**1.19 试编写算法，计算的值并存入数组a[0..arrsize-1]的第i-1个分量中(i=1,2,…,n)。假设计算机中允许的整数最大值为maxint，则当n>arrsize或对某个，使时，应按出错处理。注意选择你认为较好的出错处理方法。**

**解：**

#include<iostream.h>

#include<stdlib.h>

#define MAXINT 65535

#define ArrSize 100

int fun(int i);

int main()

{

int i,k;

int a[ArrSize];

cout<<"Enter k:";

cin>>k;

if(k>ArrSize-1) exit(0);

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

if(i==0) a[i]=1;

else{

if(2\*i\*a[i-1]>MAXINT) exit(0);

else a[i]=2\*i\*a[i-1];

}

}

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

if(a[i]>MAXINT) exit(0);

else cout<<a[i]<<" ";

}

return 0;

}

**1.20 试编写算法求一元多项式的值的值，并确定算法中每一语句的执行次数和整个算法的时间复杂度。注意选择你认为较好的输入和输出方法。本题的输入为，和，输出为。**

**解：**

#include<iostream.h>

#include<stdlib.h>

#define N 10

double polynomail(int a[],int i,double x,int n);

int main()

{

double x;

int n,i;

int a[N];

cout<<"输入变量的值x:";

cin>>x;

cout<<"输入多项式的阶次n:";

cin>>n;

if(n>N-1) exit(0);

cout<<"输入多项式的系数a[0]--a[n]:";

for(i=0;i<=n;i++) cin>>a[i];

cout<<"The polynomail value is "<<polynomail(a,n,x,n)<<endl;

return 0;

}

double polynomail(int a[],int i,double x,int n)

{

if(i>0) return a[n-i]+polynomail(a,i-1,x,n)\*x;

else return a[n];

}

本算法的时间复杂度为o(n)。

**2.19 已知线性表中的元素以值递增有序排列，并以单链表作存储结构。试写一高效的算法，删除表中所有值大于mink且小于maxk的元素（若表中存在这样的元素），同时释放被删结点空间，并分析你的算法的时间复杂度（注意，mink和maxk是给定的两个参变量，它们的值可以和表中的元素相同，也可以不同）。**

Status ListDelete\_L(LinkList &L,ElemType mink,ElemType maxk)

{

LinkList p,q,prev=NULL;

if(mink>maxk)return ERROR;

p=L;

prev=p;

p=p->next;

while(p&&p->data<maxk){

if(p->data<=mink){

prev=p;

p=p->next;

}

else{

prev->next=p->next;

q=p;

p=p->next;

free(q);

}

}

return OK;

}

**2.24 假设有两个按元素值递增有序排列的线性表A和B，均以单链表作存储结构，请编写算法将A表和B表归并成一个按元素值递减有序（即非递增有序，允许表中含有值相同的元素）排列的线性表C，并要求利用原表（即A表和B表）的结点空间构造C表。**

**解：**

// 将合并逆置后的结果放在C表中，并删除B表

Status ListMergeOppose\_L(LinkList &A,LinkList &B,LinkList &C)

{

LinkList pa,pb,qa,qb;

pa=A;

pb=B;

qa=pa; // 保存pa的前驱指针

qb=pb; // 保存pb的前驱指针

pa=pa->next;

pb=pb->next;

A->next=NULL;

C=A;

while(pa&&pb){

if(pa->data<pb->data){

qa=pa;

pa=pa->next;

qa->next=A->next; //将当前最小结点插入A表表头

A->next=qa;

}

else{

qb=pb;

pb=pb->next;

qb->next=A->next; //将当前最小结点插入A表表头

A->next=qb;

}

}

while(pa){

qa=pa;

pa=pa->next;

qa->next=A->next;

A->next=qa;

}

while(pb){

qb=pb;

pb=pb->next;

qb->next=A->next;

A->next=qb;

}

pb=B;

free(pb);

return OK;

}

**2.29 已知A，B和C为三个递增有序的线性表，现要求对A表作如下操作：删去那些既在B表中出现又在C表中出现的元素。试对顺序表编写实现上述操作的算法，并分析你的算法的时间复杂度（注意：题中没有特别指明同一表中的元素值各不相同）。**

**解：**

// 在A中删除既在B中出现又在C中出现的元素，结果放在D中

Status ListUnion\_Sq(SqList &D,SqList &A,SqList &B,SqList &C)

{

SqList Temp;

InitList\_Sq(Temp);

ListCross\_L(B,C,Temp);

ListMinus\_L(A,Temp,D);

}

**2.30 要求同2.29题。试对单链表编写算法，请释放A表中的无用结点空间。**

**解：**

// 在A中删除既在B中出现又在C中出现的元素，并释放B、C

Status ListUnion\_L(LinkList &A,LinkList &B,LinkList &C)

{

ListCross\_L(B,C);

ListMinus\_L(A,B);

}

// 求集合A-B，结果放在A表中，并删除B表

Status ListMinus\_L(LinkList &A,LinkList &B)

{

LinkList pa,pb,qa,qb,pt;

pa=A;

pb=B;

qa=pa; // 保存pa的前驱指针

qb=pb; // 保存pb的前驱指针

pa=pa->next;

pb=pb->next;

while(pa&&pb){

if(pb->data<pa->data){

pt=pb;

pb=pb->next;

qb->next=pb;

free(pt);

}

else

if(pb->data>pa->data){

qa=pa;

pa=pa->next;

}

else{

pt=pa;

pa=pa->next;

qa->next=pa;

free(pt);

}

}

while(pb){

pt=pb;

pb=pb->next;

qb->next=pb;

free(pt);

}

pb=B;

free(pb);

return OK;

}

**2.32 已知有一个单向循环链表，其每个结点中含三个域：pre，data和next，其中data为数据域，next为指向后继结点的指针域，pre也为指针域，但它的值为空，试编写算法将此单向循环链表改为双向循环链表，即使pre成为指向前驱结点的指针域。**

**解：**

// 建立一个空的循环链表

Status InitList\_DL(DuLinkList &L)

{

L=(DuLinkList)malloc(sizeof(DuLNode));

if(!L) exit(OVERFLOW);

L->pre=NULL;

L->next=L;

return OK;

}

// 向循环链表中插入一个结点

Status ListInsert\_DL(DuLinkList &L,ElemType e)

{

DuLinkList p;

p=(DuLinkList)malloc(sizeof(DuLNode));

if(!p) return ERROR;

p->data=e;

p->next=L->next;

L->next=p;

return OK;

}

// 将单循环链表改成双向链表

Status ListCirToDu(DuLinkList &L)

{

DuLinkList p,q;

q=L;

p=L->next;

while(p!=L){

p->pre=q;

q=p;

p=p->next;

}

if(p==L) p->pre=q;

return OK;

}

**2.37 设以带头结点的双向循环链表表示的线性表。试写一时间复杂度O(n)的算法，将L改造为。**

**解：**

// 将双向链表L=(a1,a2,...,an)改造为(a1,a3,...,an,...,a2)

Status ListChange\_DuL(DuLinkList &L)

{

int i;

DuLinkList p,q,r;

p=L->next;

r=L->pre;

i=1;

while(p!=r){

if(i%2==0){

q=p;

p=p->next;

// 删除结点

q->pre->next=q->next;

q->next->pre=q->pre;

// 插入到头结点的左面

q->pre=r->next->pre;

r->next->pre=q;

q->next=r->next;

r->next=q;

}

else p=p->next;

i++;

}

return OK;

}

**2.38 设有一个双向循环链表，每个结点中除有pre，data和next三个域外，还增设了一个访问频度域freq。在链表被起用之前，频度域freq的值均初始化为零，而每当对链表进行一次Locate(L,x)的操作后，被访问的结点（即元素值等于x的结点）中的频度域freq的值便增1，同时调整链表中结点之间的次序，使其按访问频度非递增的次序顺序排列，以便始终保持被频繁访问的结点总是靠近表头结点。试编写符合上述要求的Locate操作的算法。**

**解：**

DuLinkList ListLocate\_DuL(DuLinkList &L,ElemType e)

{

DuLinkList p,q;

p=L->next;

while(p!=L && p->data!=e) p=p->next;

if(p==L) return NULL;

else{

p->freq++;

// 删除结点

p->pre->next=p->next;

p->next->pre=p->pre;

// 插入到合适的位置

q=L->next;

while(q!=L && q->freq>p->freq) q=q->next;

if(q==L){

p->next=q->next;

q->next=p;

p->pre=q->pre;

q->pre=p;

}

else{

// 在q之前插入

p->next=q->pre->next;

q->pre->next=p;

p->pre=q->pre;

q->pre=p;

}

return p;

}

}

**3.20 假设以二维数组g(1…m, 1…n)表示一个图像区域，g[i,j]表示该区域中点(i,j)所具颜色，其值为从0到k的整数。编写算法置换点(i0,j0)所在区域的颜色。约定和(i0,j0)同色的上、下、左、右的邻接点为同色区域的点。**

**解：**

#include <iostream.h>

#include <stdlib.h>

typedef struct{

int x;

int y;

}PosType;

typedef struct{

int Color;

int Visited;

PosType seat;

}ElemType;

#include "d:\VC99\Stack.h"

#define M 8

#define N 8

ElemType g[M][N];

void CreateGDS(ElemType g[M][N]);

void ShowGraphArray(ElemType g[M][N]);

void RegionFilling(ElemType g[M][N],PosType CurPos,int NewColor);

int main()

{

CreateGDS(g);

ShowGraphArray(g);

PosType StartPos;

StartPos.x=5;

StartPos.y=5;

int FillColor=6;

RegionFilling(g,StartPos,FillColor);

cout<<endl;

ShowGraphArray(g);

return 0;

}

void RegionFilling(ElemType g[M][N],PosType CurPos,int FillColor)

{

Stack s;

InitStack(s);

ElemType e;

int OldColor=g[CurPos.x][CurPos.y].Color;

Push(s,g[CurPos.x][CurPos.y]);

while(!StackEmpty(s)){

Pop(s,e);

CurPos=e.seat;

g[CurPos.x][CurPos.y].Color=FillColor;

g[CurPos.x][CurPos.y].Visited=1;

if(CurPos.x<M &&

!g[CurPos.x+1][CurPos.y].Visited &&

g[CurPos.x+1][CurPos.y].Color==OldColor

)

Push(s,g[CurPos.x+1][CurPos.y]);

if(CurPos.x>0 &&

!g[CurPos.x-1][CurPos.y].Visited &&

g[CurPos.x-1][CurPos.y].Color==OldColor

)

Push(s,g[CurPos.x-1][CurPos.y]);

if(CurPos.y<N &&

!g[CurPos.x][CurPos.y+1].Visited &&

g[CurPos.x][CurPos.y+1].Color==OldColor

)

Push(s,g[CurPos.x][CurPos.y+1]);

if(CurPos.y>0 &&

!g[CurPos.x][CurPos.y-1].Visited &&

g[CurPos.x][CurPos.y-1].Color==OldColor

)

Push(s,g[CurPos.x][CurPos.y-1]);

}

}

void CreateGDS(ElemType g[M][N])

{

int i,j;

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

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

g[i][j].seat.x=i;

g[i][j].seat.y=j;

g[i][j].Visited=0;

g[i][j].Color=0;

}

for(i=2;i<5;i++)

for(j=2;j<4;j++)

g[i][j].Color=3;

for(i=5;i<M-1;i++)

for(j=3;j<6;j++)

g[i][j].Color=3;

}

void ShowGraphArray(ElemType g[M][N])

{

int i,j;

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

for(j=0;j<N;j++)

cout<<g[i][j].Color;

cout<<endl;

}

}

**3.28 假设以带头结点的循环链表表示队列，并且只设一个指针指向队尾元素结点（注意不设头指针），试编写相应的队列初始化、入队列何处队列的算法。**

**解：**

typedef int ElemType;

typedef struct NodeType{

ElemType data;

NodeType \*next;

}QNode,\*QPtr;

typedef struct{

QPtr rear;

int size;

}Queue;

Status InitQueue(Queue& q)

{

q.rear=NULL;

q.size=0;

return OK;

}

Status EnQueue(Queue& q,ElemType e)

{

QPtr p;

p=new QNode;

if(!p) return FALSE;

p->data=e;

if(!q.rear){

q.rear=p;

p->next=q.rear;

}

else{

p->next=q.rear->next;

q.rear->next=p;

q.rear=p;

}

q.size++;

return OK;

}

Status DeQueue(Queue& q,ElemType& e)

{

QPtr p;

if(q.size==0)return FALSE;

if(q.size==1){

p=q.rear;

e=p->data;

q.rear=NULL;

delete p;

}

else{

p=q.rear->next;

e=p->data;

q.rear->next=p->next;

delete p;

}

q.size--;

return OK;

}

**3.32 试利用循环队列编写求k阶菲波那契序列中前n+1项的算法，要求满足：而，其中max为某个约定的常数。（注意：本题所用循环队列的容量仅为k，则在算法执行结束时，留在循环队列中的元素应是所求k阶菲波那契序列中的最后k项）**

**解：**

int Fibonacci(int k,int n)

{

if(k<1) exit(OVERFLOW);

Queue q;

InitQueue(q,k);

ElemType x,e;

int i=0;

while(i<=n){

if(i<k-1){

if(!EnQueue(q,0)) exit(OVERFLOW);

}

if(i==k-1){

if(!EnQueue(q,1)) exit(OVERFLOW);

}

if(i>=k){

// 队列求和

x=sum(q);

DeQueue(q,e);

EnQueue(q,x);

}

i++;

}

return q.base[(q.rear+q.MaxSize-1)%q.MaxSize];

}

0、1、1、2、3、5、8、13、21、34、

**6.42 编写递归算法，计算二叉树中叶子结点的数目。**

**解：**

// 求二叉树中叶子结点的数目

Status POLeafNodeNum(int& i,BiTree& T)

{

if(T){

if(!T->lchild && !T->rchild) i++;

POLeafNodeNum(i,T->lchild);

POLeafNodeNum(i,T->rchild);

}

return OK;

}

**6.45 编写递归算法，对于二叉树中每一个元素值为x的结点，删去以它为根的子树，并释放相应空间。**

**解：**

// 删除以元素值为x的结点为根的子树

Status DelChildTree(BiTree& T,TElemType x)

{

if(T){

if(T->data==x){

DelBTree(T);

T=NULL;

return OK;

}

else{

if(DelChildTree(T->lchild,x))

return OK;

else{

if(DelChildTree(T->rchild,x))

return OK;

else return ERROR;

}

}

}

else return ERROR;

}

// 删除二叉树

Status DelBTree(BiTree& T)

{

if(T){

DelBTree(T->lchild);

DelBTree(T->rchild);

delete T;

return OK;

}

else return ERROR;

}

**6.47 编写按层次顺序（同一层自左至右）遍历二叉树的算法。**

**解：**

typedef BiTree QElemType;

#include "c:\Yin\include\Queue.h"

Status LevelOrderTraverse(BiTree& T,Status (\*Visit)(TElemType e))

{

QElemType p;

Queue q;

InitQueue(q);

if(T) EnQueue(q,T);

while(!QueueEmpty(q)){

DeQueue(q,p);

Visit(p->data);

if(p->lchild) EnQueue(q,p->lchild);

if(p->rchild) EnQueue(q,p->rchild);

}

return OK;

}

**6.49 编写算法判别给定二叉树是否为完全二叉树**

**解：**

Status CompleteBiTree(BiTree& T)

{

int d;

if(T){

d=BiTDepth(T->lchild)-BiTDepth(T->rchild);

if(d<0 || d>1) return ERROR;

else{

if(CompleteBiTree(T->lchild) &&

CompleteBiTree(T->rchild)) return OK;

else return ERROR;

}

}

else return OK;

}

**6.60 试编写算法，对一棵以孩子-兄弟链表表示的树统计叶子的个数**

**解：**

int LeafNum(CSTree& T)

{

if(T){

if(!T->firstchild)

return 1+LeafNum(T->nextsibling);

else

return LeafNum(T->firstchild)+LeafNum(T->nextsibling);

}

else return 0;

}

**6.62 对以孩子-兄弟链表表示的树编写计算树的深度的算法**

**解：**

// 树的深度

int Depth(CSTree& T)

{

int d1,d2;

if(T){

d1=1+Depth(T->firstchild);

d2=Depth(T->nextsibling);

return d1>d2?d1:d2;

}

else return 0;

}