第一章 线性表

1. 01 线性表顺序存储_List

```
#include "stdio.h"

#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 20 /* 存储空间初始分配量 */

typedef int Status;              /* Status 是函数的类型,其值是函数结果状态代码，如OK 等 */
typedef int ElemType;            /* ElemType 类型根据实际情况而定，这里假设为 int */



Status visit(ElemType c)
{
    printf("%d ",c);
    return OK;
}

typedef struct
{
 ElemType data[MAXSIZE];          /* 数组，存储数据元素 */
 int length;                                  /* 线性表当前长度 */
}SqList;

/* 初始化顺序线性表 */
Status InitList(SqList *L)
{
    L->length=0;
    return OK;
```

```
}

/* 初始条件：顺序线性表 L 已存在。操作结果：若 L 为空表，则返回 TRUE，否则返回 FALSE */
Status ListEmpty(SqList L)
{
 if(L.length==0)
       return TRUE;
 else
       return FALSE;
}

/* 初始条件：顺序线性表 L 已存在。操作结果：将 L 重置为空表 */
Status ClearList(SqList *L)
{
      L->length=0;
      return OK;
}

/* 初始条件：顺序线性表 L 已存在。操作结果：返回 L 中数据元素个数 */
int ListLength(SqList L)
{
 return L.length;
}

/* 初始条件：顺序线性表 L 已存在，1≤i≤ListLength(L) */
/* 操作结果：用 e 返回 L 中第 i 个数据元素的值,注意 i 是指位置，第 1 个位置的数组是从 0 开始 */
Status GetElem(SqList L,int i,ElemType *e)
{
      if(L.length==0 || i<1 || i>L.length)
                  return ERROR;
      *e=L.data[i-1];

      return OK;
}

/* 初始条件：顺序线性表 L 已存在 */
/* 操作结果：返回 L 中第 1 个与 e 满足关系的数据元素的位序。 */
/* 若这样的数据元素不存在，则返回值为 0 */
int LocateElem(SqList L,ElemType e)
{
      int i;
      if (L.length==0)
```

```
            return 0;
        for(i=0;i<L.length;i++)
        {
            if (L.data[i]==e)
                break;
        }
        if(i>=L.length)
            return 0;

        return i+1;
    }


    /* 初始条件：顺序线性表 L 已存在,1≤i≤ListLength(L)， */
    /* 操作结果：在 L 中第 i 个位置之前插入新的数据元素 e，L 的长度加 1 */
    Status ListInsert(SqList *L,int i,ElemType e)
    {
     int k;
     if (L->length==MAXSIZE)   /* 顺序线性表已经满 */
          return ERROR;
     if  (i<1  ||  i>L->length+1)/* 当 i 比第一位置小或者比最后一位置后一位置还要大时
*/
          return ERROR;

     if (i<=L->length)            /* 若插入数据位置不在表尾 */
     {
          for(k=L->length-1;k>=i-1;k--)   /* 将要插入位置之后的数据元素向后移动一位
*/
               L->data[k+1]=L->data[k];
     }
     L->data[i-1]=e;               /* 将新元素插入 */
     L->length++;

     return OK;
    }

    /* 初始条件：顺序线性表 L 已存在，1≤i≤ListLength(L) */
    /* 操作结果：删除 L 的第 i 个数据元素，并用 e 返回其值，L 的长度减 1 */
    Status ListDelete(SqList *L,int i,ElemType *e)
    {
        int k;
        if (L->length==0)                  /* 线性表为空 */
         return ERROR;
        if (i<1 || i>L->length)            /* 删除位置不正确 */
```

```
        return ERROR;
    *e=L->data[i-1];
    if (i<L->length)                    /* 如果删除不是最后位置 */
    {
        for(k=i;k<L->length;k++)/* 将删除位置后继元素前移 */
         L->data[k-1]=L->data[k];
    }
    L->length--;
    return OK;
}

/* 初始条件：顺序线性表 L 已存在 */
/* 操作结果：依次对 L 的每个数据元素输出 */
Status ListTraverse(SqList L)
{
 int i;
    for(i=0;i<L.length;i++)
            visit(L.data[i]);
    printf("\n");
    return OK;
}

void unionL(SqList *La,SqList Lb)
{
 int La_len,Lb_len,i;
 ElemType e;
 La_len=ListLength(*La);
 Lb_len=ListLength(Lb);
 for (i=1;i<=Lb_len;i++)
 {
     GetElem(Lb,i,&e);
     if (!LocateElem(*La,e))
         ListInsert(La,++La_len,e);
 }
}

int main()
{

    SqList L;
    ElemType e;
    Status i;
    int j,k;
    i=InitList(&L);
```

```
printf("初始化 L 后：L.length=%d\n",L.length);
for(j=1;j<=5;j++)
	i=ListInsert(&L,1,j);
printf("在 L 的表头依次插入 1～5 后：L.data=");
ListTraverse(L);

printf("L.length=%d \n",L.length);
i=ListEmpty(L);
printf("L 是否空：i=%d(1:是  0:否)\n",i);

i=ClearList(&L);
printf("清空 L 后：L.length=%d\n",L.length);
i=ListEmpty(L);
printf("L 是否空：i=%d(1:是  0:否)\n",i);

for(j=1;j<=10;j++)
	ListInsert(&L,j,j);
printf("在 L 的表尾依次插入 1～10 后：L.data=");
ListTraverse(L);

printf("L.length=%d \n",L.length);

ListInsert(&L,1,0);
printf("在 L 的表头插入 0 后：L.data=");
ListTraverse(L);
printf("L.length=%d \n",L.length);

GetElem(L,5,&e);
printf("第 5 个元素的值为：%d\n",e);
for(j=3;j<=4;j++)
{
	k=LocateElem(L,j);
	if(k)
		printf("第%d 个元素的值为%d\n",k,j);
	else
		printf("没有值为%d 的元素\n",j);
}


k=ListLength(L); /* k 为表长  */
for(j=k+1;j>=k;j--)
{
	i=ListDelete(&L,j,&e); /*  删除第 j 个数据  */
	if(i==ERROR)
```

```
                printf("删除第%d 个数据失败\n",j);
            else
                printf("删除第%d 个的元素值为：%d\n",j,e);
        }
        printf("依次输出 L 的元素：");
        ListTraverse(L);

        j=5;
        ListDelete(&L,j,&e); /* 删除第 5 个数据 */
        printf("删除第%d 个的元素值为：%d\n",j,e);

        printf("依次输出 L 的元素：");
        ListTraverse(L);

    //构造一个有 10 个数的 Lb
    SqList Lb;
        i=InitList(&Lb);
        for(j=6;j<=15;j++)
            i=ListInsert(&Lb,1,j);

    unionL(&L,Lb);

    printf("依次输出合并了 Lb 的 L 的元素：");
        ListTraverse(L);

        return 0;
    }
```

02 线性表链式存储_LinkList

```
#include "stdio.h"
#include "string.h"
#include "ctype.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 20 /* 存储空间初始分配量 */
```

```
typedef int Status;/* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */
typedef int ElemType;/* ElemType 类型根据实际情况而定，这里假设为 int */


Status visit(ElemType c)
{
	printf("%d ",c);
	return OK;
}

typedef struct Node
{
	ElemType data;
	struct Node *next;
}Node;
typedef struct Node *LinkList; /* 定义 LinkList */

/* 初始化顺序线性表 */
Status InitList(LinkList *L)
{
	*L=(LinkList)malloc(sizeof(Node)); /* 产生头结点,并使 L 指向此头结点 */
	if(!(*L)) /* 存储分配失败 */
		return ERROR;
	(*L)->next=NULL; /* 指针域为空 */

	return OK;
}

/* 初始条件：顺序线性表 L 已存在。操作结果：若 L 为空表，则返回 TRUE，否则返回 FALSE
*/
Status ListEmpty(LinkList L)
{
	if(L->next)
		return FALSE;
	else
		return TRUE;
}

/* 初始条件：顺序线性表 L 已存在。操作结果：将 L 重置为空表 */
Status ClearList(LinkList *L)
{
	LinkList p,q;
	p=(*L)->next;			/*  p 指向第一个结点 */
```

```
    while(p)                    /*  没到表尾 */
    {
        q=p->next;
        free(p);
        p=q;
    }
    (*L)->next=NULL;            /* 头结点指针域为空 */
    return OK;
}

/* 初始条件：顺序线性表 L 已存在。操作结果：返回 L 中数据元素个数 */
int ListLength(LinkList L)
{
    int i=0;
    LinkList p=L->next; /* p 指向第一个结点 */
    while(p)
    {
        i++;
        p=p->next;
    }
    return i;
}

/* 初始条件：顺序线性表 L 已存在，1≤i≤ListLength(L) */
/* 操作结果：用 e 返回 L 中第 i 个数据元素的值 */
Status GetElem(LinkList L,int i,ElemType *e)
{
    int j;
    LinkList p;             /* 声明一结点 p */
    p = L->next;            /* 让 p 指向链表 L 的第一个结点 */
    j = 1;      /*  j 为计数器 */
    while (p && j<i)    /* p 不为空或者计数器 j 还没有等于 i 时，循环继续 */
    {
        p = p->next;    /* 让 p 指向下一个结点 */
        ++j;
    }
    if ( !p || j>i )
        return ERROR;   /*  第 i 个元素不存在 */
    *e = p->data;     /*  取第 i 个元素的数据 */
    return OK;
}

/* 初始条件：顺序线性表 L 已存在 */
/* 操作结果：返回 L 中第 1 个与 e 满足关系的数据元素的位序。 */
```

```
/* 若这样的数据元素不存在，则返回值为 0 */
int LocateElem(LinkList L,ElemType e)
{
    int i=0;
    LinkList p=L->next;
    while(p)
    {
        i++;
        if(p->data==e) /* 找到这样的数据元素 */
            return i;
        p=p->next;
    }

    return 0;
}


/* 初始条件：顺序线性表 L 已存在,1≤i≤ListLength(L)， */
/* 操作结果：在 L 中第 i 个位置之前插入新的数据元素 e，L 的长度加 1 */
Status ListInsert(LinkList *L,int i,ElemType e)
{
    int j;
    LinkList p,s;
    p = *L;
    j = 1;
    while (p && j < i)        /* 寻找第 i 个结点 */
    {
        p = p->next;
        ++j;
    }
    if (!p || j > i)
        return ERROR;     /* 第 i 个元素不存在 */
    s = (LinkList)malloc(sizeof(Node));  /*  生成新结点(C 语言标准函数) */
    s->data = e;
    s->next = p->next;          /* 将 p 的后继结点赋值给 s 的后继  */
    p->next = s;                /* 将 s 赋值给 p 的后继 */
    return OK;
}

/* 初始条件：顺序线性表 L 已存在，1≤i≤ListLength(L) */
/* 操作结果：删除 L 的第 i 个数据元素，并用 e 返回其值，L 的长度减 1 */
Status ListDelete(LinkList *L,int i,ElemType *e)
{
    int j;
```

```
    LinkList p,q;
    p = *L;
    j = 1;
    while (p->next && j < i)  /* 遍历寻找第 i 个元素 */
    {
        p = p->next;
        ++j;
    }
    if (!(p->next) || j > i)
        return ERROR;                /* 第 i 个元素不存在 */
    q = p->next;
    p->next = q->next;              /* 将 q 的后继赋值给 p 的后继 */
    *e = q->data;                   /* 将 q 结点中的数据给 e */
    free(q);                        /* 让系统回收此结点，释放内存 */
    return OK;
}

/* 初始条件：顺序线性表 L 已存在 */
/* 操作结果：依次对 L 的每个数据元素输出 */
Status ListTraverse(LinkList L)
{
    LinkList p=L->next;
    while(p)
    {
        visit(p->data);
        p=p->next;
    }
    printf("\n");
    return OK;
}

/*  随机产生 n 个元素的值，建立带表头结点的单链线性表 L（头插法） */
void CreateListHead(LinkList *L, int n)
{
    LinkList p;
    int i;
    srand(time(0));                         /* 初始化随机数种子 */
    *L = (LinkList)malloc(sizeof(Node));
    (*L)->next = NULL;                      /*  先建立一个带头结点的单链表 */
    for (i=0; i<n; i++)
    {
        p = (LinkList)malloc(sizeof(Node)); /*  生成新结点 */
        p->data = rand()%100+1;             /*  随机生成 100 以内的数字 */
        p->next = (*L)->next;
```

```
            (*L)->next = p;                          /*   插入到表头 */
        }
}

/*   随机产生 n 个元素的值，建立带表头结点的单链线性表 L（尾插法） */
void CreateListTail(LinkList *L, int n)
{
        LinkList p,r;
        int i;
        srand(time(0));                              /* 初始化随机数种子 */
        *L = (LinkList)malloc(sizeof(Node)); /* L 为整个线性表 */
        r=*L;                                        /* r 为指向尾部的结点 */
        for (i=0; i<n; i++)
        {
            p = (Node *)malloc(sizeof(Node)); /*   生成新结点 */
            p->data = rand()%100+1;                  /*   随机生成 100 以内的数字 */
            r->next=p;                               /* 将表尾终端结点的指针指向新结点 */
            r = p;                                   /* 将当前的新结点定义为表尾终端结点 */
        }
        r->next = NULL;                              /* 表示当前链表结束 */
}

int main()
{
        LinkList L;
        ElemType e;
        Status i;
        int j,k;
        i=InitList(&L);
        printf("初始化 L 后：ListLength(L)=%d\n",ListLength(L));
        for(j=1;j<=5;j++)
                i=ListInsert(&L,1,j);
        printf("在 L 的表头依次插入 1～5 后：L.data=");
        ListTraverse(L);

        printf("ListLength(L)=%d \n",ListLength(L));
        i=ListEmpty(L);
        printf("L 是否空：i=%d(1:是  0:否)\n",i);

        i=ClearList(&L);
        printf("清空 L 后：ListLength(L)=%d\n",ListLength(L));
        i=ListEmpty(L);
        printf("L 是否空：i=%d(1:是  0:否)\n",i);
```

```
for(j=1;j<=10;j++)
        ListInsert(&L,j,j);
printf("在 L 的表尾依次插入 1～10 后：L.data=");
ListTraverse(L);

printf("ListLength(L)=%d \n",ListLength(L));

ListInsert(&L,1,0);
printf("在 L 的表头插入 0 后：L.data=");
ListTraverse(L);
printf("ListLength(L)=%d \n",ListLength(L));

GetElem(L,5,&e);
printf("第 5 个元素的值为：%d\n",e);
for(j=3;j<=4;j++)
{
        k=LocateElem(L,j);
        if(k)
                printf("第%d 个元素的值为%d\n",k,j);
        else
                printf("没有值为%d 的元素\n",j);
}


k=ListLength(L); /* k 为表长 */
for(j=k+1;j>=k;j--)
{
        i=ListDelete(&L,j,&e); /* 删除第 j 个数据 */
        if(i==ERROR)
                printf("删除第%d 个数据失败\n",j);
        else
                printf("删除第%d 个的元素值为：%d\n",j,e);
}
printf("依次输出 L 的元素：");
ListTraverse(L);

j=5;
ListDelete(&L,j,&e); /* 删除第 5 个数据 */
printf("删除第%d 个的元素值为：%d\n",j,e);

printf("依次输出 L 的元素：");
ListTraverse(L);

i=ClearList(&L);
```

```
    printf("\n 清空 L 后：ListLength(L)=%d\n",ListLength(L));
    CreateListHead(&L,20);
    printf("整体创建 L 的元素(头插法)：");
    ListTraverse(L);

    i=ClearList(&L);
    printf("\n 删除 L 后：ListLength(L)=%d\n",ListLength(L));
    CreateListTail(&L,20);
    printf("整体创建 L 的元素(尾插法)：");
    ListTraverse(L);


    return 0;
}
```

03 静态链表_StaticLinkList

```
#include "string.h"
#include "ctype.h"

#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 1000 /* 存储空间初始分配量 */

typedef int Status;                 /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */
typedef char ElemType;              /* ElemType 类型根据实际情况而定，这里假设为 char */


Status visit(ElemType c)
{
    printf("%c ",c);
    return OK;
}

/* 线性表的静态链表存储结构 */
```

```
typedef struct
{
	ElemType data;
	int cur;   /* 游标(Cursor) ，为 0 时表示无指向 */
} Component,StaticLinkList[MAXSIZE];


/* 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 */
Status InitList(StaticLinkList space)
{
	int i;
	for (i=0; i<MAXSIZE-1; i++)
		space[i].cur = i+1;
	space[MAXSIZE-1].cur = 0; /* 目前静态链表为空，最后一个元素的 cur 为 0 */
	return OK;
}


/* 若备用空间链表非空，则返回分配的结点下标，否则返回 0 */
int Malloc_SSL(StaticLinkList space)
{
	int i = space[0].cur;                    /* 当前数组第一个元素的 cur 存的值 */
                                             /* 就是要返回的第一个备用空闲的下标
*/
	if (space[0]. cur)
		space[0]. cur = space[i].cur;        /* 由于要拿出一个分量来使用了， */
                                             /* 所以我们就得把它的下一个 */
                                             /* 分量用来做备用 */
	return i;
}


/*  将下标为 k 的空闲结点回收到备用链表 */
void Free_SSL(StaticLinkList space, int k)
{
	space[k].cur = space[0].cur;      /* 把第一个元素的 cur 值赋给要删除的分量 cur */
	space[0].cur = k;                 /* 把要删除的分量下标赋值给第一个元素的 cur */
}

/* 初始条件：静态链表 L 已存在。操作结果：返回 L 中数据元素个数 */
int ListLength(StaticLinkList L)
{
	int j=0;
	int i=L[MAXSIZE-1].cur;
```

```
    while(i)
    {
        i=L[i].cur;
        j++;
    }
    return j;
}

/*  在 L 中第 i 个元素之前插入新的数据元素 e    */
Status ListInsert(StaticLinkList L, int i, ElemType e)
{
    int j, k, l;
    k = MAXSIZE - 1;    /* 注意 k 首先是最后一个元素的下标 */
    if (i < 1 || i > ListLength(L) + 1)
        return ERROR;
    j = Malloc_SSL(L);    /* 获得空闲分量的下标 */
    if (j)
    {
        L[j].data = e;    /* 将数据赋值给此分量的 data */
        for(l = 1; l <= i - 1; l++)    /* 找到第 i 个元素之前的位置 */
            k = L[k].cur;
        L[j].cur = L[k].cur;      /* 把第 i 个元素之前的 cur 赋值给新元素的 cur */
        L[k].cur = j;                  /* 把新元素的下标赋值给第 i 个元素之前元素的 ur */
        return OK;
    }
    return ERROR;
}

/*  删除在 L 中第 i 个数据元素    */
Status ListDelete(StaticLinkList L, int i)
{
    int j, k;
    if (i < 1 || i > ListLength(L))
        return ERROR;
    k = MAXSIZE - 1;
    for (j = 1; j <= i - 1; j++)
        k = L[k].cur;
    j = L[k].cur;
    L[k].cur = L[j].cur;
    Free_SSL(L, j);
    return OK;
}

Status ListTraverse(StaticLinkList L)
```

```
{
	int j=0;
	int i=L[MAXSIZE-1].cur;
	while(i)
	{
		visit(L[i].data);
		i=L[i].cur;
		j++;
	}
	return j;
	printf("\n");
	return OK;
}


int main()
{
	StaticLinkList L;
	Status i;
	i=InitList(L);
	printf("初始化 L 后：L.length=%d\n",ListLength(L));

	i=ListInsert(L,1,'F');
	i=ListInsert(L,1,'E');
	i=ListInsert(L,1,'D');
	i=ListInsert(L,1,'B');
	i=ListInsert(L,1,'A');

	printf("\n 在 L 的表头依次插入 FEDBA 后：\nL.data=");
	ListTraverse(L);

	i=ListInsert(L,3,'C');
	printf("\n 在 L 的“B”与“D”之间插入“C”后：\nL.data=");
	ListTraverse(L);

	i=ListDelete(L,1);
	printf("\n 在 L 的删除“A”后：\nL.data=");
	ListTraverse(L);

	printf("\n");

	return 0;
}
```

第四章 栈与队列

01 顺序栈_Stack

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define MAXSIZE 20 /* 存储空间初始分配量 */

typedef int Status;
typedef int SElemType; /* SElemType 类型根据实际情况而定，这里假设为 int */

/* 顺序栈结构 */
typedef struct
{
	SElemType data[MAXSIZE];
	int top; /* 用于栈顶指针 */
}SqStack;

Status visit(SElemType c)
{
	printf("%d ",c);
	return OK;
}

/*  构造一个空栈 S */
Status InitStack(SqStack *S)
{
	/* S.data=(SElemType *)malloc(MAXSIZE*sizeof(SElemType)); */
	S->top=-1;
	return OK;
}

/* 把 S 置为空栈 */
Status ClearStack(SqStack *S)
```

```
{
	S->top=-1;
	return OK;
}

/* 若栈 S 为空栈，则返回 TRUE，否则返回 FALSE */
Status StackEmpty(SqStack S)
{
	if (S.top==-1)
		return TRUE;
	else
		return FALSE;
}

/* 返回 S 的元素个数，即栈的长度 */
int StackLength(SqStack S)
{
	return S.top+1;
}

/* 若栈不空，则用 e 返回 S 的栈顶元素，并返回 OK；否则返回 ERROR */
Status GetTop(SqStack S,SElemType *e)
{
	if (S.top==-1)
		return ERROR;
	else
		*e=S.data[S.top];
	return OK;
}

/* 插入元素 e 为新的栈顶元素 */
Status Push(SqStack *S,SElemType e)
{
	if(S->top == MAXSIZE -1) /* 栈满 */
	{
		return ERROR;
	}
	S->top++;		/* 栈顶指针增加一 */
	S->data[S->top]=e;	/* 将新插入元素赋值给栈顶空间 */
	return OK;
}

/* 若栈不空，则删除 S 的栈顶元素，用 e 返回其值，并返回 OK；否则返回 ERROR */
Status Pop(SqStack *S,SElemType *e)
```

```
{
	if(S->top==-1)
		return ERROR;
	*e=S->data[S->top];	/* 将要删除的栈顶元素赋值给 e */
	S->top--;		/* 栈顶指针减一 */
	return OK;
}

/* 从栈底到栈顶依次对栈中每个元素显示 */
Status StackTraverse(SqStack S)
{
	int i;
	i=0;
	while(i<=S.top)
	{
		visit(S.data[i++]);
	}
	printf("\n");
	return OK;
}

int main()
{
	int j;
	SqStack s;
	int e;
	if(InitStack(&s)==OK)
		for(j=1;j<=10;j++)
			Push(&s,j);
	printf("栈中元素依次为：");
	StackTraverse(s);
	Pop(&s,&e);
	printf("弹出的栈顶元素 e=%d\n",e);
	printf("栈空否：%d(1:空 0:否)\n",StackEmpty(s));
	GetTop(s,&e);
	printf("栈顶元素 e=%d 栈的长度为%d\n",e,StackLength(s));
	ClearStack(&s);
	printf("清空栈后，栈空否：%d(1:空 0:否)\n",StackEmpty(s));

	return 0;
}
```

02 两栈共享空间_DoubleStack

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define MAXSIZE 20 /* 存储空间初始分配量 */

typedef int Status;

typedef int SElemType; /* SElemType 类型根据实际情况而定，这里假设为 int */


/* 两栈共享空间结构 */
typedef struct
{
	SElemType data[MAXSIZE];
	int top1;  /* 栈 1 栈顶指针 */
	int top2;  /* 栈 2 栈顶指针 */
}SqDoubleStack;


Status visit(SElemType c)
{
	printf("%d ",c);
	return OK;
}

/*  构造一个空栈 S */
Status InitStack(SqDoubleStack *S)
{
	S->top1=-1;
	S->top2=MAXSIZE;
	return OK;
}

/* 把 S 置为空栈 */
```

```
Status ClearStack(SqDoubleStack *S)
{
	S->top1=-1;
	S->top2=MAXSIZE;
	return OK;
}

/* 若栈 S 为空栈，则返回 TRUE，否则返回 FALSE */
Status StackEmpty(SqDoubleStack S)
{
	if (S.top1==-1 && S.top2==MAXSIZE)
		return TRUE;
	else
		return FALSE;
}

/* 返回 S 的元素个数，即栈的长度 */
int StackLength(SqDoubleStack S)
{
	return (S.top1+1)+(MAXSIZE-1-S.top2);
}

/* 插入元素 e 为新的栈顶元素 */
Status Push(SqDoubleStack *S,SElemType e,int stackNumber)
{
	if (S->top1+1==S->top2) /* 栈已满，不能再 push 新元素了 */
		return ERROR;
	if (stackNumber==1)		/* 栈 1 有元素进栈 */
		S->data[++S->top1]=e; /* 若是栈 1 则先 top1+1 后给数组元素赋值。 */
	else if (stackNumber==2)	/* 栈 2 有元素进栈 */
		S->data[--S->top2]=e; /* 若是栈 2 则先 top2-1 后给数组元素赋值。 */
	return OK;
}

/* 若栈不空，则删除 S 的栈顶元素，用 e 返回其值，并返回 OK；否则返回 ERROR */
Status Pop(SqDoubleStack *S,SElemType *e,int stackNumber)
{
	if (stackNumber==1)
	{
		if (S->top1==-1)
			return ERROR; /* 说明栈 1 已经是空栈，溢出 */
		*e=S->data[S->top1--]; /* 将栈 1 的栈顶元素出栈 */
	}
	else if (stackNumber==2)
```

```
        {
                if (S->top2==MAXSIZE)
                        return ERROR; /* 说明栈 2 已经是空栈，溢出 */
                *e=S->data[S->top2++]; /* 将栈 2 的栈顶元素出栈 */
        }
        return OK;
}

Status StackTraverse(SqDoubleStack S)
{
        int i;
        i=0;
        while(i<S.top1)
        {
                visit(S.data[i++]);
        }
        i=S.top2;
        while(i<MAXSIZE)
        {
                visit(S.data[i++]);
        }
        printf("\n");
        return OK;
}

int main()
{
        int j;
        SqDoubleStack s;
        int e;
        if(InitStack(&s)==OK)
        {
                for(j=1;j<=5;j++)
                        Push(&s,j,1);
                for(j=MAXSIZE;j>=MAXSIZE-2;j--)
                        Push(&s,j,2);
        }

        printf("栈中元素依次为：");
        StackTraverse(s);

        printf("当前栈中元素有：%d \n",StackLength(s));

        Pop(&s,&e,2);
```

```
	printf("弹出的栈顶元素 e=%d\n",e);
	printf("栈空否：%d(1:空 0:否)\n",StackEmpty(s));

	for(j=6;j<=MAXSIZE-2;j++)
		Push(&s,j,1);

	printf("栈中元素依次为：");
	StackTraverse(s);

	printf("栈满否：%d(1:否 0:满)\n",Push(&s,100,1));


	ClearStack(&s);
	printf("清空栈后，栈空否：%d(1:空 0:否)\n",StackEmpty(s));

	return 0;
}
```

03 链栈_LinkStack

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define MAXSIZE 20 /* 存储空间初始分配量 */

typedef int Status;
typedef int SElemType; /* SElemType 类型根据实际情况而定，这里假设为 int */


/* 链栈结构 */
typedef struct StackNode
{
	SElemType data;
	struct StackNode *next;
}StackNode,*LinkStackPtr;
```

```
typedef struct
{
	LinkStackPtr top;
	int count;
}LinkStack;

Status visit(SElemType c)
{
	printf("%d ",c);
	return OK;
}

/*  构造一个空栈 S */
Status InitStack(LinkStack *S)
{
	S->top = (LinkStackPtr)malloc(sizeof(StackNode));
	if(!S->top)
		return ERROR;
	S->top=NULL;
	S->count=0;
	return OK;
}

/* 把 S 置为空栈  */
Status ClearStack(LinkStack *S)
{
	LinkStackPtr p,q;
	p=S->top;
	while(p)
	{
		q=p;
		p=p->next;
		free(q);
	}
	S->count=0;
	return OK;
}

/* 若栈 S 为空栈，则返回 TRUE，否则返回 FALSE */
Status StackEmpty(LinkStack S)
{
	if (S.count==0)
		return TRUE;
	else
```

```
        return FALSE;
}

/* 返回 S 的元素个数，即栈的长度 */
int StackLength(LinkStack S)
{
    return S.count;
}

/* 若栈不空，则用 e 返回 S 的栈顶元素，并返回 OK；否则返回 ERROR */
Status GetTop(LinkStack S,SElemType *e)
{
    if (S.top==NULL)
        return ERROR;
    else
        *e=S.top->data;
    return OK;
}

/* 插入元素 e 为新的栈顶元素 */
Status Push(LinkStack *S,SElemType e)
{
    LinkStackPtr s=(LinkStackPtr)malloc(sizeof(StackNode));
    s->data=e;
    s->next=S->top;     /* 把当前的栈顶元素赋值给新结点的直接后继，见图中① */
    S->top=s;           /* 将新的结点 s 赋值给栈顶指针，见图中② */
    S->count++;
    return OK;
}

/* 若栈不空，则删除 S 的栈顶元素，用 e 返回其值，并返回 OK；否则返回 ERROR */
Status Pop(LinkStack *S,SElemType *e)
{
    LinkStackPtr p;
    if(StackEmpty(*S))
        return ERROR;
    *e=S->top->data;
    p=S->top;                   /* 将栈顶结点赋值给 p，见图中③ */
    S->top=S->top->next;        /* 使得栈顶指针下移一位，指向后一结点，见图中④ */
    free(p);                    /* 释放结点 p */
    S->count--;
    return OK;
}
```

```
Status StackTraverse(LinkStack S)
{
	LinkStackPtr p;
	p=S.top;
	while(p)
	{
		visit(p->data);
		p=p->next;
	}
	printf("\n");
	return OK;
}

int main()
{
	int j;
	LinkStack s;
	int e;
	if(InitStack(&s)==OK)
		for(j=1;j<=10;j++)
			Push(&s,j);
	printf("栈中元素依次为：");
	StackTraverse(s);
	Pop(&s,&e);
	printf("弹出的栈顶元素 e=%d\n",e);
	printf("栈空否：%d(1:空 0:否)\n",StackEmpty(s));
	GetTop(s,&e);
	printf("栈顶元素 e=%d 栈的长度为%d\n",e,StackLength(s));
	ClearStack(&s);
	printf("清空栈后，栈空否：%d(1:空 0:否)\n",StackEmpty(s));
	return 0;
}
```

04 斐波那契函数_Fibonacci

```
#include "stdio.h"

int Fbi(int i)   /* 斐波那契的递归函数 */
{
    if( i < 2 )
        return i == 0 ? 0 : 1;
    return Fbi(i - 1) + Fbi(i - 2);   /* 这里 Fbi 就是函数自己，等于在调用自己 */
}
```

```
int main()
{
	int i;
	int a[40];
	printf("迭代显示斐波那契数列：\n");
	a[0]=0;
	a[1]=1;
	printf("%d ",a[0]);
	printf("%d ",a[1]);
	for(i = 2;i < 40;i++)
	{
		a[i] = a[i-1] + a[i-2];
		printf("%d ",a[i]);
	}
	printf("\n");

	printf("递归显示斐波那契数列：\n");
	for(i = 0;i < 40;i++)
		printf("%d ", Fbi(i));
	return 0;
}
```

05 顺序队列_Queue

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define MAXSIZE 20 /* 存储空间初始分配量 */

typedef int Status;
typedef int QElemType; /* QElemType 类型根据实际情况而定，这里假设为 int */

/* 循环队列的顺序存储结构 */
typedef struct
{
	QElemType data[MAXSIZE];
	int front;		/* 头指针 */
```

```
    int rear;        /* 尾指针，若队列不空，指向队列尾元素的下一个位置 */
}SqQueue;

Status visit(QElemType c)
{
    printf("%d ",c);
    return OK;
}

/* 初始化一个空队列 Q */
Status InitQueue(SqQueue *Q)
{
    Q->front=0;
    Q->rear=0;
    return  OK;
}

/* 将 Q 清为空队列 */
Status ClearQueue(SqQueue *Q)
{
    Q->front=Q->rear=0;
    return OK;
}

/* 若队列 Q 为空队列,则返回 TRUE,否则返回 FALSE */
Status QueueEmpty(SqQueue Q)
{
    if(Q.front==Q.rear) /* 队列空的标志 */
        return TRUE;
    else
        return FALSE;
}

/* 返回 Q 的元素个数，也就是队列的当前长度 */
int QueueLength(SqQueue Q)
{
    return  (Q.rear-Q.front+MAXSIZE)%MAXSIZE;
}

/* 若队列不空,则用 e 返回 Q 的队头元素,并返回 OK,否则返回 ERROR */
Status GetHead(SqQueue Q,QElemType *e)
{
    if(Q.front==Q.rear) /* 队列空 */
        return ERROR;
```

```
    *e=Q.data[Q.front];
    return OK;
}

/* 若队列未满，则插入元素 e 为 Q 新的队尾元素 */
Status EnQueue(SqQueue *Q,QElemType e)
{
    if ((Q->rear+1)%MAXSIZE == Q->front)  /* 队列满的判断 */
        return ERROR;
    Q->data[Q->rear]=e;                   /* 将元素 e 赋值给队尾 */
    Q->rear=(Q->rear+1)%MAXSIZE;/* rear 指针向后移一位置， */
                                          /* 若到最后则转到数组头部 */
    return  OK;
}

/* 若队列不空，则删除 Q 中队头元素，用 e 返回其值 */
Status DeQueue(SqQueue *Q,QElemType *e)
{
    if (Q->front == Q->rear)              /* 队列空的判断 */
        return ERROR;
    *e=Q->data[Q->front];                 /* 将队头元素赋值给 e */
    Q->front=(Q->front+1)%MAXSIZE; /* front 指针向后移一位置， */
                                          /* 若到最后则转到数组头部 */
    return  OK;
}

/* 从队头到队尾依次对队列 Q 中每个元素输出 */
Status QueueTraverse(SqQueue Q)
{
    int i;
    i=Q.front;
    while((i+Q.front)!=Q.rear)
    {
        visit(Q.data[i]);
        i=(i+1)%MAXSIZE;
    }
    printf("\n");
    return OK;
}

int main()
{
    Status j;
    int i=0,l;
```

```
QElemType d;
SqQueue Q;
InitQueue(&Q);
printf("初始化队列后，队列空否？%u(1:空 0:否)\n",QueueEmpty(Q));

printf("请输入整型队列元素(不超过%d 个),-1 为提前结束符: ",MAXSIZE-1);
do
{
    /* scanf("%d",&d); */
    d=i+100;
    if(d==-1)
        break;
    i++;
    EnQueue(&Q,d);
}while(i<MAXSIZE-1);

printf("队列长度为: %d\n",QueueLength(Q));
printf("现在队列空否？%u(1:空 0:否)\n",QueueEmpty(Q));
printf("连续%d 次由队头删除元素,队尾插入元素:\n",MAXSIZE);
for(l=1;l<=MAXSIZE;l++)
{
    DeQueue(&Q,&d);
    printf("删除的元素是%d,插入的元素:%d \n",d,l+1000);
    /* scanf("%d",&d); */
    d=l+1000;
    EnQueue(&Q,d);
}
l=QueueLength(Q);

printf("现在队列中的元素为: \n");
QueueTraverse(Q);
printf("共向队尾插入了%d 个元素\n",i+MAXSIZE);
if(l-2>0)
    printf("现在由队头删除%d 个元素:\n",l-2);
while(QueueLength(Q)>2)
{
    DeQueue(&Q,&d);
    printf("删除的元素值为%d\n",d);
}

j=GetHead(Q,&d);
if(j)
    printf("现在队头元素为: %d\n",d);
ClearQueue(&Q);
```

```
    printf("清空队列后，队列空否？%u(1:空 0:否)\n",QueueEmpty(Q));
    return 0;
}
```

06 链队列_LinkQueue

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define MAXSIZE 20 /* 存储空间初始分配量 */

typedef int Status;

typedef int QElemType; /* QElemType 类型根据实际情况而定，这里假设为 int */

typedef struct QNode    /* 结点结构 */
{
    QElemType data;
    struct QNode *next;
}QNode,*QueuePtr;

typedef struct              /* 队列的链表结构 */
{
    QueuePtr front,rear; /* 队头、队尾指针 */
}LinkQueue;

Status visit(QElemType c)
{
    printf("%d ",c);
    return OK;
}

/* 构造一个空队列 Q */
Status InitQueue(LinkQueue *Q)
{
    Q->front=Q->rear=(QueuePtr)malloc(sizeof(QNode));
    if(!Q->front)
```

```
        exit(OVERFLOW);
    Q->front->next=NULL;
    return OK;
}

/*  销毁队列 Q */
Status DestroyQueue(LinkQueue *Q)
{
    while(Q->front)
    {
        Q->rear=Q->front->next;
        free(Q->front);
        Q->front=Q->rear;
    }
    return OK;
}

/*  将 Q 清为空队列  */
Status ClearQueue(LinkQueue *Q)
{
    QueuePtr p,q;
    Q->rear=Q->front;
    p=Q->front->next;
    Q->front->next=NULL;
    while(p)
    {
        q=p;
        p=p->next;
        free(q);
    }
    return OK;
}

/*  若 Q 为空队列,则返回 TRUE,否则返回 FALSE */
Status QueueEmpty(LinkQueue Q)
{
    if(Q.front==Q.rear)
        return TRUE;
    else
        return FALSE;
}

/*  求队列的长度  */
int QueueLength(LinkQueue Q)
```

```
{
    int i=0;
    QueuePtr p;
    p=Q.front;
    while(Q.rear!=p)
    {
        i++;
        p=p->next;
    }
    return i;
}

/* 若队列不空,则用 e 返回 Q 的队头元素,并返回 OK,否则返回 ERROR */
Status GetHead(LinkQueue Q,QElemType *e)
{
    QueuePtr p;
    if(Q.front==Q.rear)
        return ERROR;
    p=Q.front->next;
    *e=p->data;
    return OK;
}


/* 插入元素 e 为 Q 的新的队尾元素 */
Status EnQueue(LinkQueue *Q,QElemType e)
{
    QueuePtr s=(QueuePtr)malloc(sizeof(QNode));
    if(!s) /* 存储分配失败 */
        exit(OVERFLOW);
    s->data=e;
    s->next=NULL;
    Q->rear->next=s;    /* 把拥有元素 e 的新结点 s 赋值给原队尾结点的后继，见图中① */
    Q->rear=s;          /* 把当前的 s 设置为队尾结点，rear 指向 s，见图中② */
    return OK;
}

/* 若队列不空,删除 Q 的队头元素,用 e 返回其值,并返回 OK,否则返回 ERROR */
Status DeQueue(LinkQueue *Q,QElemType *e)
{
    QueuePtr p;
    if(Q->front==Q->rear)
        return ERROR;
    p=Q->front->next;       /* 将欲删除的队头结点暂存给 p，见图中① */
```

```
    *e=p->data;                    /* 将欲删除的队头结点的值赋值给 e */
    Q->front->next=p->next;/* 将原队头结点的后继 p->next 赋值给头结点后继，见图中②
*/
    if(Q->rear==p)      /* 若队头就是队尾，则删除后将 rear 指向头结点，见图中③ */
        Q->rear=Q->front;
    free(p);
    return OK;
}

/* 从队头到队尾依次对队列 Q 中每个元素输出 */
Status QueueTraverse(LinkQueue Q)
{
    QueuePtr p;
    p=Q.front->next;
    while(p)
    {
        visit(p->data);
        p=p->next;
    }
    printf("\n");
    return OK;
}

int main()
{
    int i;
    QElemType d;
    LinkQueue q;
    i=InitQueue(&q);
    if(i)
        printf("成功地构造了一个空队列!\n");
    printf("是否空队列？%d(1:空 0:否)  ",QueueEmpty(q));
    printf("队列的长度为%d\n",QueueLength(q));
    EnQueue(&q,-5);
    EnQueue(&q,5);
    EnQueue(&q,10);
    printf("插入 3 个元素(-5,5,10)后,队列的长度为%d\n",QueueLength(q));
    printf("是否空队列？%d(1:空 0:否)  ",QueueEmpty(q));
    printf("队列的元素依次为：");
    QueueTraverse(q);
    i=GetHead(q,&d);
    if(i==OK)
     printf("队头元素是：%d\n",d);
    DeQueue(&q,&d);
```

```
    printf("删除了队头元素%d\n",d);
    i=GetHead(q,&d);
    if(i==OK)
        printf("新的队头元素是：%d\n",d);
    ClearQueue(&q);
    printf("清空队列后,q.front=%u q.rear=%u q.front->next=%u\n",q.front,q.rear,q.front->next);
    DestroyQueue(&q);
    printf("销毁队列后,q.front=%u q.rear=%u\n",q.front, q.rear);

    return 0;
}
```

第五章 串

01 串_String

```
#include "string.h"
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 40 /* 存储空间初始分配量 */

typedef int Status;        /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */
typedef int ElemType;    /* ElemType 类型根据实际情况而定，这里假设为 int */

typedef char String[MAXSIZE+1]; /*   0 号单元存放串的长度 */

/* 生成一个其值等于 chars 的串 T */
Status StrAssign(String T,char *chars)
{
    int i;
    if(strlen(chars)>MAXSIZE)
        return ERROR;
```

```
    else
    {
        T[0]=strlen(chars);
        for(i=1;i<=T[0];i++)
            T[i]=*(chars+i-1);
        return OK;
    }
}

/* 由串 S 复制得串 T */
Status StrCopy(String T,String S)
{
    int i;
    for(i=0;i<=S[0];i++)
        T[i]=S[i];
    return OK;
}

/* 若 S 为空串,则返回 TRUE,否则返回 FALSE */
Status StrEmpty(String S)
{
    if(S[0]==0)
        return TRUE;
    else
        return FALSE;
}

/*   初始条件: 串 S 和 T 存在 */
/*   操作结果: 若 S>T,则返回值>0;若 S=T,则返回值=0;若 S<T,则返回值<0 */
int StrCompare(String S,String T)
{
    int i;
    for(i=1;i<=S[0]&&i<=T[0];++i)
        if(S[i]!=T[i])
            return S[i]-T[i];
    return S[0]-T[0];
}

/* 返回串的元素个数 */
int StrLength(String S)
{
    return S[0];
}
```

```
/* 初始条件:串 S 存在。操作结果:将 S 清为空串 */
Status ClearString(String S)
{
    S[0]=0;/*  令串长为零 */
    return OK;
}

/* 用 T 返回 S1 和 S2 联接而成的新串。若未截断，则返回 TRUE，否则 FALSE */
Status Concat(String T,String S1,String S2)
{
    int i;
    if(S1[0]+S2[0]<=MAXSIZE)
    { /*  未截断 */
        for(i=1;i<=S1[0];i++)
            T[i]=S1[i];
        for(i=1;i<=S2[0];i++)
            T[S1[0]+i]=S2[i];
        T[0]=S1[0]+S2[0];
        return TRUE;
    }
    else
    { /*  截断 S2 */
        for(i=1;i<=S1[0];i++)
            T[i]=S1[i];
        for(i=1;i<=MAXSIZE-S1[0];i++)
            T[S1[0]+i]=S2[i];
        T[0]=MAXSIZE;
        return FALSE;
    }
}

/* 用 Sub 返回串 S 的第 pos 个字符起长度为 len 的子串。 */
Status SubString(String Sub,String S,int pos,int len)
{
    int i;
    if(pos<1||pos>S[0]||len<0||len>S[0]-pos+1)
        return ERROR;
    for(i=1;i<=len;i++)
        Sub[i]=S[pos+i-1];
    Sub[0]=len;
    return OK;
}

/* 返回子串 T 在主串 S 中第 pos 个字符之后的位置。若不存在,则函数返回值为 0。 */
```

```
/*  其中,T 非空,1≤pos≤StrLength(S)。  */
int Index(String S, String T, int pos)
{
     int i = pos;      /*  i 用于主串 S 中当前位置下标值，若 pos 不为 1，则从 pos 位置开始匹
配  */
     int j = 1;                    /* j 用于子串 T 中当前位置下标值  */
     while (i <= S[0] && j <= T[0]) /*  若 i 小于 S 的长度并且 j 小于 T 的长度时，循环继续  */
     {
          if (S[i] == T[j])  /*  两字母相等则继续  */
          {
               ++i;
               ++j;
          }
          else                         /*  指针后退重新开始匹配  */
          {
               i = i-j+2;         /* i 退回到上次匹配首位的下一位  */
               j = 1;                    /* j 退回到子串 T 的首位  */
          }
     }
     if (j > T[0])
          return i-T[0];
     else
          return 0;
}


/*   T 为非空串。若主串 S 中第 pos 个字符之后存在与 T 相等的子串，  */
/*   则返回第一个这样的子串在 S 中的位置，否则返回 0 */
int Index2(String S, String T, int pos)
{
     int n,m,i;
     String sub;
     if (pos > 0)
     {
          n = StrLength(S);    /*  得到主串 S 的长度  */
          m = StrLength(T);   /*  得到子串 T 的长度  */
          i = pos;
          while (i <= n-m+1)
          {
               SubString (sub, S, i, m);  /*  取主串中第 i 个位置长度与 T 相等的子串给 sub */
               if (StrCompare(sub,T) != 0)       /*  如果两串不相等  */
                    ++i;
               else                          /*  如果两串相等  */
                    return i;         /*  则返回 i 值  */
```

```
            }
        }
        return 0; /* 若无子串与 T 相等，返回 0 */
}


/*   初始条件: 串 S 和 T 存在,1≤pos≤StrLength(S)+1 */
/*   操作结果: 在串 S 的第 pos 个字符之前插入串 T。完全插入返回 TRUE,部分插入返回 FALSE
*/
Status StrInsert(String S,int pos,String T)
{
        int i;
        if(pos<1||pos>S[0]+1)
                return ERROR;
        if(S[0]+T[0]<=MAXSIZE)
        { /*   完全插入 */
                for(i=S[0];i>=pos;i--)
                        S[i+T[0]]=S[i];
                for(i=pos;i<pos+T[0];i++)
                        S[i]=T[i-pos+1];
                S[0]=S[0]+T[0];
                return TRUE;
        }
        else
        { /*   部分插入 */
                for(i=MAXSIZE;i<=pos;i--)
                        S[i]=S[i-T[0]];
                for(i=pos;i<pos+T[0];i++)
                        S[i]=T[i-pos+1];
                S[0]=MAXSIZE;
                return FALSE;
        }
}

/*   初始条件: 串 S 存在,1≤pos≤StrLength(S)-len+1 */
/*   操作结果: 从串 S 中删除第 pos 个字符起长度为 len 的子串 */
Status StrDelete(String S,int pos,int len)
{
        int i;
        if(pos<1||pos>S[0]-len+1||len<0)
                return ERROR;
        for(i=pos+len;i<=S[0];i++)
                S[i-len]=S[i];
        S[0]-=len;
```

```
    return OK;
}

/*   初始条件: 串 S,T 和 V 存在,T 是非空串（此函数与串的存储结构无关） */
/*   操作结果: 用 V 替换主串 S 中出现的所有与 T 相等的不重叠的子串 */
Status Replace(String S,String T,String V)
{
    int i=1; /*   从串 S 的第一个字符起查找串 T */
    if(StrEmpty(T)) /*   T 是空串 */
        return ERROR;
    do
    {
        i=Index(S,T,i); /*   结果 i 为从上一个 i 之后找到的子串 T 的位置 */
        if(i) /*   串 S 中存在串 T */
        {
            StrDelete(S,i,StrLength(T)); /*   删除该串 T */
            StrInsert(S,i,V); /*   在原串 T 的位置插入串 V */
            i+=StrLength(V); /*   在插入的串 V 后面继续查找串 T */
        }
    }while(i);
    return OK;
}

/*   输出字符串 T */
void StrPrint(String T)
{
    int i;
    for(i=1;i<=T[0];i++)
        printf("%c",T[i]);
    printf("\n");
}


int main()
{

    int i,j;
    Status k;
    char s;
    String t,s1,s2;
    printf("请输入串 s1: ");

    k=StrAssign(s1,"abcd");
    if(!k)
```

```
{
    printf("串长超过 MAXSIZE(=%d)\n",MAXSIZE);
    exit(0);
}
printf("串长为%d  串空否？%d(1:是  0:否)\n",StrLength(s1),StrEmpty(s1));
StrCopy(s2,s1);
printf("拷贝 s1 生成的串为: ");
StrPrint(s2);
printf("请输入串 s2: ");

k=StrAssign(s2,"efghijk");
if(!k)
{
    printf("串长超过 MAXSIZE(%d)\n",MAXSIZE);
    exit(0);
}
i=StrCompare(s1,s2);
if(i<0)
    s='<';
else if(i==0)
    s='=';
else
    s='>';
printf("串 s1%c 串 s2\n",s);
k=Concat(t,s1,s2);
printf("串 s1 联接串 s2 得到的串 t 为: ");
StrPrint(t);
if(k==FALSE)
    printf("串 t 有截断\n");
ClearString(s1);
printf("清为空串后,串 s1 为: ");
StrPrint(s1);
printf("串长为%d  串空否？%d(1:是  0:否)\n",StrLength(s1),StrEmpty(s1));
printf("求串 t 的子串,请输入子串的起始位置,子串长度: ");

i=2;
j=3;
printf("%d,%d \n",i,j);

k=SubString(s2,t,i,j);
if(k)
{
    printf("子串 s2 为: ");
    StrPrint(s2);
```

```
    }
    printf("从串 t 的第 pos 个字符起,删除 len 个字符，请输入 pos,len: ");

    i=4;
    j=2;
    printf("%d,%d \n",i,j);


    StrDelete(t,i,j);
    printf("删除后的串 t 为: ");
    StrPrint(t);
    i=StrLength(s2)/2;
    StrInsert(s2,i,t);
    printf("在串 s2 的第%d 个字符之前插入串 t 后,串 s2 为:\n",i);
    StrPrint(s2);
    i=Index(s2,t,1);
    printf("s2 的第%d 个字母起和 t 第一次匹配\n",i);
    SubString(t,s2,1,1);
    printf("串 t 为：");
    StrPrint(t);
    Concat(s1,t,t);
    printf("串 s1 为：");
    StrPrint(s1);
    Replace(s2,t,s1);
    printf("用串 s1 取代串 s2 中和串 t 相同的不重叠的串后,串 s2 为: ");
    StrPrint(s2);


    return 0;
}
```

02 模式匹配_KMP

```
#include "string.h"
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
```

```
#define MAXSIZE 100 /* 存储空间初始分配量 */

typedef int Status;        /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */
typedef int ElemType;    /* ElemType 类型根据实际情况而定，这里假设为 int */

typedef char String[MAXSIZE+1]; /*    0 号单元存放串的长度 */

/* 生成一个其值等于 chars 的串 T */
Status StrAssign(String T,char *chars)
{
      int i;
      if(strlen(chars)>MAXSIZE)
            return ERROR;
      else
      {
            T[0]=strlen(chars);
            for(i=1;i<=T[0];i++)
                  T[i]=*(chars+i-1);
            return OK;
      }
}

Status ClearString(String S)
{
      S[0]=0;/*   令串长为零 */
      return OK;
}

/*   输出字符串 T。 */
void StrPrint(String T)
{
      int i;
      for(i=1;i<=T[0];i++)
            printf("%c",T[i]);
      printf("\n");
}

/*   输出 Next 数组值。 */
void NextPrint(int next[],int length)
{
      int i;
      for(i=1;i<=length;i++)
            printf("%d",next[i]);
      printf("\n");
```

```
}

/* 返回串的元素个数 */
int StrLength(String S)
{
    return S[0];
}

/* 朴素的模式匹配法 */
int Index(String S, String T, int pos)
{
    int i = pos;    /* i 用于主串 S 中当前位置下标值，若 pos 不为 1，则从 pos 位置开始匹配 */
    int j = 1;                  /* j 用于子串 T 中当前位置下标值 */
    while (i <= S[0] && j <= T[0]) /* 若 i 小于 S 的长度并且 j 小于 T 的长度时，循环继续 */
    {
        if (S[i] == T[j])  /* 两字母相等则继续 */
        {
            ++i;
            ++j;
        }
        else                        /* 指针后退重新开始匹配 */
        {
            i = i-j+2;      /* i 退回到上次匹配首位的下一位 */
            j = 1;                  /* j 退回到子串 T 的首位 */
        }
    }
    if (j > T[0])
        return i-T[0];
    else
        return 0;
}

/* 通过计算返回子串 T 的 next 数组。 */
void get_next(String T, int *next)
{
    int i,j;
    i=1;
    j=0;
    next[1]=0;
    while (i<T[0])   /* 此处 T[0]表示串 T 的长度 */
    {
        if(j==0 || T[i]== T[j])     /* T[i]表示后缀的单个字符，T[j]表示前缀的单个字符 */
        {
```

```
            ++i;
            ++j;
            next[i] = j;
        }
        else
            j= next[j];/* 若字符不相同，则 j 值回溯 */
    }
}

/* 返回子串 T 在主串 S 中第 pos 个字符之后的位置。若不存在，则函数返回值为 0。 */
/*  T 非空，1≤pos≤StrLength(S)。 */
int Index_KMP(String S, String T, int pos)
{
    int i = pos;        /* i 用于主串 S 中当前位置下标值，若 pos 不为 1，则从 pos 位置开始匹配 */
    int j = 1;          /* j 用于子串 T 中当前位置下标值 */
    int next[255];      /* 定义一 next 数组 */
    get_next(T, next);  /* 对串 T 作分析，得到 next 数组 */
    while (i <= S[0] && j <= T[0]) /* 若 i 小于 S 的长度并且 j 小于 T 的长度时，循环继续 */
    {
        if (j==0 || S[i] == T[j])   /* 两字母相等则继续，与朴素算法增加了 j=0 判断 */
        {
            ++i;
            ++j;
        }
        else                /* 指针后退重新开始匹配 */
            j = next[j];/* j 退回合适的位置，i 值不变 */
    }
    if (j > T[0])
        return i-T[0];
    else
        return 0;
}

/* 求模式串 T 的 next 函数修正值并存入数组 nextval */
void get_nextval(String T, int *nextval)
{
    int i,j;
    i=1;
    j=0;
    nextval[1]=0;
    while (i<T[0])  /* 此处 T[0]表示串 T 的长度 */
    {
        if(j==0 || T[i]== T[j])     /* T[i]表示后缀的单个字符，T[j]表示前缀的单个字符 */
```

```
        {
            ++i;
            ++j;
            if (T[i]!=T[j])          /* 若当前字符与前缀字符不同 */
                nextval[i] = j;   /* 则当前的 j 为 nextval 在 i 位置的值 */
            else
                nextval[i] = nextval[j];    /* 如果与前缀字符相同，则将前缀字符的 */
                                            /* nextval 值赋值给 nextval 在 i 位置的值
*/
        }
        else
            j= nextval[j];              /* 若字符不相同，则 j 值回溯 */
    }
}

int Index_KMP1(String S, String T, int pos)
{
    int i = pos;            /* i 用于主串 S 中当前位置下标值，若 pos 不为 1，则从 pos 位置开
始匹配 */
    int j = 1;              /* j 用于子串 T 中当前位置下标值 */
    int next[255];          /* 定义一 next 数组 */
    get_nextval(T, next);     /* 对串 T 作分析，得到 next 数组 */
    while (i <= S[0] && j <= T[0]) /* 若 i 小于 S 的长度并且 j 小于 T 的长度时，循环继续 */
    {
        if (j==0 || S[i] == T[j])    /* 两字母相等则继续，与朴素算法增加了 j=0 判断 */
        {
            ++i;
            ++j;
        }
        else                    /* 指针后退重新开始匹配 */
            j = next[j];/* j 退回合适的位置，i 值不变 */
    }
    if (j > T[0])
        return i-T[0];
    else
        return 0;
}

int main()
{
    int i,*p;
    String s1,s2;

    StrAssign(s1,"abcdex");
```

```
printf("子串为: ");
StrPrint(s1);
i=StrLength(s1);
p=(int*)malloc((i+1)*sizeof(int));
get_next(s1,p);
printf("Next 为: ");
NextPrint(p,StrLength(s1));
printf("\n");

StrAssign(s1,"abcabx");
printf("子串为: ");
StrPrint(s1);
i=StrLength(s1);
p=(int*)malloc((i+1)*sizeof(int));
get_next(s1,p);
printf("Next 为: ");
NextPrint(p,StrLength(s1));
printf("\n");

StrAssign(s1,"ababaaaba");
printf("子串为: ");
StrPrint(s1);
i=StrLength(s1);
p=(int*)malloc((i+1)*sizeof(int));
get_next(s1,p);
printf("Next 为: ");
NextPrint(p,StrLength(s1));
printf("\n");

StrAssign(s1,"aaaaaaaab");
printf("子串为: ");
StrPrint(s1);
i=StrLength(s1);
p=(int*)malloc((i+1)*sizeof(int));
get_next(s1,p);
printf("Next 为: ");
NextPrint(p,StrLength(s1));
printf("\n");

StrAssign(s1,"ababaaaba");
printf("      子串为: ");
StrPrint(s1);
i=StrLength(s1);
p=(int*)malloc((i+1)*sizeof(int));
```

```
    get_next(s1,p);
    printf("      Next 为: ");
    NextPrint(p,StrLength(s1));
    get_nextval(s1,p);
    printf("NextVal 为: ");
    NextPrint(p,StrLength(s1));
    printf("\n");

    StrAssign(s1,"aaaaaaaab");
    printf("      子串为: ");
    StrPrint(s1);
    i=StrLength(s1);
    p=(int*)malloc((i+1)*sizeof(int));
    get_next(s1,p);
    printf("      Next 为: ");
    NextPrint(p,StrLength(s1));
    get_nextval(s1,p);
    printf("NextVal 为: ");
    NextPrint(p,StrLength(s1));

    printf("\n");

    StrAssign(s1,"000000000000000000000000000000000000000000000000001");
    printf("主串为: ");
    StrPrint(s1);
    StrAssign(s2,"0000000001");
    printf("子串为: ");
    StrPrint(s2);
    printf("\n");
    printf("主串和子串在第%d 个字符处首次匹配（朴素模式匹配算法）\n",Index(s1,s2,1));
    printf("主串和子串在第%d 个字符处首次匹配（KMP 算法） \n",Index_KMP(s1,s2,1));
    printf("主串和子串在第%d 个字符处首次匹配(KMP 改良算法）\n",Index_KMP1(s1,s2,1));

    return 0;
}
```

第六章 树

01 二叉树顺序结构实现_BiTreeArray

```
#include "stdio.h"
#include "stdlib.h"
```

```
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 100 /* 存储空间初始分配量 */
#define MAX_TREE_SIZE 100 /* 二叉树的最大结点数 */

typedef int Status;        /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */
typedef int TElemType;   /* 树结点的数据类型，目前暂定为整型 */
typedef TElemType SqBiTree[MAX_TREE_SIZE]; /* 0 号单元存储根结点  */

typedef struct
{
    int level,order; /* 结点的层,本层序号(按满二叉树计算) */
}Position;

TElemType Nil=0; /*  设整型以 0 为空 */

Status visit(TElemType c)
{
    printf("%d ",c);
    return OK;
}

/* 构造空二叉树 T。因为 T 是固定数组，不会改变，故不需要& */
Status InitBiTree(SqBiTree T)
{
    int i;
    for(i=0;i<MAX_TREE_SIZE;i++)
        T[i]=Nil; /* 初值为空 */
    return OK;
}

/* 按层序次序输入二叉树中结点的值(字符型或整型), 构造顺序存储的二叉树 T */
Status CreateBiTree(SqBiTree T)
{
    int i=0;
    printf("请按层序输入结点的值(整型)，0 表示空结点，输 999 结束。结点数≤%d:\n",MAX_TREE_SIZE);
```

```
    while(i<10)
    {
        T[i]=i+1;

        if(i!=0&&T[(i+1)/2-1]==Nil&&T[i]!=Nil) /* 此结点(不空)无双亲且不是根 */
        {
            printf("出现无双亲的非根结点%d\n",T[i]);
            exit(ERROR);
        }
        i++;
    }
    while(i<MAX_TREE_SIZE)
    {
        T[i]=Nil; /* 将空赋值给 T 的后面的结点 */
        i++;
    }

    return OK;
}

#define ClearBiTree InitBiTree /* 在顺序存储结构中，两函数完全一样 */

/* 初始条件: 二叉树 T 存在 */
/* 操作结果: 若 T 为空二叉树,则返回 TRUE,否则 FALSE */
Status BiTreeEmpty(SqBiTree T)
{
    if(T[0]==Nil) /* 根结点为空,则树空 */
        return TRUE;
    else
        return FALSE;
}

/* 初始条件: 二叉树 T 存在。操作结果: 返回 T 的深度 */
int BiTreeDepth(SqBiTree T)
{
    int i,j=-1;
    for(i=MAX_TREE_SIZE-1;i>=0;i--) /* 找到最后一个结点 */
      if(T[i]!=Nil)
        break;
    i++;
    do
      j++;
    while(i>=powl(2,j));/* 计算 2 的 j 次幂。 */
    return j;
```

```
}

/* 初始条件: 二叉树 T 存在 */
/* 操作结果:  当 T 不空,用 e 返回 T 的根,返回 OK;否则返回 ERROR,e 无定义 */
Status Root(SqBiTree T,TElemType *e)
{
    if(BiTreeEmpty(T)) /* T 空 */
        return ERROR;
    else
    {
        *e=T[0];
        return OK;
    }
}

/* 初始条件: 二叉树 T 存在,e 是 T 中某个结点(的位置) */
/* 操作结果: 返回处于位置 e(层,本层序号)的结点的值 */
TElemType Value(SqBiTree T,Position e)
{
    return T[(int)powl(2,e.level-1)+e.order-2];
}

/* 初始条件: 二叉树 T 存在,e 是 T 中某个结点(的位置) */
/* 操作结果: 给处于位置 e(层,本层序号)的结点赋新值 value */
Status Assign(SqBiTree T,Position e,TElemType value)
{
    int i=(int)powl(2,e.level-1)+e.order-2; /* 将层、本层序号转为矩阵的序号 */
    if(value!=Nil&&T[(i+1)/2-1]==Nil) /* 给叶子赋非空值但双亲为空 */
        return ERROR;
    else if(value==Nil&&(T[i*2+1]!=Nil||T[i*2+2]!=Nil)) /*  给双亲赋空值但有叶子(不空) */
        return ERROR;
    T[i]=value;
    return OK;
}

/* 初始条件: 二叉树 T 存在,e 是 T 中某个结点 */
/* 操作结果: 若 e 是 T 的非根结点,则返回它的双亲,否则返回 " 空 " */
TElemType Parent(SqBiTree T,TElemType e)
{
    int i;
    if(T[0]==Nil) /* 空树 */
        return Nil;
    for(i=1;i<=MAX_TREE_SIZE-1;i++)
        if(T[i]==e) /* 找到 e */
```

```
            return T[(i+1)/2-1];
    return Nil; /*  没找到 e */
}


/*  初始条件: 二叉树 T 存在,e 是 T 中某个结点  */
/*  操作结果: 返回 e 的左孩子。若 e 无左孩子,则返回 " 空 "   */
TElemType LeftChild(SqBiTree T,TElemType e)
{
    int i;
    if(T[0]==Nil) /*  空树  */
        return Nil;
    for(i=0;i<=MAX_TREE_SIZE-1;i++)
        if(T[i]==e) /*  找到 e */
            return T[i*2+1];
    return Nil; /*  没找到 e */
}

/*  初始条件: 二叉树 T 存在,e 是 T 中某个结点  */
/*  操作结果: 返回 e 的右孩子。若 e 无右孩子,则返回 " 空 "   */
TElemType RightChild(SqBiTree T,TElemType e)
{
    int i;
    if(T[0]==Nil) /*  空树  */
        return Nil;
    for(i=0;i<=MAX_TREE_SIZE-1;i++)
        if(T[i]==e) /*  找到 e */
            return T[i*2+2];
    return Nil; /*  没找到 e */
}

/*  初始条件: 二叉树 T 存在,e 是 T 中某个结点  */
/*  操作结果: 返回 e 的左兄弟。若 e 是 T 的左孩子或无左兄弟,则返回 " 空 "   */
TElemType LeftSibling(SqBiTree T,TElemType e)
{
    int i;
    if(T[0]==Nil) /*  空树  */
        return Nil;
    for(i=1;i<=MAX_TREE_SIZE-1;i++)
        if(T[i]==e&&i%2==0) /*  找到 e 且其序号为偶数(是右孩子) */
            return T[i-1];
    return Nil; /*  没找到 e */
}
```

```
/* 初始条件: 二叉树 T 存在,e 是 T 中某个结点 */
/* 操作结果: 返回 e 的右兄弟。若 e 是 T 的右孩子或无右兄弟,则返回 " 空 " */
TElemType RightSibling(SqBiTree T,TElemType e)
{
    int i;
    if(T[0]==Nil) /* 空树 */
        return Nil;
    for(i=1;i<=MAX_TREE_SIZE-1;i++)
        if(T[i]==e&&i%2) /* 找到 e 且其序号为奇数(是左孩子) */
            return T[i+1];
    return Nil; /* 没找到 e */
}

/* PreOrderTraverse()调用 */
void PreTraverse(SqBiTree T,int e)
{
    visit(T[e]);
    if(T[2*e+1]!=Nil) /* 左子树不空 */
        PreTraverse(T,2*e+1);
    if(T[2*e+2]!=Nil) /* 右子树不空 */
        PreTraverse(T,2*e+2);
}

/* 初始条件: 二叉树存在 */
/* 操作结果: 先序遍历 T。 */
Status PreOrderTraverse(SqBiTree T)
{
    if(!BiTreeEmpty(T)) /* 树不空 */
      PreTraverse(T,0);
    printf("\n");
    return OK;
}

/* InOrderTraverse()调用 */
void InTraverse(SqBiTree T,int e)
{
    if(T[2*e+1]!=Nil) /* 左子树不空 */
        InTraverse(T,2*e+1);
    visit(T[e]);
    if(T[2*e+2]!=Nil) /* 右子树不空 */
        InTraverse(T,2*e+2);
}

/* 初始条件: 二叉树存在 */
```

```
/* 操作结果: 中序遍历 T。 */
Status InOrderTraverse(SqBiTree T)
{
    if(!BiTreeEmpty(T)) /* 树不空 */
        InTraverse(T,0);
    printf("\n");
    return OK;
}

/* PostOrderTraverse()调用 */
void PostTraverse(SqBiTree T,int e)
{
    if(T[2*e+1]!=Nil) /* 左子树不空 */
        PostTraverse(T,2*e+1);
    if(T[2*e+2]!=Nil) /* 右子树不空 */
        PostTraverse(T,2*e+2);
    visit(T[e]);
}

/* 初始条件: 二叉树 T 存在 */
/* 操作结果: 后序遍历 T。 */
Status PostOrderTraverse(SqBiTree T)
{
    if(!BiTreeEmpty(T)) /* 树不空 */
        PostTraverse(T,0);
    printf("\n");
    return OK;
}

/* 层序遍历二叉树 */
void LevelOrderTraverse(SqBiTree T)
{
    int i=MAX_TREE_SIZE-1,j;
    while(T[i]==Nil)
        i--; /* 找到最后一个非空结点的序号 */
    for(j=0;j<=i;j++)  /* 从根结点起,按层序遍历二叉树 */
        if(T[j]!=Nil)
            visit(T[j]); /* 只遍历非空的结点 */
    printf("\n");
}

/* 逐层、按本层序号输出二叉树 */
void Print(SqBiTree T)
{
```

```
    int j,k;
    Position p;
    TElemType e;
    for(j=1;j<=BiTreeDepth(T);j++)
    {
        printf("第%d 层: ",j);
        for(k=1;k<=powl(2,j-1);k++)
        {
            p.level=j;
            p.order=k;
            e=Value(T,p);
            if(e!=Nil)
                printf("%d:%d ",k,e);
        }
        printf("\n");
    }
}


int main()
{
    Status i;
    Position p;
    TElemType e;
    SqBiTree T;
    InitBiTree(T);
    CreateBiTree(T);
    printf(" 建 立 二 叉 树 后 , 树 空 否 ？ %d(1: 是  0: 否 ) 树 的 深 度
=%d\n",BiTreeEmpty(T),BiTreeDepth(T));
    i=Root(T,&e);
    if(i)
        printf("二叉树的根为：%d\n",e);
    else
        printf("树空，无根\n");
    printf("层序遍历二叉树:\n");
    LevelOrderTraverse(T);
    printf("前序遍历二叉树:\n");
    PreOrderTraverse(T);
    printf("中序遍历二叉树:\n");
    InOrderTraverse(T);
    printf("后序遍历二叉树:\n");
    PostOrderTraverse(T);
    printf("修改结点的层号 3 本层序号 2。");
    p.level=3;
```

```
    p.order=2;
    e=Value(T,p);
    printf("待修改结点的原值为%d 请输入新值:50 ",e);
    e=50;
    Assign(T,p,e);
    printf("前序遍历二叉树:\n");
    PreOrderTraverse(T);
    printf("结点%d 的双亲为%d,左右孩子分别为",e,Parent(T,e));
    printf("%d,%d,左右兄弟分别为",LeftChild(T,e),RightChild(T,e));
    printf("%d,%d\n",LeftSibling(T,e),RightSibling(T,e));
    ClearBiTree(T);
    printf(" 清 除 二 叉 树 后 , 树 空 否 ? %d(1: 是  0: 否 ) 树 的 深 度
=%d\n",BiTreeEmpty(T),BiTreeDepth(T));
    i=Root(T,&e);
    if(i)
        printf("二叉树的根为：%d\n",e);
    else
        printf("树空，无根\n");

    return 0;
}
```

02 二叉树链式结构实现_BiTreeLink

```
#include "string.h"
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 100 /* 存储空间初始分配量 */

typedef int Status;        /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */

/* 用于构造二叉树********************************** */
int index=1;
typedef char String[24]; /*  0 号单元存放串的长度 */
String str;
```

```
Status StrAssign(String T,char *chars)
{
	int i;
	if(strlen(chars)>MAXSIZE)
		return ERROR;
	else
	{
		T[0]=strlen(chars);
		for(i=1;i<=T[0];i++)
			T[i]=*(chars+i-1);
		return OK;
	}
}
/* ********************************************** */

typedef char TElemType;
TElemType Nil=' '; /*  字符型以空格符为空  */

Status visit(TElemType e)
{
	printf("%c ",e);
	return OK;
}

typedef struct BiTNode    /*  结点结构  */
{
    TElemType data;            /*  结点数据  */
    struct BiTNode *lchild,*rchild; /*  左右孩子指针  */
}BiTNode,*BiTree;


/*  构造空二叉树 T */
Status InitBiTree(BiTree *T)
{
	*T=NULL;
	return OK;
}

/*  初始条件: 二叉树 T 存在。操作结果: 销毁二叉树 T */
void DestroyBiTree(BiTree *T)
{
	if(*T)
	{
```

```
        if((*T)->lchild) /* 有左孩子 */
            DestroyBiTree(&(*T)->lchild); /* 销毁左孩子子树 */
        if((*T)->rchild) /* 有右孩子 */
            DestroyBiTree(&(*T)->rchild); /* 销毁右孩子子树 */
        free(*T); /* 释放根结点 */
        *T=NULL; /* 空指针赋 0 */
    }
}

/* 按前序输入二叉树中结点的值（一个字符） */
/* #表示空树，构造二叉链表表示二叉树 T。 */
void CreateBiTree(BiTree *T)
{
    TElemType ch;

    /* scanf("%c",&ch); */
    ch=str[index++];

    if(ch=='#')
        *T=NULL;
    else
    {
        *T=(BiTree)malloc(sizeof(BiTNode));
        if(!*T)
            exit(OVERFLOW);
        (*T)->data=ch; /* 生成根结点 */
        CreateBiTree(&(*T)->lchild); /* 构造左子树 */
        CreateBiTree(&(*T)->rchild); /* 构造右子树 */
    }
 }

/* 初始条件: 二叉树 T 存在 */
/* 操作结果: 若 T 为空二叉树,则返回 TRUE,否则 FALSE */
Status BiTreeEmpty(BiTree T)
{
    if(T)
        return FALSE;
    else
        return TRUE;
}

#define ClearBiTree DestroyBiTree

/* 初始条件: 二叉树 T 存在。操作结果: 返回 T 的深度 */
```

```
int BiTreeDepth(BiTree T)
{
	int i,j;
	if(!T)
		return 0;
	if(T->lchild)
		i=BiTreeDepth(T->lchild);
	else
		i=0;
	if(T->rchild)
		j=BiTreeDepth(T->rchild);
	else
		j=0;
	return i>j?i+1:j+1;
}

/* 初始条件: 二叉树 T 存在。操作结果: 返回 T 的根 */
TElemType Root(BiTree T)
{
	if(BiTreeEmpty(T))
		return Nil;
	else
		return T->data;
}

/* 初始条件: 二叉树 T 存在，p 指向 T 中某个结点 */
/* 操作结果: 返回 p 所指结点的值 */
TElemType Value(BiTree p)
{
	return p->data;
}

/* 给 p 所指结点赋值为 value */
void Assign(BiTree p,TElemType value)
{
	p->data=value;
}

/* 初始条件: 二叉树 T 存在 */
/* 操作结果: 前序递归遍历 T */
void PreOrderTraverse(BiTree T)
{
	if(T==NULL)
		return;
```

```
    printf("%c",T->data);/* 显示结点数据，可以更改为其它对结点操作 */
    PreOrderTraverse(T->lchild); /* 再先序遍历左子树 */
    PreOrderTraverse(T->rchild); /* 最后先序遍历右子树 */
}

/* 初始条件: 二叉树 T 存在 */
/* 操作结果: 中序递归遍历 T */
void InOrderTraverse(BiTree T)
{
    if(T==NULL)
        return;
    InOrderTraverse(T->lchild); /* 中序遍历左子树 */
    printf("%c",T->data);/* 显示结点数据，可以更改为其它对结点操作 */
    InOrderTraverse(T->rchild); /* 最后中序遍历右子树 */
}

/* 初始条件: 二叉树 T 存在 */
/* 操作结果: 后序递归遍历 T */
void PostOrderTraverse(BiTree T)
{
    if(T==NULL)
        return;
    PostOrderTraverse(T->lchild); /* 先后序遍历左子树  */
    PostOrderTraverse(T->rchild); /* 再后序遍历右子树  */
    printf("%c",T->data);/* 显示结点数据，可以更改为其它对结点操作 */
}


int main()
{
    int i;
    BiTree T;
    TElemType e1;
    InitBiTree(&T);


    StrAssign(str,"ABDH#K###E##CFI###G#J##");

    CreateBiTree(&T);

    printf("构造空二叉树后，树空否？ %d(1: 是  0: 否) 树的深度
=%d\n",BiTreeEmpty(T),BiTreeDepth(T));
    e1=Root(T);
    printf("二叉树的根为: %c\n",e1);
```

```
    printf("\n 前序遍历二叉树:");
    PreOrderTraverse(T);
    printf("\n 中序遍历二叉树:");
    InOrderTraverse(T);
    printf("\n 后序遍历二叉树:");
    PostOrderTraverse(T);
    ClearBiTree(&T);
    printf("\n 清除二叉树后,树空否? %d(1: 是  0: 否) 树的深度
=%d\n",BiTreeEmpty(T),BiTreeDepth(T));
    i=Root(T);
    if(!i)
        printf("树空，无根\n");

    return 0;
}
```

03 线索二叉树_ThreadBinaryTree

```
#include "string.h"
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 100 /* 存储空间初始分配量 */

typedef int Status;  /* Status 是函数的类型,其值是函数结果状态代码,如 OK 等 */
typedef char TElemType;
typedef enum {Link,Thread} PointerTag;       /* Link==0 表示指向左右孩子指针, */
                                             /* Thread==1 表示指向前驱或后继的线索 */
typedef   struct BiThrNode   /* 二叉线索存储结点结构 */
{
    TElemType data;    /* 结点数据 */
    struct BiThrNode *lchild, *rchild;  /* 左右孩子指针 */
    PointerTag LTag;
    PointerTag RTag;           /* 左右标志 */
} BiThrNode, *BiThrTree;
```

```
TElemType Nil='#'; /* 字符型以空格符为空 */

Status visit(TElemType e)
{
    printf("%c ",e);
    return OK;
}

/* 按前序输入二叉线索树中结点的值,构造二叉线索树 T */
/* 0(整型)/空格(字符型)表示空结点 */
Status CreateBiThrTree(BiThrTree *T)
{
    TElemType h;
    scanf("%c",&h);

    if(h==Nil)
        *T=NULL;
    else
    {
        *T=(BiThrTree)malloc(sizeof(BiThrNode));
        if(!*T)
            exit(OVERFLOW);
        (*T)->data=h; /* 生成根结点(前序) */
        CreateBiThrTree(&(*T)->lchild); /* 递归构造左子树 */
        if((*T)->lchild) /* 有左孩子 */
            (*T)->LTag=Link;
        CreateBiThrTree(&(*T)->rchild); /* 递归构造右子树 */
        if((*T)->rchild) /* 有右孩子 */
            (*T)->RTag=Link;
    }
    return OK;
}

BiThrTree pre; /* 全局变量,始终指向刚刚访问过的结点 */
/* 中序遍历进行中序线索化 */
void InThreading(BiThrTree p)
{
    if(p)
    {
        InThreading(p->lchild); /* 递归左子树线索化 */
        if(!p->lchild) /* 没有左孩子 */
        {
            p->LTag=Thread; /* 前驱线索 */
```

```
            p->lchild=pre; /* 左孩子指针指向前驱 */
        }
        if(!pre->rchild) /* 前驱没有右孩子 */
        {
            pre->RTag=Thread; /* 后继线索 */
            pre->rchild=p; /* 前驱右孩子指针指向后继(当前结点 p) */
        }
        pre=p; /* 保持 pre 指向 p 的前驱 */
        InThreading(p->rchild); /* 递归右子树线索化 */
    }
}

/* 中序遍历二叉树 T,并将其中序线索化,Thrt 指向头结点 */
Status InOrderThreading(BiThrTree *Thrt,BiThrTree T)
{
    *Thrt=(BiThrTree)malloc(sizeof(BiThrNode));
    if(!*Thrt)
        exit(OVERFLOW);
    (*Thrt)->LTag=Link; /* 建头结点 */
    (*Thrt)->RTag=Thread;
    (*Thrt)->rchild=(*Thrt); /* 右指针回指 */
    if(!T) /* 若二叉树空,则左指针回指 */
        (*Thrt)->lchild=*Thrt;
    else
    {
        (*Thrt)->lchild=T;
        pre=(*Thrt);
        InThreading(T); /* 中序遍历进行中序线索化 */
        pre->rchild=*Thrt;
        pre->RTag=Thread; /* 最后一个结点线索化 */
        (*Thrt)->rchild=pre;
    }
    return OK;
}

/* 中序遍历二叉线索树 T(头结点)的非递归算法 */
Status InOrderTraverse_Thr(BiThrTree T)
{
    BiThrTree p;
    p=T->lchild; /* p 指向根结点 */
    while(p!=T)
    { /* 空树或遍历结束时,p==T */
        while(p->LTag==Link)
            p=p->lchild;
```

```
            if(!visit(p->data)) /* 访问其左子树为空的结点 */
                return ERROR;
            while(p->RTag==Thread&&p->rchild!=T)
            {
                p=p->rchild;
                visit(p->data); /* 访问后继结点 */
            }
            p=p->rchild;
        }
        return OK;
}

int main()
{
        BiThrTree H,T;
        printf("请按前序输入二叉树(如:'ABDH##I##EJ###CF##G##')\n");
        CreateBiThrTree(&T); /* 按前序产生二叉树 */
        InOrderThreading(&H,T); /* 中序遍历,并中序线索化二叉树 */
        printf("中序遍历(输出)二叉线索树:\n");
        InOrderTraverse_Thr(H); /* 中序遍历(输出)二叉线索树 */
        printf("\n");

        return 0;
}
```

第七章 图

01 邻接矩阵创建_CreateMGraph

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define MAXVEX 100 /* 最大顶点数，应由用户定义 */
#define INFINITY 65535
```

```
typedef int Status;  /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */
typedef char VertexType; /* 顶点类型应由用户定义 */
typedef int EdgeType; /* 边上的权值类型应由用户定义 */
typedef struct
{
    VertexType vexs[MAXVEX]; /* 顶点表 */
    EdgeType arc[MAXVEX][MAXVEX];/* 邻接矩阵，可看作边表 */
    int numNodes, numEdges; /* 图中当前的顶点数和边数 */
}MGraph;

/* 建立无向网图的邻接矩阵表示 */
void CreateMGraph(MGraph *G)
{
    int i,j,k,w;
    printf("输入顶点数和边数:\n");
    scanf("%d,%d",&G->numNodes,&G->numEdges); /* 输入顶点数和边数 */
    for(i = 0;i <G->numNodes;i++) /* 读入顶点信息,建立顶点表 */
        scanf(&G->vexs[i]);
    for(i = 0;i <G->numNodes;i++)
        for(j = 0;j <G->numNodes;j++)
            G->arc[i][j]=INFINITY;    /* 邻接矩阵初始化 */
    for(k = 0;k <G->numEdges;k++) /* 读入 numEdges 条边，建立邻接矩阵 */
    {
        printf("输入边(vi,vj)上的下标 i，下标 j 和权 w:\n");
        scanf("%d,%d,%d",&i,&j,&w); /* 输入边(vi,vj)上的权 w */
        G->arc[i][j]=w;
        G->arc[j][i]= G->arc[i][j]; /* 因为是无向图，矩阵对称 */
    }
}

int main(void)
{
    MGraph G;
    CreateMGraph(&G);

    return 0;
}
```

02 邻接表创建_CreateALGraph

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
```

```
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define MAXVEX 100 /* 最大顶点数,应由用户定义 */

typedef int Status;  /* Status 是函数的类型,其值是函数结果状态代码,如 OK 等 */
typedef char VertexType; /* 顶点类型应由用户定义 */
typedef int EdgeType; /* 边上的权值类型应由用户定义 */

typedef struct EdgeNode /* 边表结点  */
{
    int adjvex;      /* 邻接点域,存储该顶点对应的下标 */
    EdgeType info;      /* 用于存储权值,对于非网图可以不需要 */
    struct EdgeNode *next; /* 链域,指向下一个邻接点 */
}EdgeNode;

typedef struct VertexNode /* 顶点表结点 */
{
    VertexType data; /* 顶点域,存储顶点信息 */
    EdgeNode *firstedge;/* 边表头指针 */
}VertexNode, AdjList[MAXVEX];

typedef struct
{
    AdjList adjList;
    int numNodes,numEdges; /* 图中当前顶点数和边数 */
}GraphAdjList;

/* 建立图的邻接表结构 */
void   CreateALGraph(GraphAdjList *G)
{
    int i,j,k;
    EdgeNode *e;
    printf("输入顶点数和边数:\n");
    scanf("%d,%d",&G->numNodes,&G->numEdges); /* 输入顶点数和边数 */
    for(i = 0;i < G->numNodes;i++) /* 读入顶点信息,建立顶点表 */
    {
        scanf(&G->adjList[i].data);    /* 输入顶点信息 */
        G->adjList[i].firstedge=NULL;       /* 将边表置为空表 */
    }
```

```
    for(k = 0;k < G->numEdges;k++)/* 建立边表 */
    {
        printf("输入边(vi,vj)上的顶点序号:\n");
        scanf("%d,%d",&i,&j); /* 输入边(vi,vj)上的顶点序号 */
        e=(EdgeNode *)malloc(sizeof(EdgeNode)); /* 向内存申请空间,生成边表结点 */
        e->adjvex=j;                    /* 邻接序号为 j */
        e->next=G->adjList[i].firstedge;    /* 将 e 的指针指向当前顶点上指向的结点 */
        G->adjList[i].firstedge=e;      /* 将当前顶点的指针指向 e */

        e=(EdgeNode *)malloc(sizeof(EdgeNode)); /* 向内存申请空间,生成边表结点 */
        e->adjvex=i;                    /* 邻接序号为 i */
        e->next=G->adjList[j].firstedge;    /* 将 e 的指针指向当前顶点上指向的结点 */
        G->adjList[j].firstedge=e;      /* 将当前顶点的指针指向 e */
    }
}

int main(void)
{
    GraphAdjList G;
    CreateALGraph(&G);

    return 0;
}
```

03 邻接矩阵深度和广度遍历 DFS_BFS

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

typedef int Status;  /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */
typedef int Boolean; /* Boolean 是布尔类型,其值是 TRUE 或 FALSE */

typedef char VertexType; /* 顶点类型应由用户定义 */
typedef int EdgeType; /* 边上的权值类型应由用户定义 */
```

```
#define MAXSIZE 9 /* 存储空间初始分配量 */
#define MAXEDGE 15
#define MAXVEX 9
#define INFINITY 65535

typedef struct
{
    VertexType vexs[MAXVEX]; /* 顶点表 */
    EdgeType arc[MAXVEX][MAXVEX];/* 邻接矩阵，可看作边表 */
    int numVertexes, numEdges; /* 图中当前的顶点数和边数 */
}MGraph;

/* 用到的队列结构与函数********************************** */

/* 循环队列的顺序存储结构 */
typedef struct
{
    int data[MAXSIZE];
    int front;      /* 头指针 */
    int rear;       /* 尾指针，若队列不空，指向队列尾元素的下一个位置 */
}Queue;

/* 初始化一个空队列 Q */
Status InitQueue(Queue *Q)
{
    Q->front=0;
    Q->rear=0;
    return  OK;
}

/* 若队列 Q 为空队列,则返回 TRUE,否则返回 FALSE */
Status QueueEmpty(Queue Q)
{
    if(Q.front==Q.rear) /* 队列空的标志 */
        return TRUE;
    else
        return FALSE;
}

/* 若队列未满，则插入元素 e 为 Q 新的队尾元素 */
Status EnQueue(Queue *Q,int e)
{
    if ((Q->rear+1)%MAXSIZE == Q->front)  /* 队列满的判断 */
```

```
        return ERROR;
    Q->data[Q->rear]=e;                 /* 将元素 e 赋值给队尾 */
    Q->rear=(Q->rear+1)%MAXSIZE;/* rear 指针向后移一位置， */
                                        /* 若到最后则转到数组头部 */
    return  OK;
}

/* 若队列不空，则删除 Q 中队头元素，用 e 返回其值 */
Status DeQueue(Queue *Q,int *e)
{
    if (Q->front == Q->rear)            /* 队列空的判断 */
        return ERROR;
    *e=Q->data[Q->front];                   /* 将队头元素赋值给 e */
    Q->front=(Q->front+1)%MAXSIZE; /* front 指针向后移一位置， */
                                            /* 若到最后则转到数组头部 */
    return  OK;
}
/* ********************************************************** */


void CreateMGraph(MGraph *G)
{
    int i, j;

    G->numEdges=15;
    G->numVertexes=9;

    /* 读入顶点信息，建立顶点表 */
    G->vexs[0]='A';
    G->vexs[1]='B';
    G->vexs[2]='C';
    G->vexs[3]='D';
    G->vexs[4]='E';
    G->vexs[5]='F';
    G->vexs[6]='G';
    G->vexs[7]='H';
    G->vexs[8]='I';


    for (i = 0; i < G->numVertexes; i++)/* 初始化图 */
    {
        for ( j = 0; j < G->numVertexes; j++)
        {
            G->arc[i][j]=0;
```

```
            }
        }

        G->arc[0][1]=1;
        G->arc[0][5]=1;

        G->arc[1][2]=1;
        G->arc[1][8]=1;
        G->arc[1][6]=1;

        G->arc[2][3]=1;
        G->arc[2][8]=1;

        G->arc[3][4]=1;
        G->arc[3][7]=1;
        G->arc[3][6]=1;
        G->arc[3][8]=1;

        G->arc[4][5]=1;
        G->arc[4][7]=1;

        G->arc[5][6]=1;

        G->arc[6][7]=1;


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

}

Boolean visited[MAXVEX]; /* 访问标志的数组 */

/* 邻接矩阵的深度优先递归算法 */
void DFS(MGraph G, int i)
{
    int j;
    visited[i] = TRUE;
    printf("%c ", G.vexs[i]);/* 打印顶点，也可以其它操作 */
```

```
    for(j = 0; j < G.numVertexes; j++)
        if(G.arc[i][j] == 1 && !visited[j])
            DFS(G, j);/* 对为访问的邻接顶点递归调用 */
}

/* 邻接矩阵的深度遍历操作 */
void DFSTraverse(MGraph G)
{
    int i;
    for(i = 0; i < G.numVertexes; i++)
        visited[i] = FALSE; /* 初始所有顶点状态都是未访问过状态 */
    for(i = 0; i < G.numVertexes; i++)
        if(!visited[i]) /* 对未访问过的顶点调用 DFS，若是连通图，只会执行一次 */
            DFS(G, i);
}

/* 邻接矩阵的广度遍历算法 */
void BFSTraverse(MGraph G)
{
    int i, j;
    Queue Q;
    for(i = 0; i < G.numVertexes; i++)
        visited[i] = FALSE;
    InitQueue(&Q);          /* 初始化一辅助用的队列 */
    for(i = 0; i < G.numVertexes; i++)   /* 对每一个顶点做循环 */
    {
        if (!visited[i])  /* 若是未访问过就处理 */
        {
            visited[i]=TRUE;          /* 设置当前顶点访问过 */
            printf("%c ", G.vexs[i]);/* 打印顶点，也可以其它操作 */
            EnQueue(&Q,i);          /* 将此顶点入队列 */
            while(!QueueEmpty(Q)) /* 若当前队列不为空 */
            {
                DeQueue(&Q,&i);  /* 将队对元素出队列，赋值给 i */
                for(j=0;j<G.numVertexes;j++)
                {
                    /* 判断其它顶点若与当前顶点存在边且未访问过   */
                    if(G.arc[i][j] == 1 && !visited[j])
                    {
                        visited[j]=TRUE;              /* 将找到的此顶点标记为已访问
*/
                        printf("%c ", G.vexs[j]);  /* 打印顶点 */
                        EnQueue(&Q,j);                    /* 将找到的此顶点入队列   */
                    }
```

```
                    }
                }
            }
        }
}


int main(void)
{
    MGraph G;
    CreateMGraph(&G);
    printf("\n 深度遍历： ");
    DFSTraverse(G);
    printf("\n 广度遍历： ");
    BFSTraverse(G);
    return 0;
}
```

04 邻接表深度和广度遍历 DFS_BFS

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 9 /* 存储空间初始分配量 */
#define MAXEDGE 15
#define MAXVEX 9
#define INFINITY 65535

typedef int Status;  /* Status 是函数的类型,其值是函数结果状态代码,如 OK 等 */
typedef int Boolean; /* Boolean 是布尔类型,其值是 TRUE 或 FALSE */

typedef char VertexType; /* 顶点类型应由用户定义 */
typedef int EdgeType; /* 边上的权值类型应由用户定义 */

/* 邻接矩阵结构 */
typedef struct
```

```
{
    VertexType vexs[MAXVEX]; /* 顶点表 */
    EdgeType arc[MAXVEX][MAXVEX];/* 邻接矩阵,可看作边表 */
    int numVertexes, numEdges; /* 图中当前的顶点数和边数 */
}MGraph;

/* 邻接表结构***************** */
typedef struct EdgeNode /* 边表结点 */
{
    int adjvex;      /* 邻接点域,存储该顶点对应的下标 */
    int weight;          /* 用于存储权值,对于非网图可以不需要 */
    struct EdgeNode *next; /* 链域,指向下一个邻接点 */
}EdgeNode;

typedef struct VertexNode /* 顶点表结点 */
{
    int in;     /* 顶点入度 */
    char data; /* 顶点域,存储顶点信息 */
    EdgeNode *firstedge;/* 边表头指针 */
}VertexNode, AdjList[MAXVEX];

typedef struct
{
    AdjList adjList;
    int numVertexes,numEdges; /* 图中当前顶点数和边数 */
}graphAdjList,*GraphAdjList;
/* ************************** */

/* 用到的队列结构与函数********************************** */
/* 循环队列的顺序存储结构 */
typedef struct
{
    int data[MAXSIZE];
    int front;       /* 头指针 */
    int rear;        /* 尾指针,若队列不空,指向队列尾元素的下一个位置 */
}Queue;

/* 初始化一个空队列 Q */
Status InitQueue(Queue *Q)
{
    Q->front=0;
    Q->rear=0;
    return   OK;
}
```

```
/*  若队列 Q 为空队列,则返回 TRUE,否则返回 FALSE */
Status QueueEmpty(Queue Q)
{
    if(Q.front==Q.rear) /*  队列空的标志  */
        return TRUE;
    else
        return FALSE;
}

/*  若队列未满,则插入元素 e 为 Q 新的队尾元素  */
Status EnQueue(Queue *Q,int e)
{
    if ((Q->rear+1)%MAXSIZE == Q->front)  /*  队列满的判断  */
        return ERROR;
    Q->data[Q->rear]=e;                   /*  将元素 e 赋值给队尾  */
    Q->rear=(Q->rear+1)%MAXSIZE;/* rear 指针向后移一位置, */
                                          /*  若到最后则转到数组头部  */
    return    OK;
}

/*  若队列不空,则删除 Q 中队头元素,用 e 返回其值  */
Status DeQueue(Queue *Q,int *e)
{
    if (Q->front == Q->rear)              /*  队列空的判断  */
        return ERROR;
    *e=Q->data[Q->front];                     /*  将队头元素赋值给 e */
    Q->front=(Q->front+1)%MAXSIZE; /* front 指针向后移一位置, */
                                          /*  若到最后则转到数组头部  */
    return    OK;
}
/* ********************************************************** */



void CreateMGraph(MGraph *G)
{
    int i, j;

    G->numEdges=15;
    G->numVertexes=9;

    /*  读入顶点信息,建立顶点表  */
    G->vexs[0]='A';
```

```
G->vexs[1]='B';
G->vexs[2]='C';
G->vexs[3]='D';
G->vexs[4]='E';
G->vexs[5]='F';
G->vexs[6]='G';
G->vexs[7]='H';
G->vexs[8]='I';


for (i = 0; i < G->numVertexes; i++)/* 初始化图 */
{
    for ( j = 0; j < G->numVertexes; j++)
    {
        G->arc[i][j]=0;
    }
}

G->arc[0][1]=1;
G->arc[0][5]=1;

G->arc[1][2]=1;
G->arc[1][8]=1;
G->arc[1][6]=1;

G->arc[2][3]=1;
G->arc[2][8]=1;

G->arc[3][4]=1;
G->arc[3][7]=1;
G->arc[3][6]=1;
G->arc[3][8]=1;

G->arc[4][5]=1;
G->arc[4][7]=1;

G->arc[5][6]=1;

G->arc[6][7]=1;


for(i = 0; i < G->numVertexes; i++)
{
    for(j = i; j < G->numVertexes; j++)
```

```
        {
            G->arc[j][i] =G->arc[i][j];
        }
    }

}

/* 利用邻接矩阵构建邻接表 */
void CreateALGraph(MGraph G,GraphAdjList *GL)
{
    int i,j;
    EdgeNode *e;

    *GL = (GraphAdjList)malloc(sizeof(graphAdjList));

    (*GL)->numVertexes=G.numVertexes;
    (*GL)->numEdges=G.numEdges;
    for(i= 0;i <G.numVertexes;i++) /* 读入顶点信息,建立顶点表 */
    {
        (*GL)->adjList[i].in=0;
        (*GL)->adjList[i].data=G.vexs[i];
        (*GL)->adjList[i].firstedge=NULL;  /* 将边表置为空表 */
    }

    for(i=0;i<G.numVertexes;i++) /* 建立边表 */
    {
        for(j=0;j<G.numVertexes;j++)
        {
            if (G.arc[i][j]==1)
            {
                e=(EdgeNode *)malloc(sizeof(EdgeNode));
                e->adjvex=j;                    /* 邻接序号为 j */
                e->next=(*GL)->adjList[i].firstedge;    /* 将当前顶点上的指向的结点指针
赋值给 e */
                (*GL)->adjList[i].firstedge=e;      /* 将当前顶点的指针指向 e */
                (*GL)->adjList[j].in++;

            }
        }
    }

}

Boolean visited[MAXSIZE]; /* 访问标志的数组 */
```

```
/* 邻接表的深度优先递归算法 */
void DFS(GraphAdjList GL, int i)
{
    EdgeNode *p;
    visited[i] = TRUE;
    printf("%c ",GL->adjList[i].data);/* 打印顶点,也可以其它操作 */
    p = GL->adjList[i].firstedge;
    while(p)
    {
        if(!visited[p->adjvex])
            DFS(GL, p->adjvex);/* 对为访问的邻接顶点递归调用 */
        p = p->next;
    }
}

/* 邻接表的深度遍历操作 */
void DFSTraverse(GraphAdjList GL)
{
    int i;
    for(i = 0; i < GL->numVertexes; i++)
        visited[i] = FALSE; /* 初始所有顶点状态都是未访问过状态 */
    for(i = 0; i < GL->numVertexes; i++)
        if(!visited[i]) /* 对未访问过的顶点调用 DFS,若是连通图,只会执行一次 */
            DFS(GL, i);
}

/* 邻接表的广度遍历算法 */
void BFSTraverse(GraphAdjList GL)
{
    int i;
    EdgeNode *p;
    Queue Q;
    for(i = 0; i < GL->numVertexes; i++)
        visited[i] = FALSE;
    InitQueue(&Q);
    for(i = 0; i < GL->numVertexes; i++)
    {
        if (!visited[i])
        {
            visited[i]=TRUE;
            printf("%c ",GL->adjList[i].data);/* 打印顶点,也可以其它操作 */
            EnQueue(&Q,i);
            while(!QueueEmpty(Q))
```

```
            {
                DeQueue(&Q,&i);
                p = GL->adjList[i].firstedge;    /* 找到当前顶点的边表链表头指针 */
                while(p)
                {
                    if(!visited[p->adjvex])    /* 若此顶点未被访问 */
                    {
                        visited[p->adjvex]=TRUE;
                        printf("%c ",GL->adjList[p->adjvex].data);
                        EnQueue(&Q,p->adjvex);      /* 将此顶点入队列 */
                    }
                    p = p->next;    /* 指针指向下一个邻接点 */
                }
            }
        }
    }
}

int main(void)
{
    MGraph G;
    GraphAdjList GL;
    CreateMGraph(&G);
    CreateALGraph(G,&GL);

    printf("\n 深度遍历:");
    DFSTraverse(GL);
    printf("\n 广度遍历:");
    BFSTraverse(GL);
    return 0;
}
```

05 最小生成树_Prim

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
```

```
#define MAXEDGE 20
#define MAXVEX 20
#define INFINITY 65535

typedef int Status;  /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等  */

typedef struct
{
    int arc[MAXVEX][MAXVEX];
    int numVertexes, numEdges;
}MGraph;

void CreateMGraph(MGraph *G)/*  构件图  */
{
    int i, j;

    /* printf("请输入边数和顶点数:"); */
    G->numEdges=15;
    G->numVertexes=9;

    for (i = 0; i < G->numVertexes; i++)/*  初始化图  */
    {
        for ( j = 0; j < G->numVertexes; j++)
        {
            if (i==j)
                G->arc[i][j]=0;
            else
                G->arc[i][j] = G->arc[j][i] = INFINITY;
        }
    }

    G->arc[0][1]=10;
    G->arc[0][5]=11;
    G->arc[1][2]=18;
    G->arc[1][8]=12;
    G->arc[1][6]=16;
    G->arc[2][8]=8;
    G->arc[2][3]=22;
    G->arc[3][8]=21;
    G->arc[3][6]=24;
    G->arc[3][7]=16;
    G->arc[3][4]=20;
    G->arc[4][7]=7;
```

```
    G->arc[4][5]=26;
    G->arc[5][6]=17;
    G->arc[6][7]=19;

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

}

/* Prim 算法生成最小生成树  */
void MiniSpanTree_Prim(MGraph G)
{
    int min, i, j, k;
    int adjvex[MAXVEX];          /* 保存相关顶点下标 */
    int lowcost[MAXVEX];   /* 保存相关顶点间边的权值 */
    lowcost[0] = 0;/* 初始化第一个权值为 0，即 v0 加入生成树 */
            /* lowcost 的值为 0，在这里就是此下标的顶点已经加入生成树 */
    adjvex[0] = 0;           /* 初始化第一个顶点下标为 0 */
    for(i = 1; i < G.numVertexes; i++)   /* 循环除下标为 0 外的全部顶点 */
    {
        lowcost[i] = G.arc[0][i];  /* 将 v0 顶点与之有边的权值存入数组 */
        adjvex[i] = 0;                    /* 初始化都为 v0 的下标 */
    }
    for(i = 1; i < G.numVertexes; i++)
    {
        min = INFINITY;      /* 初始化最小权值为∞， */
                             /* 通常设置为不可能的大数字如 32767、65535 等 */
        j = 1;k = 0;
        while(j < G.numVertexes)      /* 循环全部顶点 */
        {
            if(lowcost[j]!=0 && lowcost[j] < min)/* 如果权值不为 0 且权值小于 min */
            {
                min = lowcost[j];    /* 则让当前权值成为最小值 */
                k = j;               /* 将当前最小值的下标存入 k */
            }
            j++;
        }
        printf("(%d, %d)\n", adjvex[k], k);/* 打印当前顶点边中权值最小的边 */
        lowcost[k] = 0;/* 将当前顶点的权值设置为 0,表示此顶点已经完成任务 */
```

```
            for(j = 1; j < G.numVertexes; j++)    /* 循环所有顶点 */
            {
                if(lowcost[j]!=0 && G.arc[k][j] < lowcost[j])
                {/* 如果下标为 k 顶点各边权值小于此前这些顶点未被加入生成树权值 */
                    lowcost[j] = G.arc[k][j];/* 将较小的权值存入 lowcost 相应位置 */
                    adjvex[j] = k;                    /* 将下标为 k 的顶点存入 adjvex */
                }
            }
        }
}

int main(void)
{
    MGraph G;
    CreateMGraph(&G);
    MiniSpanTree_Prim(G);

    return 0;

}
```

06 最小生成树_Kruskal

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

typedef int Status;  /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */

#define MAXEDGE 20
#define MAXVEX 20
#define INFINITY 65535

typedef struct
{
    int arc[MAXVEX][MAXVEX];
    int numVertexes, numEdges;
```

```
}MGraph;

typedef struct
{
    int begin;
    int end;
    int weight;
}Edge;    /* 对边集数组 Edge 结构的定义 */

/* 构件图 */
void CreateMGraph(MGraph *G)
{
    int i, j;

    /* printf("请输入边数和顶点数:"); */
    G->numEdges=15;
    G->numVertexes=9;

    for (i = 0; i < G->numVertexes; i++)/* 初始化图 */
    {
        for ( j = 0; j < G->numVertexes; j++)
        {
            if (i==j)
                G->arc[i][j]=0;
            else
                G->arc[i][j] = G->arc[j][i] = INFINITY;
        }
    }

    G->arc[0][1]=10;
    G->arc[0][5]=11;
    G->arc[1][2]=18;
    G->arc[1][8]=12;
    G->arc[1][6]=16;
    G->arc[2][8]=8;
    G->arc[2][3]=22;
    G->arc[3][8]=21;
    G->arc[3][6]=24;
    G->arc[3][7]=16;
    G->arc[3][4]=20;
    G->arc[4][7]=7;
    G->arc[4][5]=26;
    G->arc[5][6]=17;
    G->arc[6][7]=19;
```

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

}

/* 交换权值 以及头和尾 */
void Swapn(Edge *edges,int i, int j)
{
    int temp;
    temp = edges[i].begin;
    edges[i].begin = edges[j].begin;
    edges[j].begin = temp;
    temp = edges[i].end;
    edges[i].end = edges[j].end;
    edges[j].end = temp;
    temp = edges[i].weight;
    edges[i].weight = edges[j].weight;
    edges[j].weight = temp;
}

/* 对权值进行排序 */
void sort(Edge edges[],MGraph *G)
{
    int i, j;
    for ( i = 0; i < G->numEdges; i++)
    {
        for ( j = i + 1; j < G->numEdges; j++)
        {
            if (edges[i].weight > edges[j].weight)
            {
                Swapn(edges, i, j);
            }
        }
    }
    printf("权排序之后的为:\n");
    for (i = 0; i < G->numEdges; i++)
    {
        printf("(%d, %d) %d\n", edges[i].begin, edges[i].end, edges[i].weight);
```

```
    }

}

/* 查找连线顶点的尾部下标 */
int Find(int *parent, int f)
{
    while ( parent[f] > 0)
    {
        f = parent[f];
    }
    return f;
}

/* 生成最小生成树 */
void MiniSpanTree_Kruskal(MGraph G)
{
    int i, j, n, m;
    int k = 0;
    int parent[MAXVEX];/* 定义一数组用来判断边与边是否形成环路 */

    Edge edges[MAXEDGE];/* 定义边集数组,edge 的结构为 begin,end,weight,均为整型 */

    /* 用来构建边集数组并排序********************** */
    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;
                edges[k].end = j;
                edges[k].weight = G.arc[i][j];
                k++;
            }
        }
    }
    sort(edges, &G);
    /* ***************************************** */


    for (i = 0; i < G.numVertexes; i++)
        parent[i] = 0;   /* 初始化数组值为 0 */
```

```
    printf("打印最小生成树：\n");
    for (i = 0; i < G.numEdges; i++)      /* 循环每一条边 */
    {
        n = Find(parent,edges[i].begin);
        m = Find(parent,edges[i].end);
        if (n != m) /* 假如 n 与 m 不等，说明此边没有与现有的生成树形成环路 */
        {
            parent[n] = m; /* 将此边的结尾顶点放入下标为起点的 parent 中。 */
                           /* 表示此顶点已经在生成树集合中 */
            printf("(%d, %d) %d\n", edges[i].begin, edges[i].end, edges[i].weight);
        }
    }
}

int main(void)
{
    MGraph G;
    CreateMGraph(&G);
    MiniSpanTree_Kruskal(G);
    return 0;
}
```

07 最短路径_Dijkstra

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXEDGE 20
#define MAXVEX 20
#define INFINITY 65535

typedef int Status;  /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */


typedef struct
{
```

```
    int vexs[MAXVEX];
    int arc[MAXVEX][MAXVEX];
    int numVertexes, numEdges;
}MGraph;

typedef int Patharc[MAXVEX];      /* 用于存储最短路径下标的数组 */
typedef int ShortPathTable[MAXVEX];/* 用于存储到各点最短路径的权值和 */

/* 构件图 */
void CreateMGraph(MGraph *G)
{
    int i, j;

    /* printf("请输入边数和顶点数:"); */
    G->numEdges=16;
    G->numVertexes=9;

    for (i = 0; i < G->numVertexes; i++)/* 初始化图 */
    {
        G->vexs[i]=i;
    }

    for (i = 0; i < G->numVertexes; i++)/* 初始化图 */
    {
        for ( j = 0; j < G->numVertexes; j++)
        {
            if (i==j)
                G->arc[i][j]=0;
            else
                G->arc[i][j] = G->arc[j][i] = INFINITY;
        }
    }

    G->arc[0][1]=1;
    G->arc[0][2]=5;
    G->arc[1][2]=3;
    G->arc[1][3]=7;
    G->arc[1][4]=5;

    G->arc[2][4]=1;
    G->arc[2][5]=7;
    G->arc[3][4]=2;
    G->arc[3][6]=3;
    G->arc[4][5]=3;
```

```
    G->arc[4][6]=6;
    G->arc[4][7]=9;
    G->arc[5][7]=5;
    G->arc[6][7]=2;
    G->arc[6][8]=7;

    G->arc[7][8]=4;


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

}

/*  Dijkstra 算法，求有向网 G 的 v0 顶点到其余顶点 v 的最短路径 P[v]及带权长度 D[v] */
/*  P[v]的值为前驱顶点下标,D[v]表示 v0 到 v 的最短路径长度和 */
void ShortestPath_Dijkstra(MGraph G, int v0, Patharc *P, ShortPathTable *D)
{
    int v,w,k,min;
    int final[MAXVEX];/* final[w]=1 表示求得顶点 v0 至 vw 的最短路径 */
    for(v=0; v<G.numVertexes; v++)      /* 初始化数据 */
    {
        final[v] = 0;                  /* 全部顶点初始化为未知最短路径状态 */
        (*D)[v] = G.arc[v0][v];/* 将与 v0 点有连线的顶点加上权值 */
        (*P)[v] = 0;                       /* 初始化路径数组 P 为 0  */
    }

    (*D)[v0] = 0;   /* v0 至 v0 路径为 0 */
    final[v0] = 1;      /* v0 至 v0 不需要求路径 */
    /* 开始主循环，每次求得 v0 到某个 v 顶点的最短路径 */
    for(v=1; v<G.numVertexes; v++)
    {
        min=INFINITY;      /* 当前所知离 v0 顶点的最近距离 */
        for(w=0; w<G.numVertexes; w++) /* 寻找离 v0 最近的顶点 */
        {
            if(!final[w] && (*D)[w]<min)
            {
                k=w;
```

```
                min = (*D)[w];      /* w 顶点离 v0 顶点更近 */
            }
        }
        final[k] = 1;       /* 将目前找到的最近的顶点置为 1 */
        for(w=0; w<G.numVertexes; w++) /* 修正当前最短路径及距离 */
        {
            /* 如果经过 v 顶点的路径比现在这条路径的长度短的话 */
            if(!final[w] && (min+G.arc[k][w]<(*D)[w]))
            { /*  说明找到了更短的路径，修改 D[w]和 P[w] */
                (*D)[w] = min + G.arc[k][w];  /* 修改当前路径长度 */
                (*P)[w]=k;
            }
        }
    }
}

int main(void)
{
    int i,j,v0;
    MGraph G;
    Patharc P;
    ShortPathTable D; /* 求某点到其余各点的最短路径 */
    v0=0;

    CreateMGraph(&G);

    ShortestPath_Dijkstra(G, v0, &P, &D);

    printf("最短路径倒序如下:\n");
    for(i=1;i<G.numVertexes;++i)
    {
        printf("v%d - v%d : ",v0,i);
        j=i;
        while(P[j]!=0)
        {
            printf("%d ",P[j]);
            j=P[j];
        }
        printf("\n");
    }
    printf("\n 源点到各顶点的最短路径长度为:\n");
    for(i=1;i<G.numVertexes;++i)
        printf("v%d - v%d : %d \n",G.vexs[0],G.vexs[i],D[i]);
    return 0;
```

}

08 最短路径_Floyd

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define MAXEDGE 20
#define MAXVEX 20
#define INFINITY 65535

typedef int Status;  /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等  */

typedef struct
{
	int vexs[MAXVEX];
	int arc[MAXVEX][MAXVEX];
	int numVertexes, numEdges;
}MGraph;

typedef int Patharc[MAXVEX][MAXVEX];
typedef int ShortPathTable[MAXVEX][MAXVEX];

/*  构件图  */
void CreateMGraph(MGraph *G)
{
	int i, j;

	/* printf("请输入边数和顶点数:"); */
	G->numEdges=16;
	G->numVertexes=9;

	for (i = 0; i < G->numVertexes; i++)/*  初始化图  */
	{
		G->vexs[i]=i;
	}
```

```
    for (i = 0; i < G->numVertexes; i++)/* 初始化图 */
    {
        for ( j = 0; j < G->numVertexes; j++)
        {
            if (i==j)
                G->arc[i][j]=0;
            else
                G->arc[i][j] = G->arc[j][i] = INFINITY;
        }
    }

    G->arc[0][1]=1;
    G->arc[0][2]=5;
    G->arc[1][2]=3;
    G->arc[1][3]=7;
    G->arc[1][4]=5;

    G->arc[2][4]=1;
    G->arc[2][5]=7;
    G->arc[3][4]=2;
    G->arc[3][6]=3;
    G->arc[4][5]=3;

    G->arc[4][6]=6;
    G->arc[4][7]=9;
    G->arc[5][7]=5;
    G->arc[6][7]=2;
    G->arc[6][8]=7;

    G->arc[7][8]=4;


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

}

/* Floyd 算法，求网图 G 中各顶点 v 到其余顶点 w 的最短路径 P[v][w]及带权长度 D[v][w]。*/
void ShortestPath_Floyd(MGraph G, Patharc *P, ShortPathTable *D)
```

```
{
    int v,w,k;
    for(v=0; v<G.numVertexes; ++v) /* 初始化 D 与 P */
    {
        for(w=0; w<G.numVertexes; ++w)
        {
            (*D)[v][w]=G.arc[v][w];  /* D[v][w]值即为对应点间的权值 */
            (*P)[v][w]=w;                    /* 初始化 P */
        }
    }
    for(k=0; k<G.numVertexes; ++k)
    {
        for(v=0; v<G.numVertexes; ++v)
        {
            for(w=0; w<G.numVertexes; ++w)
            {
                if ((*D)[v][w]>(*D)[v][k]+(*D)[k][w])
                {/* 如果经过下标为 k 顶点路径比原两点间路径更短 */
                    (*D)[v][w]=(*D)[v][k]+(*D)[k][w];/* 将当前两点间权值设为更小的一
个 */
                    (*P)[v][w]=(*P)[v][k];/* 路径设置为经过下标为 k 的顶点 */
                }
            }
        }
    }
}

int main(void)
{
    int v,w,k;
    MGraph G;

    Patharc P;
    ShortPathTable D; /* 求某点到其余各点的最短路径 */

    CreateMGraph(&G);

    ShortestPath_Floyd(G,&P,&D);

    printf("各顶点间最短路径如下:\n");
    for(v=0; v<G.numVertexes; ++v)
    {
        for(w=v+1; w<G.numVertexes; w++)
        {
```

```
            printf("v%d-v%d weight: %d ",v,w,D[v][w]);
            k=P[v][w];                  /* 获得第一个路径顶点下标 */
            printf(" path: %d",v);      /* 打印源点 */
            while(k!=w)                 /* 如果路径顶点下标不是终点 */
            {
                printf(" -> %d",k);     /* 打印路径顶点 */
                k=P[k][w];              /* 获得下一个路径顶点下标 */
            }
            printf(" -> %d\n",w);       /* 打印终点 */
        }
        printf("\n");
    }

    printf("最短路径 D\n");
    for(v=0; v<G.numVertexes; ++v)
    {
        for(w=0; w<G.numVertexes; ++w)
        {
            printf("%d\t",D[v][w]);
        }
        printf("\n");
    }
    printf("最短路径 P\n");
    for(v=0; v<G.numVertexes; ++v)
    {
        for(w=0; w<G.numVertexes; ++w)
        {
            printf("%d ",P[v][w]);
        }
        printf("\n");
    }

    return 0;
}
```

09 拓扑排序_TopologicalSort

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
```

```
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define MAXEDGE 20
#define MAXVEX 14
#define INFINITY 65535

typedef int Status;  /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */

/* 邻接矩阵结构 */
typedef struct
{
    int vexs[MAXVEX];
    int arc[MAXVEX][MAXVEX];
    int numVertexes, numEdges;
}MGraph;

/* 邻接表结构****************** */
typedef struct EdgeNode /* 边表结点  */
{
    int adjvex;     /* 邻接点域，存储该顶点对应的下标 */
    int weight;          /* 用于存储权值，对于非网图可以不需要 */
    struct EdgeNode *next; /* 链域，指向下一个邻接点 */
}EdgeNode;

typedef struct VertexNode /* 顶点表结点 */
{
    int in;    /* 顶点入度 */
    int data; /* 顶点域，存储顶点信息 */
    EdgeNode *firstedge;/* 边表头指针 */
}VertexNode, AdjList[MAXVEX];

typedef struct
{
    AdjList adjList;
    int numVertexes,numEdges; /* 图中当前顶点数和边数 */
}graphAdjList,*GraphAdjList;
/* ***************************** */


void CreateMGraph(MGraph *G)/* 构件图 */
{
    int i, j;
```

```
    /* printf("请输入边数和顶点数:"); */
    G->numEdges=MAXEDGE;
    G->numVertexes=MAXVEX;

    for (i = 0; i < G->numVertexes; i++)/*  初始化图  */
    {
        G->vexs[i]=i;
    }

    for (i = 0; i < G->numVertexes; i++)/*  初始化图  */
    {
        for ( j = 0; j < G->numVertexes; j++)
        {
            G->arc[i][j]=0;
        }
    }

    G->arc[0][4]=1;
    G->arc[0][5]=1;
    G->arc[0][11]=1;
    G->arc[1][2]=1;
    G->arc[1][4]=1;
    G->arc[1][8]=1;
    G->arc[2][5]=1;
    G->arc[2][6]=1;
    G->arc[2][9]=1;
    G->arc[3][2]=1;
    G->arc[3][13]=1;
    G->arc[4][7]=1;
    G->arc[5][8]=1;
    G->arc[5][12]=1;
    G->arc[6][5]=1;
    G->arc[8][7]=1;
    G->arc[9][10]=1;
    G->arc[9][11]=1;
    G->arc[10][13]=1;
    G->arc[12][9]=1;

}

/*  利用邻接矩阵构建邻接表  */
void CreateALGraph(MGraph G,GraphAdjList *GL)
{
    int i,j;
```

```
    EdgeNode *e;

    *GL = (GraphAdjList)malloc(sizeof(graphAdjList));

    (*GL)->numVertexes=G.numVertexes;
    (*GL)->numEdges=G.numEdges;
    for(i= 0;i <G.numVertexes;i++) /* 读入顶点信息，建立顶点表 */
    {
        (*GL)->adjList[i].in=0;
        (*GL)->adjList[i].data=G.vexs[i];
        (*GL)->adjList[i].firstedge=NULL;  /* 将边表置为空表 */
    }

    for(i=0;i<G.numVertexes;i++) /* 建立边表 */
    {
        for(j=0;j<G.numVertexes;j++)
        {
            if (G.arc[i][j]==1)
            {
                e=(EdgeNode *)malloc(sizeof(EdgeNode));
                e->adjvex=j;                    /* 邻接序号为 j  */
                e->next=(*GL)->adjList[i].firstedge;    /* 将当前顶点上的指向的结点指针
赋值给 e */
                (*GL)->adjList[i].firstedge=e;      /* 将当前顶点的指针指向 e  */
                (*GL)->adjList[j].in++;

            }
        }
    }

}


/* 拓扑排序，若 GL 无回路，则输出拓扑排序序列并返回 1，若有回路返回 0。 */
Status TopologicalSort(GraphAdjList GL)
{
    EdgeNode *e;
    int i,k,gettop;
    int top=0;   /* 用于栈指针下标  */
    int count=0;/* 用于统计输出顶点的个数  */
    int *stack;     /* 建栈将入度为 0 的顶点入栈  */
    stack=(int *)malloc(GL->numVertexes * sizeof(int) );

    for(i = 0; i<GL->numVertexes; i++)
```

```
        if(0 == GL->adjList[i].in) /* 将入度为 0 的顶点入栈 */
            stack[++top]=i;
    while(top!=0)
    {
        gettop=stack[top--];
        printf("%d -> ",GL->adjList[gettop].data);
        count++;            /* 输出 i 号顶点，并计数 */
        for(e = GL->adjList[gettop].firstedge; e; e = e->next)
        {
            k=e->adjvex;
            if( !(--GL->adjList[k].in) )  /* 将 i 号顶点的邻接点的入度减 1，如果减 1 后为 0，
则入栈 */
                stack[++top]=k;
        }
    }
    printf("\n");
    if(count < GL->numVertexes)
        return ERROR;
    else
        return OK;
}


int main(void)
{
    MGraph G;
    GraphAdjList GL;
    int result;
    CreateMGraph(&G);
    CreateALGraph(G,&GL);
    result=TopologicalSort(GL);
    printf("result:%d",result);

    return 0;
}
```

10 关键路径_CriticalPath

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"
```

```
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXEDGE 30
#define MAXVEX 30
#define INFINITY 65535

typedef int Status;  /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等  */

int *etv,*ltv; /*  事件最早发生时间和最迟发生时间数组，全局变量  */
int *stack2;     /*  用于存储拓扑序列的栈  */
int top2;       /*  用于 stack2 的指针  */

/*  邻接矩阵结构  */
typedef struct
{
      int vexs[MAXVEX];
      int arc[MAXVEX][MAXVEX];
      int numVertexes, numEdges;
}MGraph;

/*  邻接表结构****************** */
typedef struct EdgeNode /*  边表结点    */
{
      int adjvex;       /*  邻接点域，存储该顶点对应的下标  */
      int weight;            /*  用于存储权值，对于非网图可以不需要  */
      struct EdgeNode *next; /*  链域，指向下一个邻接点  */
}EdgeNode;

typedef struct VertexNode /*  顶点表结点  */
{
      int in;      /*  顶点入度  */
      int data; /*  顶点域，存储顶点信息  */
      EdgeNode *firstedge;/*  边表头指针  */
}VertexNode, AdjList[MAXVEX];

typedef struct
{
      AdjList adjList;
      int numVertexes,numEdges; /*  图中当前顶点数和边数  */
}graphAdjList,*GraphAdjList;
/* ************************** */
```

```
void CreateMGraph(MGraph *G)/*  构件图  */
{
    int i, j;
    /* printf("请输入边数和顶点数:"); */
    G->numEdges=13;
    G->numVertexes=10;

    for (i = 0; i < G->numVertexes; i++)/*  初始化图  */
    {
        G->vexs[i]=i;
    }

    for (i = 0; i < G->numVertexes; i++)/*  初始化图  */
    {
        for ( j = 0; j < G->numVertexes; j++)
        {
            if (i==j)
                G->arc[i][j]=0;
            else
                G->arc[i][j]=INFINITY;
        }
    }

    G->arc[0][1]=3;
    G->arc[0][2]=4;
    G->arc[1][3]=5;
    G->arc[1][4]=6;
    G->arc[2][3]=8;
    G->arc[2][5]=7;
    G->arc[3][4]=3;
    G->arc[4][6]=9;
    G->arc[4][7]=4;
    G->arc[5][7]=6;
    G->arc[6][9]=2;
    G->arc[7][8]=5;
    G->arc[8][9]=3;

}

/*  利用邻接矩阵构建邻接表  */
void CreateALGraph(MGraph G,GraphAdjList *GL)
{
```

```
    int i,j;
    EdgeNode *e;

    *GL = (GraphAdjList)malloc(sizeof(graphAdjList));

    (*GL)->numVertexes=G.numVertexes;
    (*GL)->numEdges=G.numEdges;
    for(i= 0;i <G.numVertexes;i++) /* 读入顶点信息，建立顶点表 */
    {
        (*GL)->adjList[i].in=0;
        (*GL)->adjList[i].data=G.vexs[i];
        (*GL)->adjList[i].firstedge=NULL;  /* 将边表置为空表 */
    }

    for(i=0;i<G.numVertexes;i++) /* 建立边表 */
    {
        for(j=0;j<G.numVertexes;j++)
        {
            if (G.arc[i][j]!=0 && G.arc[i][j]<INFINITY)
            {
                e=(EdgeNode *)malloc(sizeof(EdgeNode));
                e->adjvex=j;                    /* 邻接序号为 j */
                e->weight=G.arc[i][j];
                e->next=(*GL)->adjList[i].firstedge;    /* 将当前顶点上的指向的结点指针
赋值给 e */
                (*GL)->adjList[i].firstedge=e;      /* 将当前顶点的指针指向 e   */
                (*GL)->adjList[j].in++;

            }
        }
    }

}


/* 拓扑排序 */
Status TopologicalSort(GraphAdjList GL)
{    /* 若 GL 无回路，则输出拓扑排序序列并返回 1，若有回路返回 0。 */
    EdgeNode *e;
    int i,k,gettop;
    int top=0;   /* 用于栈指针下标   */
    int count=0;/* 用于统计输出顶点的个数 */
    int *stack;     /* 建栈将入度为 0 的顶点入栈   */
    stack=(int *)malloc(GL->numVertexes * sizeof(int) );
```

```
    for(i = 0; i<GL->numVertexes; i++)
        if(0 == GL->adjList[i].in) /* 将入度为 0 的顶点入栈 */
            stack[++top]=i;

    top2=0;
    etv=(int *)malloc(GL->numVertexes * sizeof(int) ); /* 事件最早发生时间数组 */
    for(i=0; i<GL->numVertexes; i++)
        etv[i]=0;       /* 初始化 */
    stack2=(int *)malloc(GL->numVertexes * sizeof(int) );/* 初始化拓扑序列栈 */

    printf("TopologicalSort:\t");
    while(top!=0)
    {
        gettop=stack[top--];
        printf("%d -> ",GL->adjList[gettop].data);
        count++;            /* 输出 i 号顶点，并计数 */

        stack2[++top2]=gettop;          /* 将弹出的顶点序号压入拓扑序列的栈 */

        for(e = GL->adjList[gettop].firstedge; e; e = e->next)
        {
            k=e->adjvex;
            if( !(--GL->adjList[k].in) )          /* 将 i 号顶点的邻接点的入度减 1，如果减 1
后为 0，则入栈 */
                stack[++top]=k;

            if((etv[gettop] + e->weight)>etv[k])      /* 求各顶点事件的最早发生时间 etv 值
*/
                etv[k] = etv[gettop] + e->weight;
        }
    }
    printf("\n");
    if(count < GL->numVertexes)
        return ERROR;
    else
        return OK;
}

/* 求关键路径,GL 为有向网，输出 G 的各项关键活动 */
void CriticalPath(GraphAdjList GL)
{
    EdgeNode *e;
    int i,gettop,k,j;
    int ete,lte;   /* 声明活动最早发生时间和最迟发生时间变量 */
```

```
    TopologicalSort(GL);    /* 求拓扑序列，计算数组 etv 和 stack2 的值 */
    ltv=(int *)malloc(GL->numVertexes*sizeof(int));/* 事件最早发生时间数组 */
    for(i=0; i<GL->numVertexes; i++)
        ltv[i]=etv[GL->numVertexes-1];      /* 初始化 */

    printf("etv:\t");
    for(i=0; i<GL->numVertexes; i++)
        printf("%d -> ",etv[i]);
    printf("\n");

    while(top2!=0)      /* 出栈是求 ltv */
    {
        gettop=stack2[top2--];
        for(e = GL->adjList[gettop].firstedge; e; e = e->next)          /* 求各顶点事件的最迟发生时间 ltv 值 */
        {
            k=e->adjvex;
            if(ltv[k] - e->weight < ltv[gettop])
                ltv[gettop] = ltv[k] - e->weight;
        }
    }

    printf("ltv:\t");
    for(i=0; i<GL->numVertexes; i++)
        printf("%d -> ",ltv[i]);
    printf("\n");

    for(j=0; j<GL->numVertexes; j++)            /* 求 ete,lte 和关键活动 */
    {
        for(e = GL->adjList[j].firstedge; e; e = e->next)
        {
            k=e->adjvex;
            ete = etv[j];           /* 活动最早发生时间 */
            lte = ltv[k] - e->weight; /* 活动最迟发生时间 */
            if(ete == lte)      /* 两者相等即在关键路径上 */
                printf("<v%d - v%d> length: %d \n",GL->adjList[j].data,GL->adjList[k].data,e->weight);
        }
    }
}


int main(void)
{
```

```
    MGraph G;
    GraphAdjList GL;
    CreateMGraph(&G);
    CreateALGraph(G,&GL);
    CriticalPath(GL);
    return 0;
}
```

第八章 查找

01 静态查找_Search

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define MAXSIZE 100 /* 存储空间初始分配量 */

typedef int Status; /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */

int F[100]; /* 斐波那契数列 */

/* 无哨兵顺序查找，a 为数组，n 为要查找的数组个数，key 为要查找的关键字 */
int Sequential_Search(int *a,int n,int key)
{
    int i;
    for(i=1;i<=n;i++)
    {
        if (a[i]==key)
            return i;
    }
    return 0;
}
/* 有哨兵顺序查找 */
int Sequential_Search2(int *a,int n,int key)
{
```

```
    int i;
    a[0]=key;
    i=n;
    while(a[i]!=key)
    {
        i--;
    }
    return i;
}

/* 折半查找 */
int Binary_Search(int *a,int n,int key)
{
    int low,high,mid;
    low=1;    /* 定义最低下标为记录首位 */
    high=n;   /* 定义最高下标为记录末位 */
    while(low<=high)
    {
        mid=(low+high)/2; /* 折半 */
        if (key<a[mid])       /* 若查找值比中值小 */
            high=mid-1;          /* 最高下标调整到中位下标小一位 */
        else if (key>a[mid])/* 若查找值比中值大 */
            low=mid+1;           /* 最低下标调整到中位下标大一位 */
        else
        {
            return mid;          /* 若相等则说明 mid 即为查找到的位置 */
        }

    }
    return 0;
}

/* 插值查找 */
int Interpolation_Search(int *a,int n,int key)
{
    int low,high,mid;
    low=1;    /* 定义最低下标为记录首位 */
    high=n;   /* 定义最高下标为记录末位 */
    while(low<=high)
    {
        mid=low+ (high-low)*(key-a[low])/(a[high]-a[low]); /* 插值 */
        if (key<a[mid])       /* 若查找值比插值小 */
            high=mid-1;          /* 最高下标调整到插值下标小一位 */
        else if (key>a[mid])/* 若查找值比插值大 */
```

```
            low=mid+1;          /* 最低下标调整到插值下标大一位 */
        else
            return mid;         /* 若相等则说明 mid 即为查找到的位置 */
    }
    return 0;
}

/* 斐波那契查找 */
int Fibonacci_Search(int *a,int n,int key)
{
    int low,high,mid,i,k=0;
    low=1;    /* 定义最低下标为记录首位 */
    high=n;   /* 定义最高下标为记录末位 */
    while(n>F[k]-1)
        k++;
    for (i=n;i<F[k]-1;i++)
        a[i]=a[n];

    while(low<=high)
    {
        mid=low+F[k-1]-1;
        if (key<a[mid])
        {
            high=mid-1;
            k=k-1;
        }
        else if (key>a[mid])
        {
            low=mid+1;
            k=k-2;
        }
        else
        {
            if (mid<=n)
                return mid;         /* 若相等则说明 mid 即为查找到的位置 */
            else
                return n;
        }

    }
    return 0;
}
```

```
int main(void)
{

	int a[MAXSIZE+1],i,result;
	int arr[MAXSIZE]={0,1,16,24,35,47,59,62,73,88,99};

	for(i=0;i<=MAXSIZE;i++)
	{
		a[i]=i;
	}
	result=Sequential_Search(a,MAXSIZE,MAXSIZE);
	printf("Sequential_Search:%d \n",result);
	result=Sequential_Search2(a,MAXSIZE,1);
	printf("Sequential_Search2:%d \n",result);

	result=Binary_Search(arr,10,62);
	printf("Binary_Search:%d \n",result);


	result=Interpolation_Search(arr,10,62);
	printf("Interpolation_Search:%d \n",result);


	F[0]=0;
	F[1]=1;
	for(i = 2;i < 100;i++)
	{
		F[i] = F[i-1] + F[i-2];
	}
	result=Fibonacci_Search(arr,10,62);
	printf("Fibonacci_Search:%d \n",result);

	return 0;
}
```

02 二叉排序树_BinarySortTree

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
```

```
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define MAXSIZE 100 /* 存储空间初始分配量 */

typedef int Status; /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */

/* 二叉树的二叉链表结点结构定义 */
typedef  struct BiTNode    /* 结点结构 */
{
    int data; /* 结点数据 */
    struct BiTNode *lchild, *rchild;    /* 左右孩子指针 */
} BiTNode, *BiTree;



/* 递归查找二叉排序树 T 中是否存在 key, */
/* 指针 f 指向 T 的双亲，其初始调用值为 NULL */
/* 若查找成功，则指针 p 指向该数据元素结点，并返回 TRUE */
/* 否则指针 p 指向查找路径上访问的最后一个结点并返回 FALSE */
Status SearchBST(BiTree T, int key, BiTree f, BiTree *p)
{
    if (!T)    /*  查找不成功 */
    {
        *p = f;
        return FALSE;
    }
    else if (key==T->data) /*  查找成功 */
    {
        *p = T;
        return TRUE;
    }
    else if (key<T->data)
        return SearchBST(T->lchild, key, T, p);  /*  在左子树中继续查找 */
    else
        return SearchBST(T->rchild, key, T, p);  /*  在右子树中继续查找 */
}



/*  当二叉排序树 T 中不存在关键字等于 key 的数据元素时， */
/*  插入 key 并返回 TRUE，否则返回 FALSE */
```

```
Status InsertBST(BiTree *T, int key)
{
    BiTree p,s;
    if (!SearchBST(*T, key, NULL, &p)) /* 查找不成功 */
    {
        s = (BiTree)malloc(sizeof(BiTNode));
        s->data = key;
        s->lchild = s->rchild = NULL;
        if (!p)
            *T = s;                 /*  插入 s 为新的根结点 */
        else if (key<p->data)
            p->lchild = s;  /*  插入 s 为左孩子 */
        else
            p->rchild = s;  /*  插入 s 为右孩子 */
        return TRUE;
    }
    else
        return FALSE;   /*  树中已有关键字相同的结点，不再插入 */
}

/* 从二叉排序树中删除结点 p，并重接它的左或右子树。 */
Status Delete(BiTree *p)
{
    BiTree q,s;
    if((*p)->rchild==NULL) /* 右子树空则只需重接它的左子树（待删结点是叶子也走此分支)
*/
    {
        q=*p; *p=(*p)->lchild; free(q);
    }
    else if((*p)->lchild==NULL) /* 只需重接它的右子树 */
    {
        q=*p; *p=(*p)->rchild; free(q);
    }
    else /* 左右子树均不空 */
    {
        q=*p; s=(*p)->lchild;
        while(s->rchild) /* 转左，然后向右到尽头（找待删结点的前驱） */
        {
            q=s;
            s=s->rchild;
        }
        (*p)->data=s->data; /*  s 指向被删结点的直接前驱（将被删结点前驱的值取代被删
结点的值） */
        if(q!=*p)
```

```
            q->rchild=s->lchild; /*   重接 q 的右子树  */
        else
            q->lchild=s->lchild; /*   重接 q 的左子树  */
        free(s);
    }
    return TRUE;
}

/*  若二叉排序树 T 中存在关键字等于 key 的数据元素时，则删除该数据元素结点, */
/*  并返回 TRUE；否则返回 FALSE。  */
Status DeleteBST(BiTree *T,int key)
{
    if(!*T) /*  不存在关键字等于 key 的数据元素  */
        return FALSE;
    else
    {
        if (key==(*T)->data) /*  找到关键字等于 key 的数据元素  */
            return Delete(T);
        else if (key<(*T)->data)
            return DeleteBST(&(*T)->lchild,key);
        else
            return DeleteBST(&(*T)->rchild,key);

    }
}

int main(void)
{
    int i;
    int a[10]={62,88,58,47,35,73,51,99,37,93};
    BiTree T=NULL;

    for(i=0;i<10;i++)
    {
        InsertBST(&T, a[i]);
    }
    DeleteBST(&T,93);
    DeleteBST(&T,47);
    printf("本样例建议断点跟踪查看二叉排序树结构");
    return 0;
}
```

03 平衡二叉树_AVLTree

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0
#define MAXSIZE 100 /* 存储空间初始分配量 */

typedef int Status;  /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */


/* 二叉树的二叉链表结点结构定义 */
typedef    struct BiTNode      /* 结点结构 */
{
      int data;  /* 结点数据 */
      int bf; /*   结点的平衡因子 */
      struct BiTNode *lchild, *rchild;     /* 左右孩子指针 */
} BiTNode, *BiTree;


/* 对以 p 为根的二叉排序树作右旋处理， */
/* 处理之后 p 指向新的树根结点，即旋转处理之前的左子树的根结点 */
void R_Rotate(BiTree *P)
{
      BiTree L;
      L=(*P)->lchild; /*   L 指向 P 的左子树根结点 */
      (*P)->lchild=L->rchild; /*   L 的右子树挂接为 P 的左子树 */
      L->rchild=(*P);
      *P=L; /*   P 指向新的根结点 */
}

/* 对以 P 为根的二叉排序树作左旋处理， */
/* 处理之后 P 指向新的树根结点，即旋转处理之前的右子树的根结点 0  */
void L_Rotate(BiTree *P)
{
      BiTree R;
      R=(*P)->rchild; /*   R 指向 P 的右子树根结点 */
```

```
    (*P)->rchild=R->lchild; /* R 的左子树挂接为 P 的右子树 */
    R->lchild=(*P);
    *P=R; /*  P 指向新的根结点 */
}

#define LH +1 /*  左高 */
#define EH 0  /*  等高 */
#define RH -1 /*  右高 */

/*  对以指针 T 所指结点为根的二叉树作左平衡旋转处理 */
/*  本算法结束时，指针 T 指向新的根结点 */
void LeftBalance(BiTree *T)
{
    BiTree L,Lr;
    L=(*T)->lchild; /*  L 指向 T 的左子树根结点 */
    switch(L->bf)
    { /*  检查 T 的左子树的平衡度，并作相应平衡处理 */
        case LH: /*  新结点插入在 T 的左孩子的左子树上，要作单右旋处理 */
            (*T)->bf=L->bf=EH;
            R_Rotate(T);
            break;
        case RH: /*  新结点插入在 T 的左孩子的右子树上，要作双旋处理 */
            Lr=L->rchild; /*  Lr 指向 T 的左孩子的右子树根 */
            switch(Lr->bf)
            { /*  修改 T 及其左孩子的平衡因子 */
                case LH: (*T)->bf=RH;
                         L->bf=EH;
                         break;
                case EH: (*T)->bf=L->bf=EH;
                         break;
                case RH: (*T)->bf=EH;
                         L->bf=LH;
                         break;
            }
            Lr->bf=EH;
            L_Rotate(&(*T)->lchild); /*  对 T 的左子树作左旋平衡处理 */
            R_Rotate(T); /*  对 T 作右旋平衡处理 */
    }
}

/*  对以指针 T 所指结点为根的二叉树作右平衡旋转处理， */
/*  本算法结束时，指针 T 指向新的根结点 */
void RightBalance(BiTree *T)
{
```

```
    BiTree R,Rl;
    R=(*T)->rchild; /*  R 指向 T 的右子树根结点 */
    switch(R->bf)
    { /*  检查 T 的右子树的平衡度，并作相应平衡处理 */
     case RH: /*  新结点插入在 T 的右孩子的右子树上，要作单左旋处理 */
                (*T)->bf=R->bf=EH;
                L_Rotate(T);
                break;
     case LH: /*  新结点插入在 T 的右孩子的左子树上，要作双旋处理 */
                Rl=R->lchild; /*  Rl 指向 T 的右孩子的左子树根 */
                switch(Rl->bf)
                { /*  修改 T 及其右孩子的平衡因子 */
                  case RH: (*T)->bf=LH;
                           R->bf=EH;
                           break;
                  case EH: (*T)->bf=R->bf=EH;
                           break;
                  case LH: (*T)->bf=EH;
                           R->bf=RH;
                           break;
                }
                Rl->bf=EH;
                R_Rotate(&(*T)->rchild); /*  对 T 的右子树作右旋平衡处理 */
                L_Rotate(T); /*  对 T 作左旋平衡处理 */
    }
}

/*  若在平衡的二叉排序树 T 中不存在和 e 有相同关键字的结点，则插入一个 */
/*  数据元素为 e 的新结点，并返回 1，否则返回 0。若因插入而使二叉排序树 */
/*  失去平衡，则作平衡旋转处理，布尔变量 taller 反映 T 长高与否。 */
Status InsertAVL(BiTree *T,int e,Status *taller)
{
    if(!*T)
    { /*  插入新结点，树“长高”，置 taller 为 TRUE */
        *T=(BiTree)malloc(sizeof(BiTNode));
        (*T)->data=e; (*T)->lchild=(*T)->rchild=NULL; (*T)->bf=EH;
        *taller=TRUE;
    }
    else
    {
        if (e==(*T)->data)
        { /*  树中已存在和 e 有相同关键字的结点则不再插入 */
            *taller=FALSE; return FALSE;
        }
```

```
        if (e<(*T)->data)
        { /*  应继续在 T 的左子树中进行搜索 */
            if(!InsertAVL(&(*T)->lchild,e,taller)) /*  未插入 */
                return FALSE;
            if(taller) /*  已插入到 T 的左子树中且左子树“长高” */
                switch((*T)->bf) /*  检查 T 的平衡度 */
                {
                    case LH: /*  原本左子树比右子树高，需要作左平衡处理 */
                            LeftBalance(T);      *taller=FALSE; break;
                    case EH: /*  原本左、右子树等高，现因左子树增高而使树增高 */
                            (*T)->bf=LH; *taller=TRUE; break;
                    case RH: /*  原本右子树比左子树高，现左、右子树等高 */
                            (*T)->bf=EH; *taller=FALSE; break;
                }
        }
        else
        { /*  应继续在 T 的右子树中进行搜索 */
            if(!InsertAVL(&(*T)->rchild,e,taller)) /*  未插入 */
                return FALSE;
            if(*taller) /*  已插入到 T 的右子树且右子树“长高” */
                switch((*T)->bf) /*  检查 T 的平衡度 */
                {
                    case LH: /*  原本左子树比右子树高，现左、右子树等高 */
                            (*T)->bf=EH; *taller=FALSE;  break;
                    case EH: /*  原本左、右子树等高，现因右子树增高而使树增高  */
                            (*T)->bf=RH; *taller=TRUE; break;
                    case RH: /*  原本右子树比左子树高，需要作右平衡处理 */
                            RightBalance(T); *taller=FALSE; break;
                }
        }
    }
    return TRUE;
}

int main(void)
{
    int i;
    int a[10]={3,2,1,4,5,6,7,10,9,8};
    BiTree T=NULL;
    Status taller;
    for(i=0;i<10;i++)
    {
        InsertAVL(&T,a[i],&taller);
    }
```

```
    printf("本样例建议断点跟踪查看平衡二叉树结构");
    return 0;
}
```

04B 树_Btree

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 100 /* 存储空间初始分配量 */

#define m 3 /*  B 树的阶，暂设为 3 */
#define N 17 /*  数据元素个数 */
#define MAX 5 /*  字符串最大长度+1  */

typedef int Status;  /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */

typedef struct BTNode
{
    int keynum; /*  结点中关键字个数，即结点的大小 */
    struct BTNode *parent; /*  指向双亲结点 */
    struct Node /*  结点向量类型 */
    {
        int key; /*  关键字向量 */
        struct BTNode *ptr; /*  子树指针向量 */
        int recptr; /*  记录指针向量 */
    }node[m+1]; /*  key,recptr 的 0 号单元未用 */
}BTNode,*BTree; /*  B 树结点和 B 树的类型 */

typedef struct
{
    BTNode *pt; /*  指向找到的结点 */
    int i; /*  1..m，在结点中的关键字序号 */
    int tag; /*  1:查找成功，O:查找失败 */
}Result; /*  B 树的查找结果类型 */
```

```
/*   在 p->node[1..keynum].key 中查找 i,使得 p->node[i].key≤K＜p->node[i+1].key */
int Search(BTree p, int K)
{
      int i=0,j;
      for(j=1;j<=p->keynum;j++)
       if(p->node[j].key<=K)
          i=j;
      return i;
}

/*   在 m 阶 B 树 T 上查找关键字 K，返回结果(pt,i,tag)。若查找成功，则特征值 */
/*   tag=1，指针 pt 所指结点中第 i 个关键字等于 K；否则特征值 tag=0，等于 K 的  */
/*   关键字应插入在指针 Pt 所指结点中第 i 和第 i+1 个关键字之间。 */
Result SearchBTree(BTree T, int K)
{
      BTree p=T,q=NULL; /*   初始化，p 指向待查结点，q 指向 p 的双亲   */
      Status found=FALSE;
      int i=0;
      Result r;
      while(p&&!found)
      {
       i=Search(p,K); /*    p->node[i].key≤K<p->node[i+1].key   */
       if(i>0&&p->node[i].key==K) /*   找到待查关键字 */
          found=TRUE;
       else
       {
          q=p;
          p=p->node[i].ptr;
       }
      }
      r.i=i;
      if(found) /*   查找成功   */
      {
       r.pt=p;
       r.tag=1;
      }
      else /*    查找不成功，返回 K 的插入位置信息 */
      {
       r.pt=q;
       r.tag=0;
      }
      return r;
}
```

```
/* 将 r->key、r 和 ap 分别插入到 q->key[i+1]、q->recptr[i+1]和 q->ptr[i+1]中 */
void Insert(BTree *q,int i,int key,BTree ap)
{
    int j;
    for(j=(*q)->keynum;j>i;j--) /*  空出(*q)->node[i+1]  */
        (*q)->node[j+1]=(*q)->node[j];
    (*q)->node[i+1].key=key;
    (*q)->node[i+1].ptr=ap;
    (*q)->node[i+1].recptr=key;
    (*q)->keynum++;
}

/* 将结点 q 分裂成两个结点，前一半保留，后一半移入新生结点 ap */
void split(BTree *q,BTree *ap)
{
    int i,s=(m+1)/2;
    *ap=(BTree)malloc(sizeof(BTNode)); /*  生成新结点 ap */
    (*ap)->node[0].ptr=(*q)->node[s].ptr; /*  后一半移入 ap */
    for(i=s+1;i<=m;i++)
    {
        (*ap)->node[i-s]=(*q)->node[i];
        if((*ap)->node[i-s].ptr)
            (*ap)->node[i-s].ptr->parent=*ap;
    }
    (*ap)->keynum=m-s;
    (*ap)->parent=(*q)->parent;
    (*q)->keynum=s-1; /*  q 的前一半保留，修改 keynum */
}

/* 生成含信息(T,r,ap)的新的根结点&T，原 T 和 ap 为子树指针 */
void NewRoot(BTree *T,int key,BTree ap)
{
    BTree p;
    p=(BTree)malloc(sizeof(BTNode));
    p->node[0].ptr=*T;
    *T=p;
    if((*T)->node[0].ptr)
        (*T)->node[0].ptr->parent=*T;
    (*T)->parent=NULL;
    (*T)->keynum=1;
    (*T)->node[1].key=key;
    (*T)->node[1].recptr=key;
    (*T)->node[1].ptr=ap;
    if((*T)->node[1].ptr)
```

```
        (*T)->node[1].ptr->parent=*T;
}

/*  在 m 阶 B 树 T 上结点*q 的 key[i]与 key[i+1]之间插入关键字 K 的指针 r。若引起 */
/*  结点过大,则沿双亲链进行必要的结点分裂调整,使 T 仍是 m 阶 B 树。 */
void InsertBTree(BTree *T,int key,BTree q,int i)
{
    BTree ap=NULL;
    Status finished=FALSE;
    int s;
    int rx;
    rx=key;
    while(q&&!finished)
    {
        Insert(&q,i,rx,ap); /*  将 r->key、r 和 ap 分别插入到 q->key[i+1]、q->recptr[i+1]和
q->ptr[i+1]中  */
        if(q->keynum<m)
            finished=TRUE; /*  插入完成 */
        else
        { /*  分裂结点*q */
            s=(m+1)/2;
            rx=q->node[s].recptr;
            split(&q,&ap); /*  将 q->key[s+1..m],q->ptr[s..m]和 q->recptr[s+1..m]移入新结点
*ap  */
            q=q->parent;
            if(q)
                i=Search(q,key); /*  在双亲结点*q 中查找 rx->key 的插入位置  */
        }
    }
    if(!finished) /*  T 是空树(参数 q 初值为 NULL)或根结点已分裂为结点*q 和*ap */
        NewRoot(T,rx,ap); /*  生成含信息(T,rx,ap)的新的根结点*T，原 T 和 ap 为子树指针
*/
}


void print(BTNode c,int i) /*  TraverseDSTable()调用的函数  */
{
    printf("(%d)",c.node[i].key);
}

int main()
{
    int r[N]={22,16,41,58,8,11,12,16,17,22,23,31,41,52,58,59,61};
    BTree T=NULL;
```

```
    Result s;
    int i;
    for(i=0;i<N;i++)
    {
        s=SearchBTree(T,r[i]);
        if(!s.tag)
            InsertBTree(&T,r[i],s.pt,s.i);
    }
    printf("\n 请输入待查找记录的关键字: ");
    scanf("%d",&i);
    s=SearchBTree(T,i);
    if(s.tag)
        print(*(s.pt),s.i);
    else
        printf("没找到");
    printf("\n");

    return 0;
}
```

05 散列表_HashTable

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#include "math.h"
#include "time.h"

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 100 /* 存储空间初始分配量 */

#define SUCCESS 1
#define UNSUCCESS 0
#define HASHSIZE 12 /* 定义散列表长为数组的长度 */
#define NULLKEY -32768

typedef int Status; /* Status 是函数的类型,其值是函数结果状态代码，如 OK 等 */

typedef struct
{
```

```
    int *elem; /* 数据元素存储基址，动态分配数组 */
    int count; /*  当前数据元素个数 */
}HashTable;

int m=0; /* 散列表表长，全局变量 */

/* 初始化散列表 */
Status InitHashTable(HashTable *H)
{
    int i;
    m=HASHSIZE;
    H->count=m;
    H->elem=(int *)malloc(m*sizeof(int));
    for(i=0;i<m;i++)
        H->elem[i]=NULLKEY;
    return OK;
}

/* 散列函数 */
int Hash(int key)
{
    return key % m; /* 除留余数法 */
}

/* 插入关键字进散列表 */
void InsertHash(HashTable *H,int key)
{
    int addr = Hash(key); /* 求散列地址 */
    while (H->elem[addr] != NULLKEY) /* 如果不为空，则冲突 */
    {
        addr = (addr+1) % m; /* 开放定址法的线性探测 */
    }
    H->elem[addr] = key; /* 直到有空位后插入关键字 */
}

/* 散列表查找关键字 */
Status SearchHash(HashTable H,int key,int *addr)
{
    *addr = Hash(key);  /* 求散列地址 */
    while(H.elem[*addr] != key) /* 如果不为空，则冲突 */
    {
        *addr = (*addr+1) % m; /* 开放定址法的线性探测 */
        if (H.elem[*addr] == NULLKEY || *addr == Hash(key)) /* 如果循环回到原点 */
            return UNSUCCESS;/* 则说明关键字不存在 */
```

```
    }
    return SUCCESS;
}

int main()
{
    int arr[HASHSIZE]={12,67,56,16,25,37,22,29,15,47,48,34};
    int i,p,key,result;
    HashTable H;

    key=39;

    InitHashTable(&H);
    for(i=0;i<m;i++)
        InsertHash(&H,arr[i]);

    result=SearchHash(H,key,&p);
    if (result)
        printf("查找 %d 的地址为：%d \n",key,p);
    else
        printf("查找 %d 失败。\n",key);

    for(i=0;i<m;i++)
    {
        key=arr[i];
        SearchHash(H,key,&p);
        printf("查找 %d 的地址为：%d \n",key,p);
    }

    return 0;
}
```

第九章 排序

01 排序_Sort

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <io.h>
#include <math.h>
```

```
#include <time.h>

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAX_LENGTH_INSERT_SORT 7 /* 用于快速排序时判断是否选用插入排序阈值 */

typedef int Status;


#define MAXSIZE 10000   /* 用于要排序数组个数最大值，可根据需要修改 */
typedef struct
{
    int r[MAXSIZE+1];  /* 用于存储要排序数组，r[0]用作哨兵或临时变量 */
    int length;              /* 用于记录顺序表的长度 */
}SqList;

/* 交换 L 中数组 r 的下标为 i 和 j 的值 */
void swap(SqList *L,int i,int j)
{
    int temp=L->r[i];
    L->r[i]=L->r[j];
    L->r[j]=temp;
}

void print(SqList L)
{
    int i;
    for(i=1;i<L.length;i++)
        printf("%d,",L.r[i]);
    printf("%d",L.r[i]);
    printf("\n");
}

/* 对顺序表 L 作交换排序（冒泡排序初级版） */
void BubbleSort0(SqList *L)
{
    int i,j;
    for(i=1;i<L->length;i++)
    {
        for(j=i+1;j<=L->length;j++)
        {
```

```
            if(L->r[i]>L->r[j])
            {
                swap(L,i,j);/* 交换 L->r[i]与 L->r[j]的值 */
            }
        }
    }
}

/* 对顺序表 L 作冒泡排序 */
void BubbleSort(SqList *L)
{
    int i,j;
    for(i=1;i<L->length;i++)
    {
        for(j=L->length-1;j>=i;j--)  /* 注意 j 是从后往前循环 */
        {
            if(L->r[j]>L->r[j+1]) /* 若前者大于后者（注意这里与上一算法的差异）*/
            {
                swap(L,j,j+1);/* 交换 L->r[j]与 L->r[j+1]的值 */
            }
        }
    }
}

/* 对顺序表 L 作改进冒泡算法 */
void BubbleSort2(SqList *L)
{
    int i,j;
    Status flag=TRUE;           /* flag 用来作为标记 */
    for(i=1;i<L->length && flag;i++) /* 若 flag 为 true 说明有过数据交换，否则停止循环 */
    {
        flag=FALSE;             /* 初始为 False */
        for(j=L->length-1;j>=i;j--)
        {
            if(L->r[j]>L->r[j+1])
            {
                swap(L,j,j+1);/* 交换 L->r[j]与 L->r[j+1]的值 */
                flag=TRUE;      /* 如果有数据交换，则 flag 为 true */
            }
        }
    }
}
```

```
/* 对顺序表 L 作简单选择排序 */
void SelectSort(SqList *L)
{
    int i,j,min;
    for(i=1;i<L->length;i++)
    {
        min = i;                            /* 将当前下标定义为最小值下标 */
        for (j = i+1;j<=L->length;j++)/* 循环之后的数据 */
        {
            if (L->r[min]>L->r[j])      /* 如果有小于当前最小值的关键字 */
                min = j;                    /* 将此关键字的下标赋值给 min */
        }
        if(i!=min)                          /* 若 min 不等于 i，说明找到最小值，交换 */
            swap(L,i,min);                  /* 交换 L->r[i]与 L->r[min]的值 */
    }
}

/* 对顺序表 L 作直接插入排序 */
void InsertSort(SqList *L)
{
    int i,j;
    for(i=2;i<=L->length;i++)
    {
        if (L->r[i]<L->r[i-1]) /* 需将 L->r[i]插入有序子表 */
        {
            L->r[0]=L->r[i]; /* 设置哨兵 */
            for(j=i-1;L->r[j]>L->r[0];j--)
                L->r[j+1]=L->r[j]; /* 记录后移 */
            L->r[j+1]=L->r[0]; /* 插入到正确位置 */
        }
    }
}

/* 对顺序表 L 作希尔排序 */
void ShellSort(SqList *L)
{
    int i,j,k=0;
    int increment=L->length;
    do
    {
        increment=increment/3+1;/* 增量序列 */
        for(i=increment+1;i<=L->length;i++)
        {
            if (L->r[i]<L->r[i-increment])/*  需将 L->r[i]插入有序增量子表 */
```

```
            {
                L->r[0]=L->r[i]; /*  暂存在 L->r[0] */
                for(j=i-increment;j>0 && L->r[0]<L->r[j];j-=increment)
                    L->r[j+increment]=L->r[j]; /*  记录后移，查找插入位置 */
                L->r[j+increment]=L->r[0]; /*  插入 */
            }
        }
        printf("  第%d 趟排序结果: ",++k);
        print(*L);
    }
    while(increment>1);

}


/* 堆排序********************************** */

/* 已知 L->r[s..m]中记录的关键字除 L->r[s]之外均满足堆的定义， */
/* 本函数调整 L->r[s]的关键字,使 L->r[s..m]成为一个大顶堆 */
void HeapAdjust(SqList *L,int s,int m)
{
    int temp,j;
    temp=L->r[s];
    for(j=2*s;j<=m;j*=2) /* 沿关键字较大的孩子结点向下筛选 */
    {
        if(j<m && L->r[j]<L->r[j+1])
            ++j; /* j 为关键字中较大的记录的下标 */
        if(temp>=L->r[j])
            break; /* rc 应插入在位置 s 上 */
        L->r[s]=L->r[j];
        s=j;
    }
    L->r[s]=temp; /* 插入 */
}

/*  对顺序表 L 进行堆排序 */
void HeapSort(SqList *L)
{
    int i;
    for(i=L->length/2;i>0;i--) /*  把 L 中的 r 构建成一个大根堆 */
        HeapAdjust(L,i,L->length);

    for(i=L->length;i>1;i--)
    {
```

```
            swap(L,1,i); /* 将堆顶记录和当前未经排序子序列的最后一个记录交换 */
            HeapAdjust(L,1,i-1); /*  将 L->r[1..i-1]重新调整为大根堆 */
        }
}

/* ************************************* */


/* 归并排序********************************** */

/* 将有序的 SR[i..m]和 SR[m+1..n]归并为有序的 TR[i..n] */
void Merge(int SR[],int TR[],int i,int m,int n)
{
    int j,k,l;
    for(j=m+1,k=i;i<=m && j<=n;k++)  /* 将 SR 中记录由小到大地并入 TR */
    {
        if (SR[i]<SR[j])
            TR[k]=SR[i++];
        else
            TR[k]=SR[j++];
    }
    if(i<=m)
    {
        for(l=0;l<=m-i;l++)
            TR[k+l]=SR[i+l];          /* 将剩余的 SR[i..m]复制到 TR */
    }
    if(j<=n)
    {
        for(l=0;l<=n-j;l++)
            TR[k+l]=SR[j+l];          /* 将剩余的 SR[j..n]复制到 TR */
    }
}


/* 递归法 */
/* 将 SR[s..t]归并排序为 TR1[s..t] */
void MSort(int SR[],int TR1[],int s, int t)
{
    int m;
    int TR2[MAXSIZE+1];
    if(s==t)
        TR1[s]=SR[s];
    else
    {
```

```
        m=(s+t)/2;                    /* 将 SR[s..t]平分为 SR[s..m]和 SR[m+1..t] */
        MSort(SR,TR2,s,m);      /* 递归地将 SR[s..m]归并为有序的 TR2[s..m] */
        MSort(SR,TR2,m+1,t);    /* 递归地将 SR[m+1..t]归并为有序的 TR2[m+1..t] */
        Merge(TR2,TR1,s,m,t);   /* 将 TR2[s..m]和 TR2[m+1..t]归并到 TR1[s..t] */
    }
}

/* 对顺序表 L 作归并排序 */
void MergeSort(SqList *L)
{
    MSort(L->r,L->r,1,L->length);
}

/* 非递归法 */
/* 将 SR[]中相邻长度为 s 的子序列两两归并到 TR[] */
void MergePass(int SR[],int TR[],int s,int n)
{
    int i=1;
    int j;
    while(i <= n-2*s+1)
    {/* 两两归并 */
        Merge(SR,TR,i,i+s-1,i+2*s-1);
        i=i+2*s;
    }
    if(i<n-s+1) /* 归并最后两个序列 */
        Merge(SR,TR,i,i+s-1,n);
    else /* 若最后只剩下单个子序列 */
        for(j =i;j <= n;j++)
            TR[j] = SR[j];
}

/* 对顺序表 L 作归并非递归排序 */
void MergeSort2(SqList *L)
{
    int* TR=(int*)malloc(L->length * sizeof(int));/* 申请额外空间 */
    int k=1;
    while(k<L->length)
    {
        MergePass(L->r,TR,k,L->length);
        k=2*k;/* 子序列长度加倍 */
        MergePass(TR,L->r,k,L->length);
        k=2*k;/* 子序列长度加倍 */
    }
}
```

```
/* ************************************ */

/* 快速排序********************************* */

/* 交换顺序表 L 中子表的记录，使枢轴记录到位，并返回其所在位置 */
/* 此时在它之前(后)的记录均不大(小)于它。 */
int Partition(SqList *L,int low,int high)
{
    int pivotkey;

    pivotkey=L->r[low]; /* 用子表的第一个记录作枢轴记录 */
    while(low<high) /*  从表的两端交替地向中间扫描 */
    {
        while(low<high&&L->r[high]>=pivotkey)
            high--;
        swap(L,low,high);/* 将比枢轴记录小的记录交换到低端 */
        while(low<high&&L->r[low]<=pivotkey)
            low++;
        swap(L,low,high);/* 将比枢轴记录大的记录交换到高端 */
    }
    return low; /* 返回枢轴所在位置 */
}

/* 对顺序表 L 中的子序列 L->r[low..high]作快速排序 */
void QSort(SqList *L,int low,int high)
{
    int pivot;
    if(low<high)
    {
        pivot=Partition(L,low,high); /*  将 L->r[low..high]一分为二，算出枢轴值 pivot */
        QSort(L,low,pivot-1);        /*  对低子表递归排序 */
        QSort(L,pivot+1,high);       /*  对高子表递归排序 */
    }
}

/* 对顺序表 L 作快速排序 */
void QuickSort(SqList *L)
{
    QSort(L,1,L->length);
}

/* ************************************ */
```

```
/* 改进后快速排序********************************* */

/* 快速排序优化算法 */
int Partition1(SqList *L,int low,int high)
{
    int pivotkey;

    int m = low + (high - low) / 2; /* 计算数组中间的元素的下标 */
    if (L->r[low]>L->r[high])
        swap(L,low,high);   /* 交换左端与右端数据，保证左端较小 */
    if (L->r[m]>L->r[high])
        swap(L,high,m);          /* 交换中间与右端数据，保证中间较小 */
    if (L->r[m]>L->r[low])
        swap(L,m,low);           /* 交换中间与左端数据，保证左端较小 */

    pivotkey=L->r[low]; /* 用子表的第一个记录作枢轴记录 */
    L->r[0]=pivotkey;   /* 将枢轴关键字备份到 L->r[0] */
    while(low<high) /*   从表的两端交替地向中间扫描 */
    {
        while(low<high&&L->r[high]>=pivotkey)
            high--;
        L->r[low]=L->r[high];
        while(low<high&&L->r[low]<=pivotkey)
            low++;
        L->r[high]=L->r[low];
    }
    L->r[low]=L->r[0];
    return low; /* 返回枢轴所在位置 */
}

void QSort1(SqList *L,int low,int high)
{
    int pivot;
    if((high-low)>MAX_LENGTH_INSERT_SORT)
    {
        while(low<high)
        {
            pivot=Partition1(L,low,high); /*   将 L->r[low..high]一分为二，算出枢轴值 pivot */
            QSort1(L,low,pivot-1);          /*   对低子表递归排序 */
            /* QSort(L,pivot+1,high);           /*   对高子表递归排序 */
            low=pivot+1;  /* 尾递归 */
        }
    }
    else
```

```
            InsertSort(L);
}

/*  对顺序表 L 作快速排序  */
void QuickSort1(SqList *L)
{
      QSort1(L,1,L->length);
}

/* ************************************** */
#define N 9
int main()
{
    int i;

    /* int d[N]={9,1,5,8,3,7,4,6,2}; */
    int d[N]={50,10,90,30,70,40,80,60,20};
    /* int d[N]={9,8,7,6,5,4,3,2,1}; */

    SqList l0,l1,l2,l3,l4,l5,l6,l7,l8,l9,l10;

    for(i=0;i<N;i++)
       l0.r[i+1]=d[i];
    l0.length=N;
    l1=l2=l3=l4=l5=l6=l7=l8=l9=l10=l0;
    printf("排序前:\n");
    print(l0);

    printf("初级冒泡排序:\n");
    BubbleSort0(&l0);
    print(l0);

    printf("冒泡排序:\n");
    BubbleSort(&l1);
    print(l1);

    printf("改进冒泡排序:\n");
    BubbleSort2(&l2);
    print(l2);

    printf("选择排序:\n");
    SelectSort(&l3);
    print(l3);
```

```
printf("直接插入排序:\n");
InsertSort(&l4);
print(l4);

printf("希尔排序:\n");
ShellSort(&l5);
print(l5);

printf("堆排序:\n");
HeapSort(&l6);
print(l6);

printf("归并排序（递归）:\n");
MergeSort(&l7);
print(l7);

printf("归并排序（非递归）:\n");
MergeSort2(&l8);
print(l8);

printf("快速排序:\n");
QuickSort(&l9);
print(l9);

printf("改进快速排序:\n");
QuickSort1(&l10);
print(l10);


 /*大数据排序*/
 /*
 srand(time(0));
 int Max=10000;
 int d[10000];
 int i;
 SqList l0;
 for(i=0;i<Max;i++)
       d[i]=rand()%Max+1;
 for(i=0;i<Max;i++)
       l0.r[i+1]=d[i];
 l0.length=Max;
 MergeSort(l0);
 print(l0);
 */
```

```
    return 0;
}
```