->data)

{ t->next=p1; t=p1; p1=p1->next; }

else

{ t->next=p2; t=p2; p2=p2->next; }

}

if (p1!=NULL) t->next=p1;

if (p2!=NULL) t->next=p2;

}

void merge(LinkList \*&h, LinkList \*h1,LinkList \*h2,LinkList \*h3)

{merge1(h,h1,h2);

merge1(h,h,h2);}

已知单链表L(带头结点)是一个非有序表，设计一个算法，删除表中data值在大于或等于min小于或等于max之间的结点(若表中有这样的结点)，同时释放被删结点的空间，这里min和max是两个给定的参数。并分析算法的时间复杂度。

算法如下：

void delnode(LinkList \*&L,ElemType min,Elemtype max)

{

LinkList \*p=L->data,\*pre=L,\*q;

while (p!=NULL)

{ if (p->data>=min && p->data<=max) //满足条件删除之

{ q=p;

pre->next=p->next;

p=p->next;

free(q);

}

else

{ pre=p;

p=p->next;

}

}

}

算法的时间复杂度为O(n)。

假设二叉树采用二叉链存储结构，t指向根结点，p所指结点为任一给定的结点，设计一个算法，输出从根结点到p所指结点之间路径。

算法如下：

void path(BTree\* t,BTree \*p)

{

BTree\* St[MaxSize],\*s;

int tag[MaxSize];

int top=-1,i;

s=t;

do

{ while(s!=NULL) //扫描左结点，入栈

{ top++;

St[top]=s;

tag[top]=0;

s=s->lchild;

}

if(top>-1)

{ if(tag[top]==1) //左右结点均已访问过，则要访问该结点

{ if (St[top]==p) //该结点就是要找的结点

{ printf("路径:"); //输出从栈底到栈顶的元素构成路径

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

printf("%c ",St[i]->data);

printf("\n");

break;

}

top--;

}

else

{ s=St[top];

if(top>0)

{ s=s->rchild; //扫描右结点

tag[top]=1; //表示当前结点的右子树已访问过

}

}

}

} while(s!=NULL||top!=-1);

}

有一个学生成绩线性表（a1,a2,…,an）用带头结点的单链表h存储（ai均为整数），编写一个算法，由其中所有成绩不及格的元素构成的另一个单链表h1，要求不破坏h的结点，该算法格式为：

void fun(LinkList \*h,LinkList \*&h1)

算法如下：

void fun(LinkList \*h,LinkList \*&h1)

{

LinkList \*p=h->next,\*q,\*r;

h1=(LinkList \*)malloc(sizeof(LinkList));

r=h1;

while (p!=NULL)

{

if(p->data <60){

q=(LinkList \*)malloc(sizeof(LinkList));

q->data=p->data;

r->next=q;

r=q;

}

p=p->next;

}

r->next=NULL;

}

**关于双链表**

设计一个算法，将一个带头结点的数据域依次为a1，a2，…，an（n≥3）的双链表的所有结点逆置，即第一个结点的数据域变为an，第二个结点的数据域变为an-1，…，最后一个结点的数据域为a1。

采用前插法建表，算法如下：

void Reverse(DLinkList \*&h)

{

DLinkList \*p=h->next,\*q=p->next;

h->next=NULL;

while (p!=NULL)

{ p->next=h->next; /\*将p所指结点插入到头结点之后\*/

p->prior=h;

p=q;

q=p->next;

}

}

**关于二叉树**

假设二叉树采用二叉链存储结构存储，其中所有节点值均为正整数，设计一个算法int max(BTNode \*b)，用于计算二叉树b中的最大节点值并返回该最大值。

递归算法如下：

int max(BTNode \*b)

{

if (b!=NULL)

{

int m,m1,m2;

if (b->lchild==NULL && b->rchild==NULL) //\*b为叶节点

return b->data;

m1=max(b->lchild); //求左子树的最大值

m2=max(b->rchild); //求右子树的最大值

if (m1>m2) m=m1;

else m=m2;

if (b->data>m) m=b->data)

return m;

}

}

假设二叉树采用二叉链存储结构，其中所有结点的值为正整数，设计一个算法求该二叉树中所有叶子结点的值之和。

对应的算法如下：

void leafsum(BTNode \*b,int &s)

{ //初始调用时s置为0

if (b!=NULL)

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

s+=b->data;

f(b->lchild,s); //递归求左子树的所有叶子结点值之和

f(b->rchild,s); //递归求右子树的所有叶子结点值之和

}

else s=0;

}

假设二叉树采用二叉链存储结构存储，设计一个算法copy(BTNode \*b,BTNode \*&t)，由二叉树b复制成另一棵二叉树t。

递归算法如下：

void copy(BTNode \*b,BTNode \*&t)

{

BTNode \*l,\*r;

if (b==NULL) t=NULL;

else

{

t=(BTNode \*)malloc(sizeof(BTNode));

copy(b->lchild,l);

copy(b->rchild,r);

t->lchild=l;

t->rchild=r;

}

}

以二叉链作为二叉树的存储结构，编写一个算法，将二叉树T复制到T1中，要求不破坏T的结点，该算法格式为：

void copytree(BTNode \*T，BTNode \*&T1)

算法如下：

void copytree(BTNode \*T，BTNode \*&T1)

{

if (T!=NULL)

{

T1=(BTNode \*)malloc(sizeof(BTNode));

T1->data=T->data;

copytree(T->lchild，T1->lchild);

copytree(T->rchild，T1->rchild);

}

else

T1=NULL;

}

假设一个仅包含二元运算符加、减、乘和除的算术表达式以二叉链存储结构进行存储，该二叉链的结点类型如下：

typedef struct node

{ int data;

struct node \*lchild,\*rchild;

} BTNode;



图3 表达式二叉树

假设在构造二叉链时已考虑了运算符的优先级，例如2+3\*4表达式对应的二叉链如图3所示。设计一个算法：

float ExpValue(BTNode \*b)

其功能计算二叉树b对应的表达式值。

对应的算法如下：

float ExpValue(BTNode \*b) //计算表达式值

{

float lv,rv,value=0;

if (b!=NULL)

{　if (b->data!='+' && b->data!='-' && b->data!='\*' && b->data!='/')

return(b->data);

lv=ExpValue(b->lchild);

rv=ExpValue(b->rchild);

switch(b->data)

{

case '+':value=lv+rv;break;

case '-':value=lv-rv;break;

case '\*':value=lv\*rv;break;

case '/':if (rv!=0) value=lv/rv;

else exit(0);

break;

}

}

return(value);

}

一棵具有n个结点的完全二叉树以顺序方式存储在数组A[1..n]中。设计一个算法构造该二叉树的二叉链存储结构。

对于以顺序方式存储在数组A中的一棵完全二叉树，结点A[i]的左孩子为A[2i]，右孩子为A[2i+1]。采用递归方式创建该二叉树的链接存储表示。对应的算法如下：

void Ctree(BTNode \*&b,ElemType A[],int i,int n)

{

if (i>n)

b=NULL;

else

{ b=(BTNode \*)malloc(sizeof(BTNode));

b->data=A[i];

Ctree(b->lchild,A,2\*i,n); /\*构造\*b的左子树\*/

Ctree(b->rchild,A,2\*i+1,n); /\*构造\*b的右子树\*/

}

}

**关于图**

假设一个连通图采用邻接表存储结构表示，当给定某种邻接表后，采用深度优先方法可以得到多种深度优先遍历序列。设计一个算法，在给定的邻接表G上输出从顶点v出发的所有深度优先遍历序列。

对应的算法如下：

int visited[MAXV]=[0]; //全局变量

void DFSALL(ALGraph \*G,int v,int path[],int d)

{ ArcNode \*p;

visited[v]=1; //置已访问标记

path[d]=v;

d++;

if (d==G->n) //如果已访问所有顶点，则输出访问序列

{ for (int k=0;k<d;k++)

printf("%2d",path[k]);

printf("\n");

}

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

while (p!=NULL)

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

DFSALL(G,p->adjvex,path,d);

p=p->nextarc; //p指向顶点v的下一条边

}

visited[v]=0;

}