线型数据结构

单链表的插入查找和删除

```
//单链表的插入删除和遍历
//这种是直接用数组模拟链表存的数据，比较容易实现
#include <iostream>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
#define OK 1;
#define ERROR -1
using namespace std;
typedef string ElemType;
struct SqList
{
    ElemType elem[LIST_INIT_SIZE];
    int length;
};
int InitList_Sq(SqList &L)
{
    L.length = 0;
    return OK;
}
int ListInsert_sq(SqList &L, int i, string e)
{
    for(int j=L.length; j>=i; j--)
    {
        L.elem[j] = L.elem[j-1];
    }
    L.elem[i-1] = e;
    L.length++;
    return OK;
}
int LocateElem_Sq(SqList &L, string e)
{
    int i = 0;
    while(i<L.length && L.elem[i] != e)
        ++i;
    if(i < L.length)
        return i+1;
    else
        return 0;
}
int ListDelete_Sq(SqList &L, int i, string e)
{
    if(i<1 || i>L.length)
        return ERROR;
    for(int j=i-1; j<L.length-1; j++)
    {
        L.elem[j] = L.elem[j+1];
    }
    --L.length;
    return OK;
}
void show(SqList L)
{
    for(int i=0; i<L.length; i++)
    {
        cout<<L.elem[i]<<" ";
```

```
    }
    printf("\n");
}
int main()
{
    SqList L;
    int num;
    string name;
    string task;
    InitList_Sq(L);
    while(cin>>task)
    {
        switch(task[0])
        {
        case 'i':
            scanf("%d", &num);
            cin>>name;
            ListInsert_sq(L, num, name);
            break;
        case 's':
            if(task[1] == 'h')
                show(L);
            else
            {
                cin>>name;
                printf("%d\n", LocateElem_Sq(L, name));
            }
            break;
        case 'd':
            cin>>name;
            int k = LocateElem_Sq(L, name);
            ListDelete_Sq(L, k, name);
            break;
        }
    }
    return 0;
}

// 用链表实现的线性表
// #include <stdio.h>
// #include <stdlib.h>
// #include <string.h>

// typedef struct LinkList{
//     char name[30];
//     struct LinkList * next;
// }LinkList;

// 在a位置之前插入data为e的节点

// LinkList * ListInsert_Sq(LinkList * L,int a,char *e){
//     LinkList * head,*s;
//     head = L;
//     s = (LinkList *)malloc(sizeof(LinkList));
//     strcpy(s->name,e);
//     int i;
//     for(i=1;i<a;++i)
//     {
//         L = L->next;
//     }
//     s->next = L->next;
//     L->next = s;
//     return head;
// }
```

```
//遍历线型表
// void Show_Sq(LinkList * L)
// {
//     L = L->next;
//     if(L){
//         printf("%s",L->name);
//         L = L->next;
//     }
//     while(L)
//     {
//         printf(" %s",L->name);
//         L = L->next;
//     }
//     printf("\n");
// }


//在线型表中查找data为e的元素，返回其在线型表中的位置（head的下一个节点为1）
// int Search_Sq(LinkList * L,char *e)
// {
//     LinkList * s = L->next;
//     int i=1;
//     while(s&&strcmp(e,s->name))
//     {
//         s = s->next;
//         ++i;
//     }
//     return i;
// }

// 删除线型表中data为e的节点（只能删除第一个）
// LinkList * ListDelete_Sq(LinkList * L,char * e)
// {
//     LinkList * s = L;
//     while(s->next&&strcmp(e,s->next->name))
//     {
//         s = s->next;
//     }
//     LinkList *q = s->next;
//     s->next = s->next->next;
//     free(q);
//     return L;
// }

// int main()
// {
//     LinkList * L;
//     L = (LinkList *)malloc(sizeof(LinkList));
//     char name[30];
//     int a;
//     while(scanf("%s",name)!=EOF){
//         if(strcmp(name,"insert")==0){
//             scanf("%d",&a);
//             scanf("%s",name);
//             L = ListInsert_Sq(L,a,name);
//         }
//         else if(strcmp(name,"show")==0){
//             Show_Sq(L);
//         }
//         else if(strcmp(name,"search")==0){
//             scanf("%s",name);
//             int k = Search_Sq(L,name);
//             printf("%d\n",k);
//         }
```

```
//         else if(strcmp(name,"delete")==0){
//             scanf("%s",name);
//             L=ListDelete_Sq(L,name);
//         }
//     }
//     return 0;
// }
```

单链表的头插法

```
#include<bits/stdc++.h>
using namespace std;
#define N 20
struct Node
{
 int data;
 Node *next;
};

void Delete(Node *head,int x)
{
    int flag=0;
    Node *p=head;
    for(;p->next!=NULL;p=p->next)
    {
        if(p->next->data==x)
        {
            flag=1;
            p->next->data=-1;
        }
    }
    if(!flag)cout<<"No"<<endl;

}

void Create(Node *head)
{
 int x;
 while(cin>>x)
 {
     if(x==0)break;
     Node *t=new Node;
     t->data=x;t->next=head->next;
     head->next=t;
 }

}

void OutPrint(Node *head)
{
    Node *p=head->next;
    for(;p!=NULL;p=p->next){
        if(p->data!=-1)
        cout<<p->data<<" ";
    }
}

int main(void)
{
    Node *head;
    int x;
    head=new Node;
    head->next=NULL;
```

```
    Create(head);

    cin>>x;
    Delete(head,x);

    OutPrint(head);
    return 0;
}
```

单链表的合并

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node *ptrtonode;
struct node
{
    int data;
    ptrtonode next;
};
typedef ptrtonode list;
list init()
{
    int length;
    scanf("%d", &length);
    list head = (list)malloc(sizeof(struct node));
    if(head == NULL)
        return NULL;
    else
    {
        list p = head;
        head->next = NULL;
        int i;
        for(i=1; i<=length; i++)
        {
            list n = (list)malloc(sizeof(struct node));
            int _data;
            scanf("%d", &_data);
            n->data = _data;
            n->next = NULL;
            p->next = n;
            p = n;
        }
        return head;
    }
}
void show(list head)
{
    if(head == NULL)
        return;
    list i = head->next;
    while(i != NULL)
    {
        printf("%d ", i->data);
        i = i->next;
    }
    printf("\n");
    return;
}
//删除h链表中重复的节点
void del(list &h)
{
    list p = h->next;
    while(p != NULL)
```

```
    {
        if(p->next == NULL)
            break;
        if(p->data == p->next->data)
            p->next = p->next->next;
        else
            p = p->next;
    }
}
// 将链表h1和h2按非降序合并，并且形成一个新链表h3
void resort(list &h1, list &h2, list &h3)
{
    list pa, pb, pc;
    pa = h1->next;
    pb = h2->next;
    h3 = (list)malloc(sizeof(node));
    pc = h3;
    while(pa!=NULL && pb!=NULL)
    {
        if(pa->data < pb->data)
        {
            pc->next = pa;
            pc = pa;
            pa = pa->next;
        }
        else if(pa->data > pb->data)
        {
            pc->next = pb;
            pc = pb;
            pb = pb->next;
        }
        else
            pa = pa->next;
    }
    if(pa != NULL)
        pc->next = pa;
    else
        pc->next = pb;
    return;
}
int main()
{
    list head1, head2, head3;
    head1 = init();
    head2 = init();
    //show(head1);
    //show(head2);
    resort(head1, head2, head3);
    del(head3);
    show(head3);
    return 0;
}


// #include<bits/stdc++.h>
// #include<stdlib.h>
// using namespace std;
// typedef long long ll;

// struct node{
//     double data;
//     int index;
//     node *next;
//     node():index(0){}
//     node* operator [] (int n){
```

```
//         node* end=next;
//         while(end&&n--)end=end->next;
//         return end;
//     }
//     bool operator < (const node &t) const {
//         return index>t.index;
//     }
//     node operator * (node& t);
// };
// // 新建长度为length的链表，边建边排序
// void newList(node & head,int length){
//     node *a=new node[length];
//     for(int i=0;i<length;++i)cin>>a[i].data>>a[i].index;
//     sort(a,a+length);
//     node* end=&head;
//     for(int i=0;i<length;++i){
//         node* t=new node;
//         t->data=a[i].data;
//         t->index=a[i].index;
//         end->next=t;
//         end=t;
//     }
//     delete[] a;
// }
// void show(node& head){
//     node* end=head.next;
//     while(end){
//         if(end->index==1) cout<<end->data<<"X"<<(end->next?" + ":"\n");
//         else cout<<end->data<<"X^"<<end->index<<(end->next?" + ":"\n");
//         end=end->next;
//     }
// }

// // 链表的合并，合并后的链表为a
// void combine(node& a, node& b){
//     node* p,*q,*tail,*temp;
//     double s;
//     p=a.next;
//     q=b.next;
//     tail=&a;
//     while(p&&q){
//         if(p->index>q->index){
//             tail->next=p;tail=p;p=p->next;
//         }else if(p->index==q->index){
//             s=p->data+q->data;
//             if(s){
//                 p->data=s;
//                 tail->next=p; tail=p;p=p->next;
//                 temp=q;q=q->next;delete temp;
//             }else{
//                 temp=p;p=p->next;delete temp;
//                 temp=q;q=q->next;delete temp;
//             }
//         }else{
//             tail->next=q; tail=q; q=q->next;
//         }
//     }
//     if(p)tail->next=p;
//     else tail->next=q;
// }

// int main(){
//     node a,b;
//     int n1,n2;cin>>n1;
//     newList(a,n1);
```

```
//     cin>>n2;
//     newList(b,n2);
//     combine(a,b);
//     cin>>n1;
//     cout<<fixed<<setprecision(1)<<a[n1-1]->data<<" "<<a[n1-1]->index;
// }
```

双向循环链表

```
#include <stdio.h>
#include <stdlib.h>
#define ERROR NULL
#define OK 1
typedef struct node *ptrtonode;
struct node
{
    int data;
    ptrtonode pre;
    ptrtonode next;
};
typedef ptrtonode List;
List Listinit()
{
    int n;
    scanf("%d", &n);
    List head = (List)malloc(sizeof(struct node));
    if(head == NULL)
        return ERROR;
    else
    {
        List p = head;
        for(int i=1; i<=n; i++)
        {
            List n = (List)malloc(sizeof(struct node));
            if(n == NULL)
                return ERROR;
            int num;
            scanf("%d", &num);
            n->data = num;
            n->pre = p;
            p->next = n;
            p = n;
            p->next = head;
            head->pre = p;
        }
    }
    return head;
}
List mysort(List &head)
{
    List p = head->next;
    List n = head->next;
    int num = n->data;
    p = p->next;
    while(num > p->data && p != head)
    {
        p = p->next;
    }
    head->next = n->next;
    n->next->pre = head;
    p->pre->next = n;
    n->pre = p->pre;
    p->pre = n;
    n->next = p;
```

```
    return head;
}
void show(List head)
{
    List p = head->next;
    while(p != head)
    {
        printf("%d ", p->data);
        p = p->next;
    }
    printf("\n");
}
int main()
{
    List head = Listinit();
    //show(head);
    head = mysort(head);
    show(head);
    return 0;
}


// #include <iostream>

// using namespace std;
// typedef struct DNode
// {
//     int data;
//     DNode *prior,*next;
// }DNode,*DoubleList;

// DNode* Create(int x)//
// {
//     DNode*head,*p;
//     head=new DNode;
//     p=head;
//     for(int i=0;i<x;i++)
//     {
//         p->next=new DNode;
//         p=p->next;
//         cin>>p->data;
//     }
//     p->next=head->next;//将最后一个节点和第一个节点链接起来
//     p->next->prior=p;
//     p=p->next;
//     do
//     {
//         p->next->prior=p;
//         p=p->next;
//     }while(p->next!=head->next);//最后用一个循环定义prior指针
//     return head;
// }
// // 把第一个节点按照非降序插入到后续链表中
// DNode*Change(DNode *head)
// {
//     DNode*p1,*p2,*p;
//     p1=head->next;//p1始终指向第一个节点，仅移动p2
//     p2=p1->next;
//     while(p2->next!=head->next)
//     {
//         if(p2->data<p1->data&&p2->next->data>=p1->data)
//         {
//             p=p2->next;
//             head->next=p1->next;
//             p1->next->prior=p1->prior;
```

```
//             p1->prior->next=p1->next;
//             p1->prior=p2;
//             p1->next=p;
//             p->prior=p1;
//             p2->next=p1;
//             break;
//         }
//         else
//         {
//             p2=p2->next;
//         }
//     }
//     if(p2->next==head->next&&p2->next->data>p2->prior->data)
//     {
//         head->next=head->next->next;
//     }
//     return head;
// }

// void show(DNode *head)
// {
//     DNode*p;
//     p=head->next;
//     do
//     {

//         cout<<p->data<<" ";
//          p=p->next;
//     }while(p!=head->next);
// }


// int main()
// {
//     DoubleList a;
//     int m;
//     cin>>m;
//     a=Create(m);
//     a=Change(a);
//     show(a);
//     return 0;
// }
```

静态链表

```
//
#include <stdio.h>
#include <string.h>

#define MAXSIZE 11                          // 静态链表的长度
typedef char ElemType[8];        // 元素的类型，规定姓氏不超过7个字符

typedef struct
{
    ElemType data;                          // 节点中的数据
    int cur;                                  // 下一个节点的下标（相当于指针）
} NodeType;                                         // 节点类型

NodeType space[MAXSIZE];        // 用来存储节点的数组，相当于一般链表中的内存，
                                // 只是那个内存是系统分配的，我们看不到,而这个内存是静态的
                                // 相当于模拟了内存的分配

typedef struct
{
```

```
    int elem;                                   // 静态链表存储空间基址（起始元素的下标）
    int length;                                   // 静态链表中的元素数目
    int listSize;                           // 静态链表当前的长度，可容纳元素数目
} SLinkList;                                          // 静态链表类型的定义，和一般的链表类似

int LocateElem_SL(SLinkList& S, ElemType e)
{
    // 在静态单链线性表L中查找第1个值为e的元素。
    // 若找到，则返回它在L中的位序，否则返回0。
    int i;
    i = S.elem; // i指示表中第一个结点
    while (i && strcmp(space[i].data, e))
        i = space[i].cur; // 在表中顺链查找
    return i;
}

void InitSpace_SL()
{
    // 将一维数组space中各分量链成一个备用链表，space[0].cur为头指针，
    // "0"表示空指针
    memset(space, 0 ,sizeof(space));
    for (int i = 0; i < MAXSIZE - 1; ++i)
        space[i].cur = i + 1;
    space[MAXSIZE - 1].cur = 0;
}

int Malloc_SL()
{
    /* 若备用链表非空,则返回分配的结点下标(备用链表的第一个结点),否则返回0 */
    int i = space[0].cur;
    if (i) /* 备用链表非空 */
        space[0].cur = space[i].cur;/* 备用链表的头结点指向原备用链表的第二个结点 */
    return i;/* 返回新开辟结点的坐标 */
}

void Free_SL(int k)
{/* 将下标为k的空闲结点回收到备用链表(成为备用链表的第一个结点) */
    space[k].cur = space[0].cur;/* 回收结点的＂游标＂指向备用链表的第一个结点 */
    space[0].cur = k; /* 备用链表的头结点指向新回收的结点 */
}

void Insert_SL(SLinkList& S, int i, ElemType e)
{
    // 往静态链表S中的第 i 个位置前插入e
    int cur = S.elem;                // 指向静态链表中的第一个节点
    int j=0;
    int newNodeIndex;                // 存储新分配的节点下标
    while(j < i-1)                          // 寻找第 i-1 个节点
    {
        cur = space[cur].cur;
        ++j;
    }
    newNodeIndex = Malloc_SL();        // 分配新的节点
    strcpy(space[newNodeIndex].data,e);        // 在新的节点中存入数据
    space[newNodeIndex].cur = 0;                // 指针为空，这一点很重要
    space[newNodeIndex].cur = space[cur].cur;        // 插入静态链表中
    space[cur].cur = newNodeIndex;
    S.length++;                          // 插入后静态链表长度加1
}

void Delete_SL(SLinkList& S, int i)
{
    // 删除静态链表中的第 i 个节点
    int cur = S.elem;                // 指向静态链表中的第一个节点
    int j=0;
```

```
    int delCur;                                 // 存储待删除节点的下标
    while(j < i-1)                            // 寻找第 i-1 个节点
    {
        cur = space[cur].cur;
        ++j;
    }
    delCur = space[cur].cur;                 // 找到待删除节点的下标
    space[cur].cur = space[delCur].cur;         // 删除节点
    Free_SL(delCur);                          // 释放节点
    S.length--;                           // 删除后静态链表长度减1
}

void CreateList_SL(SLinkList& S)          // 创建静态链表
{
    S.elem = Malloc_SL();                          // 分配头结点的指针
    space[S.elem].cur = 0;
    S.length = 0;
    S.listSize = 9;
}

void Show_space()
{
    // 将静态链表中所有的节点显示出来
    int i;
    for(i=0; i<MAXSIZE; i++)
    {
        printf("%-8s%2d\n", space[i].data, space[i].cur);
    }
}

int main()
{

    SLinkList S;                 // 定义静态链表
    char str[10];                 // 用来获得指令
    int a;                                  // 存储位置
    ElemType e;                           // 存储元素
    InitSpace_SL();                 // 初始化备用链表
    CreateList_SL(S);         // 创建静态链表
    while(scanf("%s", str) != EOF)
    {
        if(strcmp(str, "insert") == 0)                          // 插入元素
        {
            scanf("%d%s", &a, e);
            Insert_SL(S, a, e);
        }
        else if(strcmp(str, "delete") == 0)          // 删除元素
        {
            scanf("%d", &a);
            Delete_SL(S, a);
        }
        else if(strcmp(str, "search") == 0)          // 搜索元素
        {
            scanf("%s", e);
            printf("%2d\n********************\n", LocateElem_SL(S, e));
        }
        else if(strcmp(str, "show") == 0)                  // 显示静态链表状态
        {
            Show_space();
            puts("********************");                                                  // 注意空一行
        }
    }

    return 0;
}
```

多项式相加

```
#include<bits/stdc++.h>
#include<stdlib.h>
using namespace std;
typedef long long ll;

struct node{
    double data;
    int index;
    node* next;
    node():index(0){}
    node* operator [] (int n){
        node* end=next;
        while(end&&n--)end=end->next;
        return end;
    }
    bool operator < (const node &t) const {
        return index>t.index;
    }
    node operator * (node& t);
};
// 创建链表，另开辟一个空间用来排序结点，然后链起来
void newList(node & head,int length){
    node *a=new node[length];
    for(int i=0;i<length;++i)cin>>a[i].data>>a[i].index;
    sort(a,a+length);
    node* end=&head;
    for(int i=0;i<length;++i){
        node* t=new node;
        t->data=a[i].data;
        t->index=a[i].index;
        end->next=t;
        end=t;
    }
    delete[] a;
}
void show(node& head){
    node* end=head.next;
    while(end){
        if(end->index==1) cout<<end->data<<"X"<<(end->next?" + ":"\n");
        else cout<<end->data<<"X^"<<end->index<<(end->next?" + ":"\n");
        end=end->next;
    }
}

///多项式相加：
void combine(node& a, node& b){
    node* p,*q,*tail,*temp;
    double s;
    p=a.next;
    q=b.next;
    tail=&a;
    while(p&&q){
        if(p->index>q->index){
            tail->next=p;tail=p;p=p->next;
        }else if(p->index==q->index){
            s=p->data+q->data;
            if(s){
                p->data=s;
                tail->next=p; tail=p;p=p->next;
                temp=q;q=q->next;delete temp;
            }else{
```

```
                temp=p;p=p->next;delete temp;
                temp=q;q=q->next;delete temp;
            }
        }else{
            tail->next=q; tail=q; q=q->next;
        }
    }
    if(p)tail->next=p;
    else tail->next=q;
}

int main(){
    node a,b;
    int n1,n2;cin>>n1;
    newList(a,n1);
    cin>>n2;
    newList(b,n2);
    combine(a,b);
    cin>>n1;
    cout<<fixed<<setprecision(1)<<a[n1-1]->data<<" "<<a[n1-1]->index;
}
```

表达式求值（带括号版）

```
#include <stdio.h>
#define MAX 10000
#define ERROR -1
char compare[100][100];
typedef struct STACK
{
    int data[MAX];
    int top;
}stack;

void init(stack *s)
{
    s->top = 0;
}

int push(stack *s, int c)
{
    if(s->top >= MAX)
        return ERROR;
    else
        s->data[++s->top] = c;
    return 1;
}

int isempty(stack s)
{
    return s.top == 0;
}

int pop(stack *s)
{
    if(s->top <= 0)
        return ERROR;
    s->top--;
    return 1;
}

int front(stack s)
{
    return s.data[s.top];
```

```
}

int calcu(int a, int b, int op)
{
    if(op == '+')
        return a+b;
    if(op == '-')
        return b-a;
    if(op == '*')
        return a*b;
    if(op == '/')
        return b/a;
}
void BuildPriority()
{
    compare['+']['+'] = '>', compare['+']['-'] = '>', compare['+']['*'] = '<', compare['+']['/'] = '<', compare['+']['('] = '<', compare['+'
    compare['-']['+'] = '>', compare['-']['-'] = '>', compare['-']['*'] = '<', compare['-']['/'] = '<', compare['-']['('] = '<', compare['-'
    compare['*']['+'] = '>', compare['*']['-'] = '>', compare['*']['*'] = '>', compare['*']['/'] = '>', compare['*']['('] = '<', compare['*'
    compare['/']['+'] = '>', compare['/']['-'] = '>', compare['/']['*'] = '>', compare['/']['/'] = '>', compare['/']['('] = '<', compare['/'
    compare['(']['+'] = '<', compare['(']['-'] = '<', compare['(']['*'] = '<', compare['(']['/'] = '<', compare['(']['('] = '<', compare['('
}

char precede(char op1, char op2)
{
    return compare[op1][op2];
}

int main()
{
    //printf("%d, %d, %d, %d, %d, %d", '+', '-', '*', '/', '(', ')');
    BuildPriority();
    stack stack_num;
    stack stack_op;
    init(&stack_num);
    init(&stack_op);
    char opra[2*MAX];
    while(scanf("%s", opra) != EOF)
    {
        while(!isempty(stack_num))
            pop(&stack_num);
        while(!isempty(stack_op))
            pop(&stack_op);
        int i;
        for(i=0; opra[i] != '#'; i++)
        {
            if(opra[i] >= '0' && opra[i] <= '9')
            {
                int a = opra[i] - '0';
                while(opra[i+1] >= '0' && opra[i+1] <= '9')
                {
                    a = a*10 + opra[i+1]-'0';
                    i++;
                }
                push(&stack_num, a);
            }
            else
            {
                if(isempty(stack_op))
                    push(&stack_op, opra[i]);
                else
                {
                    char temp = (char)front(stack_op);
                    switch(precede(temp, opra[i]))
                    {
                    case '<':
```

```
                        push(&stack_op, opra[i]);
                        break;
                    case '=':
                        pop(&stack_op);
                        break;
                    case '>':
                        pop(&stack_op);
                        int a = front(stack_num);
                        pop(&stack_num);
                        int b = front(stack_num);
                        pop(&stack_num);
                        push(&stack_num, calcu(a, b, temp));
                        i--;
                        break;
                    }
                }
            }
        }
        while(!isempty(stack_op))
        {
            int a = front(stack_num);
            pop(&stack_num);
            int b = front(stack_num);
            pop(&stack_num);
            char op = (char)front(stack_op);
            pop(&stack_op);
            push(&stack_num, calcu(a, b, op));
        }
        printf("%d\n", front(stack_num));
        pop(&stack_num);
    }
   return 0;
}
```

队列（银行排队）

```
#include<bits/stdc++.h>
using namespace std;
int m,tot,a,b;
template<class T> class Pque{
protected:
    struct node{
        T data;
        node* next;
        node():next(NULL){}
    };
    node* head;
    int Size;
public:
    Pque():Size(0){
        head=new node;
    }
    ~Pque(){
        head=head->next;
        while(head) {
            node* t=head;
            head=head->next;
            delete t;
        }
    }
    void push(T t){
        node* p=head;
        for(;p->next&&p->next->data<t;p=p->next);
```

```
        node* n=new node;
        n->data=t;n->next=p->next;p->next=n;
        ++Size;
    }
    T pop(){
        node* p=head->next;
        head->next=p->next;
        T t=p->data;
        delete p;--Size;
        return t;
    }
    T front(){return head->next->data;}
    int size(){return Size;}
};

int main(){
    while(cin>>m){
        Pque<int> win;cin>>tot;
        double wt=0;
        for(int i=0;i<tot;++i){
            cin>>a>>b;
            while(win.size()&&win.front()<=a)win.pop();
            if(win.size()<m) win.push(a+b);
            else{
                int k=win.front();
                wt+=k-a;
                win.pop();
                win.push(k+b);
            }
        }
        cout<<fixed<<setprecision(2)<<wt/tot<<endl;
    }
}
// #include <stdio.h>
// #define MAXQSIZE 1000
// #define ERROR -1
// #define OK 1
// using namespace std;
// int _max(int a, int b)
// {
//     return a>b?a:b;
// }
// typedef struct
// {
//     int cometime;
//     int length;
// }QElemType;

// typedef struct
// {
//     QElemType base[MAXQSIZE];
//     int front;
//     int rear;
//     int able;
// }SqQueue;

// int InitQueue(SqQueue &Q)
// {
//     Q.front = Q.rear = 0;
//     Q.able = 0;
//     return OK;
// }

// bool QueueEmpty(SqQueue Q)
// {
```

```
//     return Q.front == Q.rear;
// }

// int EnQueue(SqQueue &Q, QElemType e)
// {
//     if((Q.rear+1) & MAXQSIZE == Q.front)
//         return ERROR;
//     Q.base[Q.rear] = e;
//     Q.rear = (Q.rear+1) % MAXQSIZE;
//     return OK;
// }

// int DeQueue(SqQueue &Q, QElemType &e)
// {
//     if(Q.front == Q.rear)
//         return ERROR;
//     e = Q.base[Q.front];
//     Q.front = (Q.front+1) % MAXQSIZE;
//     return OK;
// }

// int main()
// {
//     SqQueue windows[25];
//     int m, total;
//     while(scanf("%d %d", &m, &total) != EOF)
//     {
//         for(int i=1; i<=m; i++)
//         {
//             InitQueue(windows[i]);
//         }
//         for(int i=1; i<=total; i++)
//         {
//             int ct, len;
//             scanf("%d %d", &ct, &len);
//             QElemType e;
//             e.cometime = ct;
//             e.length = len;
//             int wait = -100000;
//             int idx = 0;
//             for(int i=1; i<=m; i++)
//             {
//                 if(wait < ct-windows[i].able)
//                 {
//                     wait = ct-windows[i].able;
//                     idx = i;
//                 }
//             }
//             EnQueue(windows[idx], e);
//             windows[idx].able = _max(windows[idx].able, e.cometime)+e.length;
//         }
//         /*for(int i=1; i<=m; i++)
//         {
//             while(!QueueEmpty(windows[i]))
//             {
//                 QElemType temp;
//                 DeQueue(windows[i], temp);
//                 printf("%d %d ", temp.cometime, temp.length);
//             }
//             printf("\n");
//         }*/
//         float res = 0;
//         for(int i=1; i<=m; i++)
//         {
//             int total_time = 0;
```

```
//             while(!QueueEmpty(windows[i]))
//             {
//                 QElemType temp;
//                 DeQueue(windows[i], temp);
//                 res += _max(0, total_time-temp.cometime);
//                 total_time = _max(total_time,temp.cometime)+temp.length;
//                 //printf("%d\n" ,total_time);
//             }
//         }
//         printf("%.2f\n", res/total);
//     }
//     return 0;
// }
```

栈（迷宫问题）

```
#include<stdio.h>
#include<stdlib.h>
using namespace std;
#define STACK_INIT_SIZE 100
#define STACKINCREMENT 10
#define ERROR 0
#define OK 1
char maze[10][10];
typedef int Status;
typedef struct{
    int x;
    int y;
}PosType;

typedef struct{
    int ord;//步数
    PosType seat;//坐标位置
    int di;//方向
}SElemType;

typedef struct {
    SElemType *base;
    SElemType *top;
    int stacksize;
}SqStack;

Status InitStack(SqStack *s)
{//初始化栈
    s->base = (SElemType*)malloc(STACK_INIT_SIZE*sizeof(SElemType));
    if(!s->base )
        return ERROR;
    s->top = s->base ;
    s->stacksize = STACK_INIT_SIZE;
    return OK;
}

Status Pass(char maze[][10],PosType *s)
{//判断是否可以通过

    if(maze[s->x][s->y]==' '||maze[s->x][s->y]=='S'||maze[s->x][s->y] == 'E')
        return OK;
    return ERROR;
}

void FootPrint(char maze[][10],PosType *s)
{//留下能够通过的标记
    maze[s->x ][s->y ] = '*';
}
```

```
Status Push(SqStack *s,SElemType *e)
{//入栈
    SElemType *newbase;
    if( (s->top-s->base )>= s->stacksize )
    {
        newbase = (SElemType*)realloc(s->base,(s->stacksize+STACKINCREMENT)*sizeof(SElemType));
        if(!newbase)
            return ERROR;
        s->base = newbase;
        s->stacksize += STACKINCREMENT;
    }
    *(s->top)++= *e;

    return OK;
}

Status Pop(SqStack *s,SElemType *e)
{//出栈
    if(s->base == s->top )
        return ERROR;
    *e = *--(s->top);
    return OK;
}

PosType NextPose(PosType *s,int i)
{//更新位置
    if(i == 1)
        s->y = s->y +1;
    else if(i == 2)
        s->x = s->x +1;
    else if( i == 3)
        s->y = s->y -1;
    else
        s->x = s->x -1;
    return *s;
}

Status EmptyStack(SqStack *s)
{//判断是否为空
    if(s->base == s->top )
        return OK;
    return ERROR;
}

void MarkPrint(char maze[][10],PosType *s)
{
    maze[s->x ][s->y] = '!';
 }
Status MazePath(char maze[][10],PosType start,PosType end)
{
    //算法3.3
    //若迷宫maze中存在从入口start到出口end的通道,则求得一条存放在栈中,并返回OK ， 否则返回ERROR
    SqStack s;
    InitStack(&s);
    PosType curpos =start;//设定当前位置为入口位置
    SElemType e;
    int curstep=1;//探索第一步
    do{
        if(Pass(maze,&curpos))//当前位置可以通过
        {
            FootPrint(maze,&curpos);//留下足迹
            e.di = 1;
            e.ord = curstep;
            e.seat = curpos;
```

```
            Push(&s,&e);//加入路径
            if(curpos.x == end.x &&curpos.y == end.y )
            {
                return OK;//到达终点
            }
            curpos = NextPose(&curpos,e.di );//下一个位置是当前位置的东邻
        curstep ++;//探索下一步
        }
        else
        {//当前位置不能通过

            if(!EmptyStack(&s))
            {
                Pop(&s,&e);
                while(e.di == 4&&!EmptyStack(&s))
                {
                    MarkPrint(maze,&e.seat);//留下不能通过的标记
                    Pop(&s,&e);//退回一步
                   // cout<<"M"<<endl;
                }
                if(e.di < 4)
                {
                    e.di ++;
                    Push(&s,&e);//换下一个方向
                    curpos = NextPose(&e.seat,e.di);//当前位置为新方向的相邻块
                }
            }
        }
    }while(!EmptyStack(&s));
    return 0;
}

int main()
{
    int i,j;
    PosType start,end;
    for( i = 0; i < 10; i ++)
    {
        for( j = 0; j <10; j ++)
        {
            scanf("%c",&maze[i][j]);
            if(maze[i][j] == 'S')
            {
                start.x = i;
                start.y = j;
            }
            if(maze[i][j] == 'E')
            {
                end.x = i;
                end.y = j;
            }
        }
        getchar();
    }

    if(MazePath(maze,start,end))
    {
        for( i = 0; i < 10; i ++)
        {
           for( j = 0; j <10; j ++)
                printf("%c",maze[i][j]);
            printf("\n");
        }

    }
```

```
    return 0;
 }
```

稀疏矩阵的快速转置

```
#include <iostream>

using namespace std;
#define MAXSIZE 2500
typedef struct
{
    int row,col;
    int e;
}Triple;
class tsmatrix
{
    int m,n,len;
public:
    Triple date[MAXSIZE];
    tsmatrix(int x=0,int y=0,int z=0):m(x),n(y),len(z){}
    void Init()
    {
        int x,y;
        cin>>x>>y;
        m=x;n=y;
        for(int i=1;i<=m;i++)
            for(int j=1;j<=n;j++)
        {
            cin>>x;
            if(x!=0)
            {
                len++;
                date[len].row=i;
                date[len].col=j;
                date[len].e=x;
            }
        }
    }
    void Print()
    {
       int counter=1;
        for(int i=1;i<=m;i++)
        {
            for(int j=1;j<=n;j++)
        {
           if(i!=date[counter].row||j!=date[counter].col)
            cout<<0<<' ';
           else
            cout<<date[counter++].e<<' ';
        }
          cout<<endl;
        }
    }
    tsmatrix fasttransposetsmatrix()
    {
        tsmatrix t(n,m,len);
        int num[MAXSIZE],cpot[MAXSIZE];    // num[col]表示矩阵M中的第col列中非零元的个数
                                        // cpot[col]指示M中第col列的第一个非零元在t.data中的确定位置
        if(len)
        {
            int col,current;
            for(int i=1;i<=n;i++)
                num[i]=0;
```

```
            for(int i=1;i<=len;i++)
                num[date[i].col]++;
            cpot[1]=1;
            for(int i=2;i<=len;i++)
                cpot[i]=cpot[i-1]+num[i-1];//cpot[1] = 1; cpot[col] = cpot[col-1] + num[col-1]
            for(int i=1;i<=len;i++)
            {
                col=date[i].col;
                current=cpot[col];
                t.date[current].row=date[i].col;
                t.date[current].col=date[i].row;
                t.date[current].e=date[i].e;
                cpot[col]++;
            }
        }
        return t;
    }
};
int main()
{
    tsmatrix t1;
    t1.Init();
    tsmatrix t2=t1.fasttransposetsmatrix();
    t2.Print();
}
```

树

二叉树的非递归遍历

```
#include<bits/stdc++.h>
using namespace std;
#define N 20

typedef struct tree
{
    char ch;
    struct tree *lchild;
    struct tree *rchild;
}BitTree;

BitTree *CreateTree()
{
    BitTree *bt;
    char str;
    scanf("%c",&str);
    if (str=='#')
        return NULL;
    else
    {
        bt=(BitTree *)malloc(sizeof(BitTree));
        bt->ch=str;
        bt->lchild=CreateTree();
        bt->rchild=CreateTree();
        return bt;
    }
}


void PreOrder(BitTree *bt)
{
    BitTree **s;
```

```
    BitTree *p;
    int top=-1;
    //创建栈；
    s=(BitTree **)malloc((N+1)*sizeof(BitTree *));
    //初始化栈；
    s[++top]=bt;
    //非递归前序遍历；
    while(top!=-1)
    {
        p=s[top--];
        printf("%c ",p->ch);
        if(p->rchild)
            s[++top]=p->rchild;
        if(p->lchild)
            s[++top]=p->lchild;
    }
    free(s);
}
void InOrder(BitTree *bt)
{
    BitTree **s;
    BitTree *p,*q;
    int top=-1;
    //创建栈；
    s=(BitTree **)malloc((N+1)*sizeof(BitTree *));
    //非递归中序遍历；
    if(bt)
    {
        while(bt)   //一直遍历左子树直到该结点的左孩子空为止；
        {
            s[++top]=bt;   //将所有左孩子存入栈中；
            bt=bt->lchild;     //指向下一个左子树；
        }
        while(top!=-1)
        {
            p=s[top--];
            printf("%c ",p->ch);  //输出左下角的结点；
            while(p->rchild)  //遍历移动后结点有没有右结点；
            {
                s[++top]=p->rchild;   //将这个结点的右子树入栈；
                q=p->rchild;          //这个右子树结点赋给q；
                while(q->lchild)      //判断结点q有没有左子树；
                {
                    s[++top]=q->lchild;  //有左子树，将与这个结点相连的所有左子树都入栈；
                    q=q->lchild;
                }
                break;
            }
        }
    }
}

//后序遍历需要判断该节点的左子树和右子树是否都已经被访问过了，确定之后才能输出该结点
void PostOrder(BitTree *bt)
{
    BitTree **s;
    BitTree *p;
    int top=-1;
    //创建栈；
    s=(BitTree **)malloc((N+1)*sizeof(BitTree *));
    //非递归后序遍历；
    do
    {
        while(bt)     //一直遍历左子树直到该左子树的左孩子空为止；
        {
```

```
            s[++top]=bt;     //将所有左孩子存入栈中；
            bt=bt->lchild;   //指向下一个左子树；
        }
        p=NULL;
        while(top!=-1)
        {
            bt=s[top];
            if(bt->rchild==p)  //p:表示为null，或者右子节点被访问过了；
            {
                printf("%c ",bt->ch);   //输出结点数据域；
                top--;           //输出以后，top--；
                p=bt;  //p记录下刚刚访问的节点；
            }
            else
            {
                bt=bt->rchild;   //访问右子树结点；
                break;
            }
        }
    }while(top!=-1);
}

int main()
{
    BitTree *btr=CreateTree();

    PreOrder(btr);
    printf("\n");

    InOrder(btr);
    printf("\n");

    PostOrder(btr);
    printf("\n");
    return 0;
}
```

二叉树的遍历互求&LCA

```
#include<iostream>
#include<cstring>

using namespace std;

class BinaryTreeNode
{
public:
    char elem;
    BinaryTreeNode* LChild;
    BinaryTreeNode* RChild;
    BinaryTreeNode():LChild(NULL),RChild(NULL){}
    bool cover(BinaryTreeNode*,char);
};

class BinaryTree
{
public:
    BinaryTreeNode* mRoot;
    BinaryTreeNode* create(char*,char*,int);
    BinaryTreeNode* common(BinaryTreeNode*,char,char);
    bool cover(BinaryTreeNode*,char);

};
// 已知前序和中序遍历序列创建二叉树
```

```
BinaryTreeNode* BinaryTree::create(char* pre,char*in,int length)
{
    if(length==0)
        return NULL;
    BinaryTreeNode* node=new BinaryTreeNode;
    node->elem=*pre;
    int i=0;
    for(;i<length;i++)//找到in中的根节点
    {
        if(*pre==*(in+i))

            break;
    }
    node->LChild=create(pre+1,in,i);
    node->RChild=create(pre+i+1,in+i+1,length-i-1);

    return node;
}

bool BinaryTree::cover(BinaryTreeNode*root,char a)
{
    if(root==NULL)
        return false;
    if(root->elem==a)
        return true;
    else
        return cover(root->LChild,a)||cover(root->RChild,a);
}
// 注意这里本结点不定义为自身的祖先
// 也可以直接遍历一遍s数据值为a的结点的祖先并标记，遍历b的祖先时最近的标记过的祖先即为LCA
BinaryTreeNode* BinaryTree::common(BinaryTreeNode*root,char a,char b)
{
    if(root==NULL)
        return NULL;
    if(a==root->elem||b==root->elem)
        return NULL;
    bool t1=cover(root->LChild,a);
    bool t2=cover(root->LChild,b);
    if(t1!=t2)
        return root;
    else
    {
        if(t1==true)
            return common(root->LChild,a,b);
        else
            return common(root->RChild,a,b);
    }
}


int main()
{
    BinaryTree t;
    char a[100],b[100];
    cin>>a>>b;
    t.mRoot=t.create(a,b,strlen(a));
    char c,d;
    cin>>c>>d;
    BinaryTreeNode*temp=t.common(t.mRoot,c,d);
    if(temp==NULL)
        cout<<"NULL";
    else
        cout<<temp->elem;
    return 0;
}
```

哈夫曼编码

```
#include<iostream>
#include<cstring>
using namespace std;
class HTNode
{
    public:
    int weight;
    int parent;
    int LChild;
    int RChild;

    HTNode(){weight=0;parent=0;LChild=0;RChild=0;}
};

class HuffmanTree
{
private:
    HTNode * Tree;
    char **HuffmanCode;
    int num;
public:
    HuffmanTree(int n){Tree=new HTNode[2*n-1];HuffmanCode=new char*[n];num=n;}
    void select(int i,int &s1,int &s2)// 选择两个权值最小的
    {

        s1=-2;s2=-1;
        int j=0;
        while(s1<0||s2<0)
                {
                    if(Tree[j].parent==0)
                    {if(s1<0)
                        s1=j;
                    else
                        s2=j;
                     }   j++;
                }
                if(Tree[s1].weight>Tree[s2].weight)
                {
                    int temp=s2;s2=s1;s1=temp;
                }
                for(;j<=i;j++)
                {
                    if(Tree[j].parent==0)
                    {
                        if(Tree[s1].weight>Tree[j].weight)
                        {
                            s2=s1;s1=j;continue;
                        }
                        if(Tree[s2].weight>Tree[j].weight)
                        {
                            s2=j;
                        }
                    }
                }

    }
    void CreateHuffmanTree(int w[])
    {
        for(int i=0;i<num;i++)Tree[i].weight=w[i];
        int m=2*num-1;// 还需要num-1个结点，用来合并num个结点（相当于num-1条边）
```

```
        for(int i=num;i<m;i++)// 每次从森林中选取两权值最小的结点，合并成一棵树
        {
            int s1,s2;
            select(i-1,s1,s2);// 保证每次左子树比右子树的权值小；如出现相同权值的，则先出现的在左子树。
            Tree[i].weight=Tree[s1].weight+Tree[s2].weight;
            Tree[s1].parent=i;Tree[s2].parent=i;
            Tree[i].LChild=s1;Tree[i].RChild=s2;
        }
    }
    // 也可以从第(2*num-1)个结点开始往前遍历，左子树+“0”， 右子树+“1”
    void CreateHuffmanCode()
    {
        char *cd=new char[num];
        cd[num-1]='\0';
        for(int i=0;i<num;i++)
        {
            int start=num-1;
            int c=i;int p=Tree[i].parent;
            while(p!=0)
            {
                start--;
                if(Tree[p].LChild==c)
                    cd[start]='0';
                else cd[start]='1';
                c=p;p=Tree[p].parent;
            }

            HuffmanCode[i]=new char [num-start];
            strcpy(HuffmanCode[i],&cd[start]);

        }
        delete []cd;
    }
    void Show()
    {
        for(int i=0;i<num;i++)
        {
            cout<<HuffmanCode[i]<<endl;
        }
    }
    void test()
    {
        for(int i=0;i<2*num-1;i++)
            cout<<Tree[i].weight<<' '<<Tree[i].parent<<' '<<Tree[i].LChild<<' '<<Tree[i].RChild<<endl;
    }
};
int main()
{
    int n;
    cin>>n;
    int *arr=new int [n];
    for(int i=0;i<n;i++)cin>>arr[i];

    HuffmanTree t(n);
    t.CreateHuffmanTree(arr);
    t.CreateHuffmanCode();
    t.Show();
    return 0;
}
```

二叉树的左右子树交换

```
#include <iostream>
```

```
#include <stdlib.h>
#include <stdio.h>
using namespace std;

typedef struct Node
{
    char data;
    struct Node* LChild;
    struct Node* RChild;
} BiTNode,* BiTree;

void CreateBiTree(BiTree *bt)
{
    char ch;
    ch=getchar();
    if(ch=='#')
        *bt=NULL;
    else
    {
        *bt=(BiTree)malloc(sizeof(BiTNode));
        (*bt)->data=ch;
        CreateBiTree(&((*bt)->LChild));
        CreateBiTree(&((*bt)->RChild));
    }
}

int LeafCount;

void Leaf(BiTree root)
{
    if(root!=NULL)
    {
        Leaf(root->LChild);
        Leaf(root->RChild);
        if(root->LChild==NULL&&root->RChild==NULL)
            LeafCount++;
    }

}

void Change(BiTree root)
{
    if(root!=NULL)
    {
        Change(root->LChild);
        Change(root->RChild);
        BiTNode *p;

        p=root->LChild;
        root->LChild=root->RChild;
        root->RChild=p;


    }
}

void PostOrder(BiTree root)
{
    if(root!=NULL)
    {
        PostOrder(root->LChild);
        PostOrder(root->RChild);
        cout<<root->data;
    }
    else
```

```
        cout<<"#";
}
int main()
{
    BiTree root;
    CreateBiTree(&root);
    LeafCount=0;
    Leaf(root);
    cout<<LeafCount<<endl;
    Change(root);
    PostOrder(root);
    return 0;
}
```

邻接矩阵存图&&DFS

```
#include <iostream>
#define MAX 1000
#define MAX_NUM 100
using namespace std;
typedef struct Graph
{
    char vertex[MAX_NUM];
    int arcs[MAX_NUM][MAX_NUM];// 邻接矩阵存边
    int vertexs, brim; //结点数，边数
}Graph;
int visit[MAX];
void g_create(Graph * graph)
{
    int i, j;
    cin>>graph->vertexs;
    for ( i = 0; i < graph->vertexs; i++ )
    for ( j = 0; j < graph->vertexs; j++ )
    {
        graph->arcs[i][j]=0;
    }

    for ( i = 0; i < graph->vertexs; i++ )
    for ( j = 0; j < graph->vertexs; j++ )
    {
        cin>>graph->arcs[i][j];
    }
}

void DFS(Graph graph,int v)
{
    int i;
    visit[v]=1;
    for(i=0;i<graph.vertexs;i++)
    {
        if(graph.arcs[v][i]==1&&!visit[i])
            DFS(graph,i);
    }
}
int main()
{
    Graph g;
    int i,sum=0;
    g_create(&g);
    visit[MAX-1]=0;
    for(i=0;i<g.vertexs;i++)
    {
        if(visit[i]!=1)
        {
```

```
            DFS(g,i);
            sum++;
        }
    }
    cout<<sum<<endl;
    return 0;
}
```

图

最小生成树（prim、kruskual）

```
#include<iostream>
#include <stdio.h>
#include <stdlib.h>
using namespace std;
#define MAX_VERTEX_NUM 20
#define MAXEDGE 20
#define MAXVEX  20
#define INFINITY 65536
struct Fuzhu
{
    int adjvex;
    int lowcost;
};
typedef struct
{
    int arc[MAXVEX][MAXVEX];
    int numVertexes, numEdges;// 结点树，边数
}MGraph;

typedef struct
{
    int begin;
    int end;
    int weight;
}Edge;   //由begin指向end的权值为weight的边
int arc[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
int n,sum=0,count;
//选出最小的lowcost，即为大根堆的作用
int Minium(Fuzhu* p)
{
    int Min=INFINITY;
    int k=0;
    for(int i=1;i<n;i++)
    {
        if(Min>(p+i)->lowcost&&(p+i)->lowcost!=0)//bug
        {
            Min=(p+i)->lowcost;
            k=i;
        }
    }
    sum+=Min;
    return k;
}
void Prim(int arc[][MAX_VERTEX_NUM],int u)
{
    int i,e,v=0;
    Fuzhu closedge[MAX_VERTEX_NUM];
    closedge[u].adjvex=0;
    closedge[u].lowcost=0;
    for(i=0;i<n;i++)
```

```
        if(i!=u)
    {
        closedge[i].adjvex=u;
        closedge[i].lowcost=arc[u][i];
     //   cout<<"+"<<closedge[e].adjvex<<" "<<closedge[i].lowcost<<endl;
    }
    for(e=1;e<n;e++)
    {
        v=Minium(closedge);
        u=closedge[v].adjvex;
      //  cout<<u<<"__"<<v<<endl;
        closedge[v].lowcost=0;
        for(i=0;i<n;i++)
        if(arc[v][i]<closedge[i].lowcost)
        {
            closedge[i].lowcost=arc[v][i];
            closedge[i].adjvex=v;
        }
    }
}




void CreateMGraph(MGraph *G,int arc[][MAX_VERTEX_NUM]) {
    int i, j;
    G->numVertexes=n;
      G->numEdges=count;

    for (i = 0; i < G->numVertexes; i++)
    {
        for ( j = 0; j < G->numVertexes; j++)
        {
            G->arc[i][j]=INFINITY;
        }
    }
    for(i=0;i<G->numVertexes;i++)
         for(j=0;j<G->numVertexes;j++)
         {
             G->arc[i][j]=arc[i][j];
         }
}


int cmp(const void* a, const void* b)
{
    return (*(Edge*)a).weight - (*(Edge*)b).weight;
}


int Find(int *parent, int f) {
    while ( parent[f] > 0) {
        f = parent[f];
    }
    return f;
}


void MiniSpanTree_Kruskal(MGraph G) {
    int i, j, n=0, m=0;
    int sum2=0;
    int k=0;
    int parent[MAXVEX];

    Edge edges[MAXEDGE];
```

```
    for ( i = 0; i < G.numVertexes-1; i++) {
        for (j = i+1; j < G.numVertexes; j++) {
            if (G.arc[i][j]<INFINITY)
            {
                edges[k].begin = i; //□□Ž□C□Ľ□□Ï□□
                edges[k].end = j;   //□□Ž0□Ľ□□Ïβ
                edges[k].weight = G.arc[i][j];
                k++;
            }
        }
    }
    //排序贪心
    qsort(edges, G.numEdges, sizeof(Edge), cmp);

    for (i = 0; i < G.numVertexes; i++)
        parent[i] = 0;
    //并查集维护
    for (i = 0; i < G.numEdges; i++)
    {
        n = Find(parent, edges[i].begin);
        m = Find(parent, edges[i].end);


        if (n!=m)
        {
            parent[n] = m;
            sum2+=edges[i].weight;
        }

    }
 cout<<sum2;
}
int main()
{
    int i,j,row,col,weight;
    MGraph G;
    cin>>n>>count;
    for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        arc[i][j]=INFINITY;
    for(i=0;i<count;i++)
    {
        cin>>row>>col>>weight;
        arc[row-1][col-1]=weight;
        arc[col-1][row-1]=weight;
    }
    Prim(arc,0);
    cout<<sum<<endl;
    CreateMGraph(&G,arc);
    MiniSpanTree_Kruskal(G);
    return 0;
}
```

dijkstra求单源最短路~

```
#include <iostream>

using namespace std;
#define INFINITY 327698
int dist[100]; //□浮μ□瀰□□□□□□□□
struct adjmatrix
{
```

```
    int ars[1000][1000];
    int vexnum;
    int start;
};
void create(adjmatrix &s,int n,int m)
{
    int c;
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            cin>>c;
            if(c)
           s.ars[i][j]=c;
           else
           s.ars[i][j]=INFINITY;
        }

    }
    s.vexnum=n;
    s.start=m;
}
void dikjstra(adjmatrix s)
{
    int i,v,w,Min;
    bool Final[INFINITY];
    for(v=0;v<s.vexnum;v++)
    {
        Final[v]=false;
        dist[v]=s.ars[s.start][v];


    }
    dist[s.start]=0;
    Final[s.start]=true;
    for(i=1;i<s.vexnum;i++)
    {
        Min=INFINITY;
        for(w=0;w<s.vexnum;w++)
        {
            if(!Final[w]&&dist[w]<Min)
            {
                v=w;
                Min=dist[w];
            }
        }
        Final[v]=true;
        for(w=0;w<s.vexnum;w++)
        {
            if(!Final[w]&&(Min+s.ars[v][w]<dist[w]))
            {
                dist[w]=Min+s.ars[v][w];
            }
        }
    }
}
int main()
{
    adjmatrix G;
    int n,m;
     cin>>n>>m;
     create(G,n,m);
    for(int i=0;i<n;i++)
        dist[i]=INFINITY;
    dikjstra(G);
```

```
    for(int i=0;i<n;i++)
    {
        if(i!=G.start&&dist[i]!=INFINITY)
            cout<<dist[i]<<" ";
        else if(i!=G.start&&dist[i]==INFINITY)
            cout<<-1<<" ";
        }
    return 0;
}
```

查找&排序

二叉排序树

```
#include<bits/stdc++.h>
using namespace std;
template<class ElemType> class Tree;
template<class ElemType>
class Tree_Node
{
public:
    ElemType elem;
    Tree_Node<ElemType> *left, *right;
    friend class Tree<ElemType>;
};

template<class ElemType>
class Tree
{
public:
    Tree_Node <ElemType> *root;

    void createTree()
    {
        ElemType elem;
        root = NULL;
        while ((cin >> elem) && (elem != 0))
            {insertTreeNode(root, elem);}
    }
    void insertTreeNode(Tree_Node <ElemType> *&root, ElemType elem)
    {
        if (root == NULL)
        {
            root = new Tree_Node<ElemType>();
            (root)->elem = elem;
            (root)->right = (root)->left = NULL;
        }
        else if (elem < (root)->elem)
        {
            insertTreeNode(root->left, elem);
        }
        else
            insertTreeNode(root->right, elem);
    }

    void inOrder(Tree_Node <ElemType> *p)
    {
        if (p != NULL)
        {
            inOrder(p->left);
            cout << p->elem<<" ";
            inOrder(p->right);
```

```
        }
    }

    void deleteNode(Tree_Node<ElemType> *&root,ElemType x)
    {
        Tree_Node<ElemType> *node = root;
        //p是node的前驱
        Tree_Node<ElemType> *p=NULL;
        //找到x的结点
        while(node!=NULL&&node->elem!=x)
        {
            p = node;
            if(node ->elem < x)
                node = node -> right;
            else
                node = node -> left;
        }//查找写到里面来了
        if(node==NULL)
            return;
        if(node -> left == NULL)// 只需重接右子树
        {
            if(p == NULL)
                root = root -> right;    //p == NULL && node != NULL
                                         //说明node->elem == x 即要删除根节点
            else if(p -> left == node)
                p -> left = node -> right;
            else p -> right = node -> right;
        }
        else if (node -> right ==NULL)// 只需重接左子树
        {
            if (p == NULL)
                root = root -> left;
            else if (p -> left ==node)
                p -> left = node -> left;
            else p -> right = node -> left;
        }
        else
        {
            Tree_Node<ElemType> *q = node->left;
            p = node;
            while(q->right!=NULL)// 左转后向右走到底，p指向q的根结点，找到最大的不大于node结点的结点
                p=q,q = q -> right;
            node -> elem = q -> elem;
            if(p==node)// node的左子树没有右子树，即左子树就是要换上来的结点
                node -> left = q -> left;
            else
                p -> left = q -> left;
        }
    }

    int findLevel(Tree_Node<ElemType> *root,ElemType x)
    {
        if(root==NULL)
            return 0;
        else if(root->elem == x)
            return 1;
        else if(root->elem < x)
            return findLevel(root -> right,x)+1;
        else
            return findLevel(root -> left,x)+1;
    }
};
using  namespace std;
int main()
{
```

```
    Tree<int> tree;
    tree.createTree();
    int x;
    cin>>x;
    tree.deleteNode(tree.root,x);
    tree.inOrder(tree.root);
    cout<<endl;
    cin>>x;
    cout<<tree.findLevel(tree.root,x)<<endl;
    return 0;
}
```

希尔排序

```
#include <iostream>
using namespace std;
const static int maxn = 1e3+10;

int shellf[4];

void shellinsert(int a[], int g, int n)
{
    for(int i=g+1; i<=n; i++)
    {
        a[0] = a[i];
        int j = i-g;
        while(j > 0 && a[j] > a[0])
        {
            a[j+g] = a[j];
            j -= g;
        }
        a[j+g] = a[0];
    }
}
void shellsort(int a[], int n)
{
    for(int i=1; i<=3; i++)
    {
        shellinsert(a, shellf[i], n);
        for(int i=1; i<=n; i++)
        {
            cout<<a[i]<<" ";
        }
        cout<<endl;
    }
}

int main()
{
    int num = 0;
    int temp;
    int a[maxn];
    while(cin>>temp, temp)
    {
        a[++num] = temp;
    }
    for(int i=1; i<=3; i++)
    {
        cin>>shellf[i];// shellf[i] = shellf[i-1]*3 + 1
    }
    shellsort(a, num);
    cout<<endl;
    return 0;
}
```

堆排序

```
#include <iostream>
using namespace std;
const static int maxn = 1e4+10;
int a[maxn];
int tok = 0;
//这个调整函数就该这样写成递归的形式，书上那种非人类的写法我是真的看不懂
void pushdown(int i, int m)//在该节点和左右字节点中选一个最大的，和该节点换位置，然后递归调整
                           //保证堆顶的元素最大
{
    int l = i*2;
    int r = i*2+1;
    int largest = i;
    if(l <= m && a[l] > a[largest])
        largest = l;
    if(r <= m && a[r] > a[largest])
        largest = r;
    if(largest != i)
    {
        swap(a[i], a[largest]);
        pushdown(largest, m);
    }
}
int main()
{
    int num;
    while(cin>>num, num)
    {
        a[++tok] = num;
    }
    for(int i=tok/2; i>=1; i--)
    {
        pushdown(i, tok);
    }
    for(int i=1; i<=tok; i++)
    {
        cout<<a[i]<<" ";
    }
    cout<<endl;
    for(int i=tok; i>1; i--)
    {
        swap(a[i], a[1]);//每次将最大的元素丢到数组的最后面，然后调整剩下的堆，得到次大的堆顶元素，循环至调整结束。
        pushdown(1, i-1);
        for(int i=1; i<=tok; i++)
        {
            cout<<a[i]<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

快速排序/双重冒泡/冒泡

```
#include <iostream>
using namespace std;
const static int maxn = 1e4+10;
int a[maxn];
int b[maxn];
int c[maxn];
```

```
int tok = 0;
int Partition(int low, int high)//分治
{
    int temp = a[low];
    while(low < high)
    {
        while(low < high && a[high] > temp)
            high--;
        a[low] = a[high];
        while(low < high && a[low] < temp)
            low++;
        a[high] = a[low];
    }
    a[low] = temp;
    return low;
}

void Qsort(int low, int high)
{
    if(low < high)
    {
        int mid = Partition(low, high);
        Qsort(low, mid-1);
        Qsort(mid+1, high);
    }
}
void Bubblesort()
{
    for(int i=1; i<tok; i++)
    {
        bool flag = true;
        for(int j=tok; j>=i+1; j--)
        {
            if(b[j] < b[j-1])
            {
                swap(b[j], b[j-1]);
                flag = false;
            }
        }
        if(flag)
            break;
    }
}
void DeBubblesort()
{
    int low = 1, high = tok;
    int temp = 1;
    while(low < high)
    {
        for(int i=low; i<high; i++)
        {
            if(c[i] > c[i+1])
            {
                swap(c[i], c[i+1]);
                temp = i;
            }
        }
        high = temp;
        for(int i=high-1; i>=low; i--)
        {
            if(c[i] > c[i+1])
            {
                swap(c[i], c[i+1]);
                temp = i;
            }
```

```
        }
        low = temp;
    }
}
int main()
{
    int num;
    while(cin>>num, num)
    {
        a[++tok] = num;
        b[tok] = num;
        c[tok] = num;
    }
    Qsort(1, tok);
    for(int i=1; i<=tok; i++)
    {
        cout<<a[i]<<" ";
    }
    cout<<endl;
    Bubblesort();
    for(int i=1; i<=tok; i++)
    {
        cout<<b[i]<<" ";
    }
    cout<<endl;
    DeBubblesort();
    for(int i=1; i<=tok; i++)
    {
        cout<<c[i]<<" ";
    }
    cout<<endl;
    return 0;
}
```