2——图的遍历·深度·广度

11——图·最小生成树·Prim·Kruskal

19——查找·线性表（顺序·前哨·折半·分块）

24——BF·KMP·next 值·教室

29——树与森林·存储结构·双亲表示法

31——树和森林·孩子表示法·网

35——树和森林·哈夫曼树`网

39——二叉树·遍历·全

47——串·堆存储结构·静态·标准

50——串·堆存储结构·动态·标准

54——数据结构·栈·顺序存储

56——数据结构·栈·链式存储

59——数据结构·队列·顺序存储

61——数据结构·队列·链式存储

63——串·顺序存储·标准

65——串·链式存储·标准

71——二叉树·应用举例·基本操作

74——广义表·基本操作

77——查找·哈希表·网

82——图·存储结构·邻接表·书

84——图·存储结构·邻接矩阵·书

87——图·有向·AOV 拓扑排序

图的遍历 ~ 深度、广度

```
//  图·遍历·深度与广度
//
//  邻接表:
//  顶点结点                        表结点
//  [  vertex  |  firstedge  ] -> [  adjvertex  |  next  ]
//  带权图边表结构:
//  [  adjvertex  |  info  |  next  ]

#include<stdio.h>
#include<stdlib.h>
#define MaxVertexNum 50
#define maxsize 100
#define Flase 0
#define True 1

typedef char VertexType;
typedef int InfoType;
typedef int Edgetype;

//邻接矩阵
typedef struct
{
    VertexType vertexs[MaxVertexNum];       //顶点向量
    Edgetype arcs[MaxVertexNum][MaxVertexNum];  //邻接矩阵
    int vertexnum,edgenum;                  //图的当前顶点数和边数
    //GraphType type;                     //图的种类标志
}MGraph;

//邻接表
typedef struct node {   //带权图表结点
    int adjvertex;   //邻接点域,一般是存放顶点对应的序号或在表头向量中的下标
    InfoType info;   //与边(或弧)相关的信息
    struct node * next;   //指向下一个邻接点的指针域
}EdgeNode;

typedef struct vnode {  //顶点结点
    VertexType vertex;    //顶点域
    EdgeNode * firstedge;   //边表头指针
}VertexNode;

typedef struct {
    VertexNode adjlist[MaxVertexNum];   //邻接表
```

```
    int vertexnum,edgenum;   //顶点数和边数
}ALGraph;  //ALGraph是以邻接表方式存储的图类型

int visited[MaxVertexNum];


//队列
typedef int datatype;

typedef struct{
    datatype data[maxsize];
    int front,rear;
}seqqueue,*pseqqueue;

pseqqueue init_seqqueue(void)
{//创建队列
    pseqqueue q;
    q=(pseqqueue)malloc(sizeof(seqqueue));
    if(q)
    {
        q->front=0;
        q->rear=0;
    }
    return q;
}

int empty_seqqueue(pseqqueue q)
{//队列判空
    if(q&&q->front==q->rear)
        return 1;
    else
        return 0;
}

int in_seqqueue(pseqqueue q,datatype x)
{//入队
    if((q->rear+1)%maxsize==q->front)
    {
        printf("队满");
        return -1;
    }
    else
    {
        q->rear=(q->rear+1)%maxsize;
```

```
        q->data[q->rear]=x;
        return 1;
    }
}

int out_seqqueue(pseqqueue q,datatype *x)
{//出队
    if(empty_seqqueue(q))
    {
        printf("队空");
        return -1;
    }
    else
    {
        q->front=(q->front+1)%maxsize;
        *x=q->data[q->front];
        return 1;
    }
}

MGraph init_mgraph(void)
{
    MGraph *G;
    G=(MGraph *)malloc(sizeof(MGraph));
    if(G)
        return *G;
    exit(0);
}
ALGraph init_alraph(void)
{
    ALGraph *G;
    G=(ALGraph *)malloc(sizeof(ALGraph));
    if(G)
        return *G;
    exit(0);
}

void creatgraph(MGraph *G)
{//建立 无向图G 的邻接矩阵存储
    int i,j,k;
    printf("邻接矩阵建立: \n");
    printf("  输入顶点数和边数: ");
    scanf("%d %d",&(G->vertexnum),&(G->edgenum));   //输入顶点数和边数
    printf("  输入%d顶点信息:",G->vertexnum);
```

```
    getchar();
    for(i=0;i<G->vertexnum;i++)      //输入顶点信息，建立顶点表v0,v1,v2,v3
        scanf("%c",&(G->vertexs[i]));
    for(i=0;i<G->vertexnum;i++)        //初始化邻接矩阵
        for(j=0;j<G->vertexnum;j++)
            G->arcs[i][j]=0;
    for(k=0;k<G->edgenum;k++)     //输入e条边，建立邻接矩阵
    {
        printf("  输入第%d个点的坐标(0<=i,j<=%d)：",k+1,G->vertexnum-1);
        scanf("%d %d",&i,&j);     //输入为 1 的点坐标 (i,j)
        G->arcs[i][j]=1;

        G->arcs[j][i]=1;    //若加入此行，则为无向图的邻接矩阵建立
    }
}

void CreateALGraph(ALGraph *G)
{//建立 无向图G 的邻接表存储
    int i,j,k;
    EdgeNode *p,*s;
    printf("邻接表建立：\n");
    printf("  输入顶点数和边数:");
    scanf("%d%d",&(G->vertexnum),&(G->edgenum));   //读人顶点数和边数
    printf("  输入%d个顶点信息:",G->vertexnum);
    getchar();
    for(i=0;i<G->vertexnum;i++)              //建立有n个顶点的顶点表
    {
        scanf("%c",&(G->adjlist[i].vertex));    //读人顶点信息 v0,v1,v2,v3
        G->adjlist[i].firstedge=NULL;     //顶点的边表头指针设为空
    }
    for(k=0;k<G->edgenum;k++)            //建立边表
    {
        printf("  输入第%d个边表连接(0<=i,j<=%d)：",k+1,G->vertexnum-1);
        scanf("%d%d",&i,&j);         //读入边<Vi,Vj>的顶点对应序号
        p=(EdgeNode*)malloc(sizeof(EdgeNode));    //生成新边表结点p*
        p->adjvertex=j;          //邻接点序号为j
        p->next=G->adjlist[i].firstedge;   //将新边表结点 p插人到顶点 Vi 的链表头部
        G->adjlist[i].firstedge=p;

        //加入以下代码，为创建无向图
        s=(EdgeNode*)malloc(sizeof(EdgeNode));
        s->adjvertex = i;       //邻接点序号为i
        s->next = G->adjlist[j].firstedge;
        G->adjlist[j].firstedge = s;
```

```
    }
}
void show_MGgraph(MGraph *G)
{//输出邻接矩阵
    int i,j;
    printf("创建的 无向图 邻接矩阵 为: \n");
    for(i=0;i<G->vertexnum;i++)
    {
        for(j=0;j<G->vertexnum;j++)
            printf(" %d",G->arcs[i][j]);
        printf("\n");
    }
}

void show_ALgraph(ALGraph *G)
{//输出邻接表
    int i;
    EdgeNode *p;
    printf("创建的 无向图 邻接表 为: \n");
    for(i=0;i<G->vertexnum;i++)
    {
        printf(" <%c> ",G->adjlist[i].vertex);
        p=G->adjlist[i].firstedge;
        while (p) {
            printf("->%c",G->adjlist[p->adjvertex].vertex);
            p=p->next;
        }
        printf("\n");
    }
}

void AL_DFS(ALGraph G,int v)
{//从第v个顶点出发遍历        <深度优先><邻接表>
    EdgeNode *p;
    int w;
    printf("%c",G.adjlist[v].vertex);
    visited[v]=True;
    for(p=G.adjlist[v].firstedge;p;p=p->next)
    {
        w=p->adjvertex;
        //printf("%d",w);
        if(!visited[w])
        {
```

```
            printf("->");
            AL_DFS(G,w);
        }//从初始顶点结点（v）开始，指针找 顶点结点指针 第一个所指的（1且False）元素，再将此元素变为顶点结点，开始找此 顶点结点指针 所指的第一个（1且Flase）元素……（若没有（1且Flase）元素则从初始顶点结点的下一个开始找）
    }
}
void AL_DFStraverse(ALGraph G)
{//遍历图G          <深度优先><邻接表>
    int v;
    printf("无向图 邻接表 深度优先遍历: ");
    for(v=0;v<G.vertexnum;v++)
        visited[v]=Flase;  //标志向量初始化
    for(v=0;v<G.vertexnum;v++)
        if(!visited[v])
            AL_DFS(G,v);
}

void AL_BFS(ALGraph G,int v)
{//从v出发按广度优先遍历图G，使用辅助队列q和访问标志数组visited      <广度优先><邻接表>
    EdgeNode *p;
    int u,w;
    pseqqueue q;       //定义一个队列
    q=init_seqqueue();      //置空队列
    printf("无向图 邻接表 广度优先遍历: ");
    printf("%c",G.adjlist[v].vertex);     //访问v
    visited[v]=True;   //把访问标志置True
    in_seqqueue(q, v);
    while(!empty_seqqueue(q))
    {
        out_seqqueue(q,&u);     //出队列
        for(p=G.adjlist[u].firstedge;p;p=p->next)
        {
            w=p->adjvertex;
            if(!visited[w])
            {
                printf("->%c",G.adjlist[w].vertex);
                visited[w]=True;
                in_seqqueue(q, w);    //u尚未访问的邻接顶点w入队列q
            }
        }//从初始顶点结点（v）开始，指针依次找（1且False）元素，访问并入队，访问完毕。再出对一个顶点元素，将其设为顶点结点，指针依次找（1且False）元素，访问并入队，访问完毕……（若顶点结点后面全True，从初始顶点结点（v）下一个开始）
    }
```

```
    printf("\n");
}

void MG_DFS(MGraph *G,int V)
{//深度优先遍历搜索,从V节点开始    <邻接矩阵>
    int w;
    printf("%c",G->vertexs[V]);
    visited[V]=True;  //标记为以访问
    for(w=0;w<G->vertexnum;w++)
    {
        if(!visited[w]&&(G->arcs[V][w]!=0||G->arcs[w][V]!=0))  //未访问 且 对应点存在为1
        {
            printf("->");
            MG_DFS(G,w);
        }//先初始行（V）循环从第一列找到最后一列，第一个为（1且Flase）的数输出，再把此数的 纵坐标数 变为 行数，再在此行循环从第一列找到最后一列……（如果循环到某一行中都访问过，则再从初始行（V）开始找）
    }

}
void MG_DFStraverse(MGraph *G)
{//深度优先遍历所有节点          <邻接矩阵>
    int i;
    for(i=0;i<G->vertexnum;i++)
        visited[i] = Flase;
    printf("无向图 邻接矩阵 深度优先遍历： ");
    for(i=0;i<G->vertexnum; i++)
    {//对与矩阵中所有元素都进行深度优先遍历搜索
        if(!visited[i])
            MG_DFS(G, i);
    }
    printf("\n");
}

void MG_BFS(MGraph G,int v)
{//以v为出发点。对G进行BFS       <广度优先><邻接矩阵>
    int i,j;
    pseqqueue q;
    q=init_seqqueue();
    printf("%c",G.vertexs[v]);     //访问v
    visited[v]=True;     //访问过
    in_seqqueue(q, v);
    while(!empty_seqqueue(q))
    {
        out_seqqueue(q,&i);     //出队列
```

```
        //printf("i=%d",i);  //i依此为0，1，2，3，4，。。。（行数变化）
        for(j=0;j<G.vertexnum;j++)  //依次搜索i行上循环j
        {
            if(G.arcs[i][j]==1&&!visited[j])   //邻接矩阵为1，且未访问
            {
                printf("->%c",G.vertexs[j]);     //访问
                visited[j]=True;
                in_seqqueue(q, j);    //j入队
            }//从初始行（V）开始，若为（1且Flase）的元素，依次输出并入队，一行结束了，出对一个元素当成行，再从那一行开始找，若为（1且Flase）的元素，依次输出并入队，一行结束，再出对下一个元素……
        }
    }
}

void MG_BFStraverse(MGraph G)
{//遍历图G          <广度优先><邻接矩阵>
    int v;
    for(v=0;v<G.vertexnum;v++)
        visited[v]=Flase;
    printf("无向图 邻接矩阵 广度优先遍历：");
    for(v=0;v<G.vertexnum;v++)
        if(!visited[v])  //v未访问过，从v开始BFS
            MG_BFS(G,v);
    printf("\n");
}

int main()
{

    MGraph MG=init_mgraph();
    creatgraph(&MG);
    show_MGgraph(&MG);

    MG_BFStraverse(MG);       //邻接矩阵，广度，无向
    MG_DFStraverse(&MG);      //邻接矩阵，深度，无向


    ALGraph AL=init_alraph();
    CreateALGraph(&AL);
    show_ALgraph(&AL);

    AL_BFS(AL, 0);         //邻接表，广度，无向
    AL_DFStraverse(AL);    //邻接表，深度，无向
```

```
    printf("\n");

    return 0;
}
/*
 邻接矩阵建立:
   输入顶点数和边数: 8 9
   输入8顶点信息:abcdefgh
   输入第1个点的坐标(0<=i,j<=8): 0 1
   输入第2个点的坐标(0<=i,j<=8): 0 2
   输入第3个点的坐标(0<=i,j<=8): 1 3
   输入第4个点的坐标(0<=i,j<=8): 1 4
   输入第5个点的坐标(0<=i,j<=8): 3 7
   输入第6个点的坐标(0<=i,j<=8): 4 7
   输入第7个点的坐标(0<=i,j<=8): 2 5
   输入第8个点的坐标(0<=i,j<=8): 2 6
   输入第9个点的坐标(0<=i,j<=8): 5 6
 创建的 无向图 邻接矩阵 为:
  0<b>c>0 0 0 0 0
  1 0 0<d>1 0 0 0
  1 0 0 0 0<f>1 0
  0 1 0 0 0 0 0<h>
  0 1 0 0 0 0 0 1
  0 0 1 0 0 0<g>0
  0 0 1 0 0 1 0 0
  0 0 0 1<e>0 0 0
 无向图 邻接矩阵 广度优先遍历: a->b->c->d->e->f->g->h   //解决
 无向图 邻接矩阵 深度优先遍历: a->b->d->h->e->c->f->g   //解决

 邻接表建立:
   输入顶点数和边数:8 9
   输入8个顶点信息:abcdefgh
   输入第1个边表连接(0<=i,j<=7): 0 1
   输入第2个边表连接(0<=i,j<=7): 0 2
   输入第3个边表连接(0<=i,j<=7): 1 3
   输入第4个边表连接(0<=i,j<=7): 1 4
   输入第5个边表连接(0<=i,j<=7): 3 7
   输入第6个边表连接(0<=i,j<=7): 4 7
   输入第7个边表连接(0<=i,j<=7): 2 5
   输入第8个边表连接(0<=i,j<=7): 2 6
   输入第9个边表连接(0<=i,j<=7): 5 6
 创建的 无向图 邻接表 为:
  <a> ->c->b->∧
  <b> ->e->d->a->∧
```

```
 <c> ->g->f->a->∧
 <d> ->h->b->∧
 <e> ->h->b->∧
 <f> ->g->c->∧
 <g> ->f->c->∧
 <h> ->e->d->∧
无向图 邻接表 广度优先遍历：a->c->b->g->f->e->d->h   //解决
无向图 邻接表 深度优先遍历：a->c->g->f->b->e->h->d   //解决
*/
```

图·最小生成树·Prim·Kruskal

```
//  图·最小生成树·Prim·Kruskal
//

#include <stdio.h>
#include <stdlib.h>
#define INFINITY 10  //定义一个权值的最大值
#define MaxVertexNum 30  //最大顶点数
#define MaxEdge 100

typedef char VertexType;
typedef int Edgetype;
typedef int WeightType;

typedef struct
{
    VertexType vertexs[MaxVertexNum];       //顶点向量
    Edgetype arcs[MaxVertexNum][MaxVertexNum];  //邻接矩阵
    int vertexnum,edgenum;      //图的当前顶点数和边数
}MGraph;

//Prim
typedef struct
{
    char adjvertex;  //某顶点与已构造好的部分生成树的顶点之间权值最小的顶点
    int lowcost;  //某顶点与已构造好的部分生成树的顶点之间的最小权值
}ClosEdge[MaxVertexNum];  //用Prim求最小生成树时的辅助数组

//Kruskal
typedef struct
{
```

```
    int initial;
    int end;
    WeightType weight;   //边的权值
}ENode;
typedef struct
{
    int vertexnum,edgenum;  //顶点个数,边的个数
    VertexType vertexs[MaxVertexNum];  //顶点信息
    ENode edges[MaxVertexNum];  //边的信息
}ELGraph;   //注意：此图的存储结构与前面介绍的几种不一样

MGraph init_mgraph(void)
{
    MGraph *G;
    G=(MGraph *)malloc(sizeof(MGraph));
    if(G)
        return *G;
    exit(0);
}

void creatgraph(MGraph *G)
{//建立 无向图G 的邻接矩阵存储
    int i,j,k,w;
    printf("邻接矩阵建立：\n");
    printf("  输入顶点数和边数：");
    scanf("%d %d",&(G->vertexnum),&(G->edgenum));   //输入顶点数和边数
    printf("  输入%d个顶点信息:",G->vertexnum);
    getchar();
    for(i=0;i<G->vertexnum;i++)     //输入顶点信息，建立顶点表v0,v1,v2,v3
        scanf("%c",&(G->vertexs[i]));
    for(i=0;i<G->vertexnum;i++)       //初始化邻接矩阵
        for(j=0;j<G->vertexnum;j++)
            G->arcs[i][j]=INFINITY;
    printf("  输入%d个点的坐标(0<=i,j<=10) 和 权值(w<10)：\n",G->edgenum);
    for(k=0;k<G->edgenum;k++)     //输入e条边，建立邻接矩阵
    {
        scanf("%d %d %d",&i,&j,&w);     //输入为 1 的点坐标 (i,j)
        G->arcs[i][j]=w;
        G->arcs[j][i]=w;    //若加入此行，则为无向图的邻接矩阵建立
    }
}
void show_MGgraph(MGraph *G)
{//输出邻接矩阵
    int i,j;
```

```
    printf("创建的 无向图 邻接矩阵 为: \n");
    for(i=0;i<G->vertexnum;i++)
    {
        for(j=0;j<G->vertexnum;j++)
            printf("%3d",G->arcs[i][j]);
        printf("\n");
    }
}

ELGraph init_elgraph(void)
{
    ELGraph *EG;
    EG=(ELGraph *)malloc(sizeof(ELGraph));
    if(EG)
        return *EG;
    exit(0);
}

void createlgraph(ELGraph *EG)
{//建立 连通网G 的存储
    int i,k;
    printf("建立连通网: \n");
    printf("  输入顶点数和边数: ");
    scanf("%d %d",&(EG->vertexnum),&(EG->edgenum));   //输入顶点数和边数
    printf("  输入连通网的%d个顶点信息: ",EG->vertexnum);
    getchar();
    for(i=0;i<EG->vertexnum;i++)      //输入顶点信息
        scanf("%c",&(EG->vertexs[i]));
    printf("  输入%d个点的坐标(0<=i,j<=10) 和 权值(w<10): \n",EG->edgenum);
    for(k=0;k<EG->edgenum;k++)     //输入e条边
    {
        scanf("%d%d%d",&EG->edges[k].initial,&EG->edges[k].end,&EG->edges[k].weight);
    }
}
void show_ELGgraph(ELGraph *EG)
{//输出连通网
    int j;
    printf("创建的 连通图 为: \n");
    for(j=0;j<EG->edgenum;j++)
        printf("|%d
<%d> %d|\n",EG->edges[j].initial,EG->edges[j].weight,EG->edges[j].end);
}

void mintree_Prim(MGraph *G,int u,ClosEdge closedge)
```

```
{//从第u个顶点出发构造最小生成树，最小生成树顶点信息存放在数组closedge中
    int i,j,k=0,w;
    for(i=0;i<G->vertexnum;i++)  //辅助数组初始化
    {
        if(i!=u)
        {
            closedge[i].adjvertex=G->vertexs[u];
            closedge[i].lowcost=G->arcs[u][i];
            //printf("[%c %d]\n",closedge[i].adjvertex,closedge[i].lowcost);
        }
    }

    closedge[u].lowcost=0;  //初始，U={u}
    for(i=0;i<G->vertexnum;i++)  //选择其余的G->vertexnum-1个顶点
    {
        w=INFINITY;
        for(j=0;j<G->vertexnum;j++)
            if(closedge[j].lowcost!=0 && closedge[j].lowcost<w)
            {
                w=closedge[j].lowcost;
                k=j;  //求出生成树的下一个顶点k
                //printf("<%d %d>\n",w,k);
            }
        closedge[k].lowcost=0;  //第k顶点并入U集
        for(j=0;j<G->vertexnum;j++)  //新顶点并入U后，修改辅助数组
            if(G->arcs[k][j]<closedge[j].lowcost)
            {
                closedge[j].adjvertex=G->vertexs[k];
                closedge[j].lowcost=G->arcs[k][j];
                //printf("|[%c %d]|\n",closedge[j].adjvertex,closedge[j].lowcost);
            }
        /*
        //检验
        printf("closedge[j].lowcost:");
        for(j=0;j<G->vertexnum;j++)
            printf(" %d",closedge[j].lowcost);
        printf("\n");
        printf("closedge[j].adjvertex:");
        for(j=0;j<G->vertexnum;j++)
            printf(" %c",closedge[j].adjvertex);
        printf("\n");
         */
    }
    printf("\n打印最小生成树的各条边:\n");
```

```
    for(i=0;i<G->vertexnum;i++)  //打印最小生成树的各条边
    {
        if(i!=u)
        {
            for(j=0;j<G->vertexnum;j++)
                if(closedge[i].adjvertex==G->vertexs[j])
                    break;

printf("%c<->%c,weight:%d\n",closedge[i].adjvertex,G->vertexs[i],G->arcs[i][j]);
        }
    }

}

ELGraph sort(ELGraph G, int n)
{//对图G的边表按 权值weight 从小到大排序
    int i,j;
    for(i=1;i<=n;i++)
    {
        for(j=0;j<n-i;j++)
        {
            if(G.edges[j].weight>G.edges[j+1].weight)
            {
                ENode t = G.edges[j];
                G.edges[j] = G.edges[j+1];
                G.edges[j+1] = t;
            }
        }
    }
    printf("排序后 的 连通图 为: \n");
    for(j=0;j<n;j++)
        printf("|%d <%d> %d|\n",G.edges[j].initial,G.edges[j].weight,G.edges[j].end);
    return G;
}

void mintree_Kruskal(ELGraph EG,ENode TE[])
{//用 Kruskal算法构成图 G的最小生成树,最小生成树存放在 TE[]中
    int i,j,k;
    int s1,s2;
    int f[MaxVertexNum];
    for(i=0;i<EG.vertexnum;i++)  //初始化f数组
        f[i]=i;
    EG=sort(EG,EG.edgenum);  //对图G的边表按 权值weight 从小到大排序
    j=0;
```

```
    k=0;
    while(k<EG.vertexnum-1)  //选n-1条边
    {
        //printf("|%d %d|\n",EG.edges[j].initial,EG.edges[j].end);
        s1=f[EG.edges[j].initial];
        s2=f[EG.edges[j].end];
        //printf("[%d %d %d]",EG.edges[j].initial,EG.edges[j].end,k);
        if(s1!=s2)   //产生一条最小边
        {
            TE[k].initial=EG.edges[j].initial;
            TE[k].end=EG.edges[j].end;
            TE[k].weight=EG.edges[j].weight;
            k++;
            for(i=0;i<EG.vertexnum;i++)
                if(f[i]==s2)
                    f[i]=s1;  //修改连通的编号
        }
        /*
        for(i=0;i<EG.vertexnum;i++)
            printf("%d ",f[i]);
        printf("\n");
         */
        j++;
    }
    for(i=0; i<EG.vertexnum-1; i++)

printf("%c<->%c,weight:%d\n",EG.vertexs[TE[i].initial],EG.vertexs[TE[i].end],TE[i].weigh
t);
}

int main()
{
    MGraph MG=init_mgraph();     //初始邻接矩阵
    ClosEdge closedge;
    creatgraph(&MG);          //创建邻接矩阵
    show_MGgraph(&MG);           //打印邻接矩阵
    mintree_Prim(&MG, 0, closedge);   //Prim求最小生成树
    printf("\n");


    ELGraph EG=init_elgraph();  //初始化连通网
    ENode TE[MaxVertexNum];
    createlgraph(&EG);       //创建连通网
    show_ELGgraph(&EG);        //打印连通表
```

```
    mintree_Kruskal(EG, TE);   //Lruskal求最小生成树
}

/*
 测试数据:
5 7
0 1 3
0 2 6
0 3 4
1 2 7
2 3 2
2 4 8
3 4 9

6 10
1 2 6
1 3 1
1 4 5
2 3 5
2 5 3
3 4 5
3 5 6
3 6 4
4 6 2
5 6 6

4 4
0 1 3
0 2 4
1 2 2
3 0 6
*/

/*
 邻接矩阵建立:
   输入顶点数和边数: 5 7
   输入5个顶点信息:abcde  输入5个点的坐标(0<=i,j<=10) 和 权值(w<10):
 0 1 3
 0 2 6
 0 3 4
 1 2 7
 2 3 2
 2 4 8
 3 4 9
```

```
创建的 无向图 邻接矩阵 为:
 10  3  6  4 10
  3 10  7 10 10
  6  7 10  2  8
  4 10  2 10  9
 10 10  8  9 10
a<->b,weight:3
d<->c,weight:2
a<->d,weight:4
c<->e,weight:8

建立连通网:
  输入顶点数和边数: 5 7
  输入连通网的5个顶点信息: 01234
  输入7个点的坐标(0<=i,j<=10) 和 权值(w<10):
 0 1 3
 0 2 6
 0 3 4
 1 2 7
 2 3 2
 2 4 8
 3 4 9
创建的 连通图 为:
|0 <3> 1|
|0 <6> 2|
|0 <4> 3|
|1 <7> 2|
|2 <2> 3|
|2 <8> 4|
|3 <9> 4|
排序后 的 连通图 为:
|2 <2> 3|
|0 <3> 1|
|0 <4> 3|
|0 <6> 2|
|1 <7> 2|
|2 <8> 4|
|3 <9> 4|
2<->3,weight:2
0<->1,weight:3
0<->3,weight:4
2<->4,weight:8

建立连通网:
```

```
输入顶点数和边数：5 7
输入连通网的5个顶点信息：abcde
输入7个点的坐标(0<=i,j<=10) 和 权值(w<10)：
0 1 3
0 2 6
0 3 4
1 2 7
2 3 2
2 4 8
3 4 9
创建的 连通图 为：
|0 <3> 1|
|0 <6> 2|
|0 <4> 3|
|1 <7> 2|
|2 <2> 3|
|2 <8> 4|
|3 <9> 4|
排序后 的 连通图 为：
|2 <2> 3|
|0 <3> 1|
|0 <4> 3|
|0 <6> 2|
|1 <7> 2|
|2 <8> 4|
|3 <9> 4|
c<->d,weight:2
a<->b,weight:3
a<->d,weight:4
c<->e,weight:8
*/
```

查找·线性表（顺序·前哨·折半·分块）

```
//
//  main.c
//  查找·线性表（顺序·前哨·折半·分块）
//

#include <stdio.h>
#include <stdlib.h>
#define maxsize 100    //查找表最大长度
typedef int  KeyType;  //整型
```

```
//顺序，前哨，折半
typedef struct
{
    KeyType key;
}DataType;

typedef struct
{
    DataType r[maxsize];  //数据元素存储空间
    int length;  //表的长度
}Sqlist;

//分块
#define MAXL 100     //顺序表的最大长度
#define MAXI 20      //索引表的最大长度
typedef int KeyType;

typedef struct
{
    KeyType key;  //顺序表数据
}NodeType;
typedef NodeType SeqList[MAXL];  //顺序表
typedef struct
{
    KeyType key;  //索引表最大关键字
    int link;  //每块的起始下标
}IdxType;
typedef IdxType IDX[MAXI];   //索引表

void init_Idxtype(SeqList R,IDX I)  //分块处理
{
    int a[]={8,14,6,9,10,22,34,18,19,31,40,38,54,66,46,71,78,68,80,85,100,94,88,96,87};
    for(int i=0;i<25;i++)    //建立顺序表
        R[i].key=a[i];
    //  8   <14>  6   9    10      0  -> 4
    //  22   34   18  19  <31>     5  -> 9
    //  40   38   54 <66>  46      10 -> 14
    //  71   78   68  80  <85>     15 -> 19
    // <100> 94   88  96   87      20 -> 24
    I[0].key=14;I[0].link=0;  //开始分块
    I[1].key=34;I[1].link=5;
    I[2].key=66;I[2].link=10;
    I[3].key=85;I[3].link=15;
    I[4].key=100;I[4].link=20;
```

```
}

Sqlist init_strlist(void)
{//创建查找表
    Sqlist *s;
    s=(Sqlist *)malloc(sizeof(Sqlist));
    printf("输入此线性表长度（前哨长度+1）：");
    scanf("%d",&s->length);
    printf("输入此线性表元素：\n");
    for(int i=0;i<s->length;i++)
        scanf("%d",&s->r[i].key);
    return *s;
}

//顺序查找算法
int SeqSearch(Sqlist s,KeyType k)
{//在表s中顺序查找关键字k，若查找成功，则函数值为该元素在表中的位置，若查找失败，返回-1
    int i;
    for(i=0;i<s.length;i++)
        if(s.r[i].key==k)
            return i;   //查找成功
    return -1;   //查找失败
}

//带前哨站的顺序查找改进算法
int SeqSearch_gai(Sqlist s,KeyType k)
{
    s.length+=1; // 让s.length长度加1，当前哨
    int i=0,n;
    n= s.length;
    s.r[n].key=k;   //设置前哨站，要多开辟一个存储空间
    while(s.r[i].key!=k)  //从表首开始向后扫描
        i++;
    return i;
}

//折半查找算法
int BinSearch(Sqlist s , KeyType k)
{//在表s中用折半查找法查找关键字k，若查找成功，则函数值为该元素在表中的位置，若查找失败，返回-1
    int low,mid,high;
    low = 0;
    high = s.length-1;
    while(low<=high)
    {
```

```
        mid=(low+high)/2;       //取区间中点
        if (s.r[mid].key==k)
            return mid;     //查找成功
        else if (s.r[mid].key>k)
            high=mid-1;       //在左区间中查找
        else
            low=mid+1;        //在右区间中查找
     }
    return -1;   //查找失败
}

//折半查找—递归算法
int BinSearch1(Sqlist s, KeyType k, int low, int high)
{//在表s中用折半查找法查找关键字k，若查找成功，则函数值为该元素在表中的位置，若查找失败，返回-1
    int mid;
    while(low <= high)
    {
        mid=(low+high)/2;       //取区间中点
        if(s.r[mid].key==k)
            return mid;     //查找成功
        else if(s.r[mid].key>k)
            return BinSearch1(s, k, low, mid-1);  //在左区间中查找
        else
            return BinSearch1(s, k, mid+1, high);  //在右区间中查找
     }
    return -1;   //查找失败
}

int IdxSearch(IDX I,int m,SeqList R,int n,KeyType k)
{//分块查找算法， m 为分块的数量，n 为顺序表数据的个数
    int low=0,high=m-1,mid,i;
    int b=n/m;     //b为每块数据的个数

    //在索引表中进行二分查找,找到存放在的对应分块
    printf("1.二分查找key在哪一分块：\n");
    while (low<=high)
    {
        mid=(low+high)/2;
        //printf("在 %d -> %d 分块中比较元素R[%d]:%d\n",low+1,high+1,mid,R[mid-1].key);
        if (I[mid].key>=k)
            high=mid-1;
        else
            low=mid+1;
    }
```

```
    if (low<m)//在索引表中查找成功后,再在线性表中进行顺序查找
    {
        printf("   该 key=%d 在索引表的第 %d 分块中\n",k,low+1);
        i=I[low].link-1;
        printf("2.顺序查找key在分块的第几个: \n");
        printf("   该分块的数据为: ");
        while (i<=I[low].link+b-1 && R[i].key!=k)
        {
            i++;
            printf("%d ",R[i].key);
        }
        printf("\n   该 key=%d 在顺序表的第 %d 位\n",k,i);
        if (i<=I[low].link+b-1)
            return i;
        else
            return -1;
    }
    return -1;
}

int main(int argc, const char * argv[])
{

    Sqlist str1=init_strlist();  //正常创建
    //Sqlist str2=init_strlist();  //多创建一个空间

    int key;
    printf("输入你想查找的key: ");
    scanf("%d",&key);

    printf("顺序查找 %d 在第 %d 位\n",key,SeqSearch(str1, key)+1);     //无前哨，顺序查找
    printf("前哨顺序查找 %d 在第 %d 位\n",key,SeqSearch_gai(str1, key)+1);    //有前哨

    printf("循环--折半查找 %d 在第 %d 位\n",key,BinSearch(str1, key)+1);    //循环--折半查找
    printf("递归--折半查找 %d 在第 %d 位\n",key,BinSearch1(str1, key, 0, str1.length)+1);
//递归--折半查找


    printf("\n分块查找:\n");

    KeyType k=46;  //分块查找要查找的元素k
    SeqList R;     //定义顺序表
    IDX I;         //定义索引表
    int i;
```

```
    init_Idxtype(R,I);   //初始化索引表，对顺序表进行分块处理
    if((i=IdxSearch(I,5,R,25,k))!=-1)
        printf("\n分块查找 key=%d 在第 %d 位\n",k,i);
    else
        printf("\n分块查找 key=%d 不在表中\n",k);
    printf("\n");


}
/*

 输入此线性表长度（前哨长度+1）：5
 输入此线性表元素:
 1
 2
 3
 4
 5
 输入你想查找的key：4
 顺序查找 4 在第 4 位
 前哨顺序查找 4 在第 4 位
 循环--折半查找 4 在第 4 位
 递归--折半查找 4 在第 4 位

 分块查找:
 1.二分查找key在哪一分块:
    该 key=46 在索引表的第 3 分块中
 2.顺序查找key在分块的第几个:
    该分块的数据为：40 38 54 66 46
    该 key=46 在顺序表的第 14 位

 分块查找 key=46 在第 14 位

 */
```

BF·KMP·next 值·教室

```
//
//  main.c
//  BF·KMP·next值·教室
//

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

```
int strindex_BF(char *s,char *t)
{//从串s的第1个字符开始找首次与串t相等的子串    //简单模式匹配
    int i=0,j=0;
    int count=0;
    int ls=(int)strlen(s),lt=(int)strlen(t);
    while(i<ls&&j<lt)
    {
        if(s[i]==t[j])
        {
            //printf("[%c %c]",s[i],t[j]);   //检验行
            i++;
            j++;
        }
        else
        {
            i=i-j+1;
            j=0;
            //printf("<%d|%c>",i,s[i]);   //检验行
        }
        count++;
    }
    printf("BF的比较次数为: %d      ",count+1);//最后一位匹配成功也要算上
    if(j>=lt)
        return i-lt;
    else
        return -1;
}

/*
int strindex_BF(char *s,char *t)
{//从串s的第1个字符开始找首次与串t相等的子串    //简单模式匹配
    int i=1,j=1;
    while(i<=s[0]&&j<=t[0])
        if(s[i]==t[j])
        {
            i++;
            j++;
        }
        else
        {
            i=i-j+2;
            j=1;
        }
    if(j>t[0])
```

```
        return i-t[0];
    else
        return -1;
}
*/
void web_getnext1(char *t,int next[])
{//第一第二位为-1，0。之后找最长相同子串，相同位数加1
    int j=0,k=-1;
    next[0]=-1;
    int lt=(int)strlen(t);
    while(j<lt-1)
    {
        if(k ==-1||t[j]==t[k])
        {
            j++;
            k++;
            /*
            if(t[j]==t[k])//当两个字符相同时，就跳过
                next[j] = next[k];
            else
                */
            next[j] = k;
        }
        else
            k = next[k];
    }
    printf("next(-1,0开始)值为：");
    for(int i=0;i<lt;i++)   //检验行
        printf("%d ",next[i]);
}
int web_KMP(char *s,char *t)
{//从串s的第1个字符开始找首次与串t相等的子串
    int next[10],i=0,j=0;
    int count=0;
    web_getnext1(t,next);
    int ls=(int)strlen(s),lt=(int)strlen(t);
    while(i<ls&&j<lt)
    {
        if(j==-1 || s[i]==t[j])
        {
            i++;
            j++;
        }
        else
```

```
            j=next[j];              //j回退
        count++;
    }
    printf("  WEB_KMP的比较次数为：%d  ",count);
    if(j>=lt)
        return (i-lt);         //匹配成功，返回子串的位置
    else
        return (-1);                //没找到
}

void getnext(char *t,int next[])
{//求模式t的next值并存入next数组中，字符串长度保留在t[0]
    //第一第二位为-1，0。之后找最长相同子串，相同位数加1
    int i=0,j=-1;
    int lt=(int)strlen(t);
    next[0]=-1;
    while(i<lt-1)
        if(j==-1||t[i]==t[j])
        {
            ++i;
            ++j;
            next[i]=j;//回溯
        }
        else
            j=next[j];
    printf("next(-1,0开始)值为：");
    for(int i=0;i<lt;i++)   //检验行
        printf("%d ",next[i]);
}

int strindex_KMP(char *s,char *t,int pos,int *next)
{//从串s的第pos个字符开始找首次与串t相等的子串    //KPM模式匹配，找匹配最长子串，平移
    int i=pos-1,j=0;
    //char next[10];
    int ls=(int)strlen(s),lt=(int)strlen(t);
    while(i<ls&&j<lt)
        if(j==-1||s[i]==t[j])
        {
            i++;
            j++;
        }
        else
            j=next[j];//回溯
    if(j>=lt)
```

```
        return i-j;//匹配成功，返回存储位置
    else
        return -1;
}

/*
int strindex_KMP(char *s,char *t,int pos)
{//从串s的第pos个字符开始找首次与串t相等的子串    //KPM模式匹配，找匹配最长子串，平移
    int i=pos,j=1;
    char next[10];
    while(i<=s[0]&&j<=t[0])
        if(j==0||s[i]==t[j])
        {
            i++;
            j++;
        }
        else
            j=next[j];//回溯
    if(j>t[0])
        return i-t[0];//匹配成功，返回存储位置
    else
        return -1;
}
*/

void getnext_rec(char *t,int next[],int l)
{//递归算法求next值，t为模式串，l为模式串长度=t[0]
    if(l==1)//l=1，递归出口
    {
        next[1]=0;//递归求next[l-1],为next[l]作准备
        return;
    }
    getnext_rec(t,next,l-1);
    int k=next[l-1];
    while(1)//直到next[l]计算完毕
    {
        if(t[k]==t[l-1])
        {
            next[l]=k+1;
            return;
        }
        k=next[k];//回溯
        if(k==0)//不存在最大相等的前缀和后缀子串，next[1]=1
        {
```

```
            next[l]=1;
            return;
        }
    }
}

int main() {
    char s[15]="001000100001010";
    char s1[5]="00101";

    int BF,KMP,WEB_KMP;
    BF=strindex_BF(s,s1);
    printf("BF匹配：第%d位 \n",BF+1);
    WEB_KMP=web_KMP(s,s1);
    printf("WEB_KMP匹配：第%d位\n",WEB_KMP+1);

    int next[10];
    getnext(s1, next);
    KMP=strindex_KMP(s, s1, 2, next);
    printf("  KMP匹配：第%d位\n",KMP+1);

    return 0;
}
/*
 BF的比较次数为：31       BF匹配：第10位
 next(-1,0开始)值为：-1 0 1 0 1   WEB_KMP的比较次数为：19  WEB_KMP匹配：第10位
 next(-1,0开始)值为：-1 0 1 0 1   KMP匹配：第10位
 */
```

树与森林·存储结构·双亲表示法

```
//
//  main.c
//  树与森林·存储结构·双亲表示法
//

#include <stdio.h>
#include <stdlib.h>
#define MaxNodeNum  100

typedef char datatype;

typedef struct
{
```

```
    datatype data;   //结点的值
    int parent;      //双亲的下标
}Parentlist;

typedef struct
{
    Parentlist elem[MaxNodeNum];
    int n;    //树中当前的结点数目
}ParentTree;

ParentTree InitPNode(ParentTree tree)
{
    int i, j;
    char ch;
    printf("请输出节点个数:\n");
    scanf("%d", &(tree.n));
    printf("请输入结点的值其双亲位于数组中的位置下标:\n");
    for (i = 0; i < tree.n; i++)
    {
        getchar();  //”吃掉“ 按下的 回车键（scanf会识别回车键，所以要消除回车键）（键盘缓冲区的知识
点）
        scanf("%c %d", &ch, &j);
        tree.elem[i].data = ch;
        tree.elem[i].parent = j;
    }
    return tree;
}
void FindParent(ParentTree tree)
{
    char a;
    int isfind = 0;
    printf("请输入要查询的结点值:\n");
    getchar();
    scanf("%c", &a);
    for (int i = 0; i < tree.n; i++)
    {
        if (tree.elem[i].data == a)
        {
            isfind = 1;
            int ad = tree.elem[i].parent;
            printf("%c的父节点为 %c,存储位置下标为 %d\n", a, tree.elem[ad].data, ad);
            break;
        }
    }
```

```
    if (isfind == 0)
        printf("树中无此节点\n");
}
int main()
{
    ParentTree *tree;
    tree=(ParentTree *)malloc(sizeof(ParentTree));
    for (int i = 0; i < MaxNodeNum; i++)
    {
        tree->elem[i].data = ' ';
        tree->elem[i].parent = 0;
    }
    *tree=InitPNode(*tree);
    FindParent(*tree);
    return 0;
}

/*
请输出节点个数:
10
请输入结点的值其双亲位于数组中的位置下标:
R -1
A 0
B 0
C 0
D 1
E 1
F 3
G 6
H 6
K 6
请输入要查询的结点值:
C
C的父节点为 R,存储位置下标为 0
*/
```

树和森林·孩子表示法·网

```
//  main.c
//  树和森林·孩子表示法·网
//

#include<stdio.h>
#include<stdlib.h>
#define MAX_SIZE 20
```

```c
#define TElemType char

typedef struct CTNode
{
    int child;    //链表中每个结点存储的不是数据本身，而是数据在数组中存储的位置下标
    struct CTNode * next;
}ChildPtr;

typedef struct
{
    TElemType data;    //结点的数据类型
    ChildPtr * firstchild;    //孩子链表的头指针
}CTBox;

typedef struct
{
    CTBox nodes[MAX_SIZE];   //存储结点的数组
    int n, r;   //结点数量和树根的位置
}CTree;

//孩子表示法存储普通树
CTree initTree(CTree tree)
{
    printf("输入节点数量: \n");
    scanf("%d", &(tree.n));
    for (int i = 0; i < tree.n; i++)
    {
        printf("输入第 %d 个节点的值: \n", i + 1);
        getchar();
        scanf("%c", &(tree.nodes[i].data));
        tree.nodes[i].firstchild = (ChildPtr*)malloc(sizeof(ChildPtr));
        tree.nodes[i].firstchild->next = NULL;

        printf("输入节点 %c 的孩子节点数量: \n", tree.nodes[i].data);
        int Num;
        scanf("%d", &Num);
        if (Num != 0)
        {
            ChildPtr * p = tree.nodes[i].firstchild;
            for (int j = 0; j < Num; j++)
            {
                ChildPtr * newEle = (ChildPtr*)malloc(sizeof(ChildPtr));
                newEle->next = NULL;
                printf("输入第 %d 个孩子节点在顺序表中的位置", j + 1);
```

```
            scanf("%d", &(newEle->child));
            p->next = newEle;
            p = p->next;
        }
    }
}
    return tree;
}

void findKids(CTree tree, char a)
{
    int hasKids = 0;
    for (int i = 0; i < tree.n; i++)
    {
        if (tree.nodes[i].data == a)
        {
            ChildPtr * p = tree.nodes[i].firstchild->next;
            while (p)
            {
                hasKids = 1;
                printf("%c ", tree.nodes[p->child].data);
                p = p->next;
            }
            break;
        }
    }
    if (hasKids == 0)
        printf("此节点为叶子节点");
}

int main()
{
    CTree *tree;
    tree=(CTree *)malloc(sizeof(CTree));
    for (int i = 0; i < MAX_SIZE; i++)
        tree->nodes[i].firstchild = NULL;
    *tree = initTree(*tree);
    //默认数根节点位于数组notes[0]处
    tree->r = 0;
    printf("找出节点 F 的所有孩子节点：");
    findKids(*tree,'F');
    return 0;
}
/*
```

```
输入节点数量:
10
输入第 1 个节点的值:
R
输入节点 R 的孩子节点数量:
3
输入第 1 个孩子节点在顺序表中的位置1
输入第 2 个孩子节点在顺序表中的位置2
输入第 3 个孩子节点在顺序表中的位置3
输入第 2 个节点的值:
A
输入节点 A 的孩子节点数量:
2
输入第 1 个孩子节点在顺序表中的位置4
输入第 2 个孩子节点在顺序表中的位置5
输入第 3 个节点的值:
B
输入节点 B 的孩子节点数量:
0
输入第 4 个节点的值:
C
输入节点 C 的孩子节点数量:
1
输入第 1 个孩子节点在顺序表中的位置6
输入第 5 个节点的值:
D
输入节点 D 的孩子节点数量:
0
输入第 6 个节点的值:
E
输入节点 E 的孩子节点数量:
0
输入第 7 个节点的值:
F
输入节点 F 的孩子节点数量:
3
输入第 1 个孩子节点在顺序表中的位置7
输入第 2 个孩子节点在顺序表中的位置8
输入第 3 个孩子节点在顺序表中的位置9
输入第 8 个节点的值:
G
输入节点 G 的孩子节点数量:
0
输入第 9 个节点的值:
```

```
H
输入节点 H 的孩子节点数量:
0
输入第 10 个节点的值:
K
输入节点 K 的孩子节点数量:
0
找出节点 F 的所有孩子节点: G H K
*/
```

树和森林·哈夫曼树`网

```
//
//  main.c
//  树和森林·哈夫曼树

#include <stdio.h>

#include <stdlib.h>
#define N 20

typedef int DataType;

typedef struct{//哈夫曼树结点类型定义
    char ch;
    DataType weight;
    int lchild,rchild,parent;
}Htnode;

//Htnode huffTree[];//采用静态链表存储哈夫曼树

typedef struct{//叶编码类型
    char *code;
    char leaf;
    int length;
}CodeType;

//CodeType cd[];//一维数组存储编码

void selectsort(Htnode huftree[],int n,int *s1,int *s2)
{
    int i,min1,min2;//两个最小数
    min1=huftree[0].weight;
```

```
    min2=huftree[0].weight;
    *s1=0;
    for(i=1;i<=n;i++)
    {
        if(huftree[i].parent==-1&&huftree[i].weight<min1)
        {//如果节点未被构建树,并且小于最小值,则更新最小值*/
            min1=huftree[i].weight;
            *s1=i;
        }
        else//为下边求另一个最小值赋初值*/
        {
            min2=huftree[i].weight;
            *s2=i;
        }
    }/*end for*/
    for(i=1;i<=n;i++)
    {
        if(huftree[i].parent==-1&&huftree[i].weight<min2&&huftree[i].weight>=
min1&&*s1!=i)
        /*如果节点未被构建树,并且小于最小值,并且大于第一个最小值,*/
        /*并且在数组中的下标不等于第一个最小值,则更新最小值*/
        {
            min2 = huftree[i].weight;
            *s2 = i;
        }
    }/*end for*/
}

void Hufcoding(Htnode huftree[],CodeType cd[],int w[],int n)
{//哈夫曼树存放在静态链表huftree中,w存放结点权重,n是叶子个数,最后的编码放在cd[]
    int i,k,s1,s2,m,f,c,sum;
    char temp[N];   //暂存叶子编码字符串，最后需要转置
    m=2*n-1;     //计算哈夫曼树的结点总数
    for(i=1;i<=n;i++)  //初始化每个叶子结点自成一 棵树
    {
        huftree[i].weight=w[i-1];
        huftree[i].lchild=huftree[i].rchild=huftree[i].parent=-1;
        //huftree[i].ch=getch();
    }
    for(i=n+1;i<=m;i++)  //初始化非叶子结点
    {
        huftree[i].weight=-1;
        huftree[i].lchild=huftree[i].rchild=huftree[i].parent=-1;
    }
```

```
    for(i=1;i<=n-1;i++)  //生成n-1个非叶子结点的循环
    {
        selectsort(huftree,n+i-1,&s1,&s2);
        //对数组 huftree[1..n+i-1]中无双亲的结点权值进行排序，s1,s2将是无双亲且权重最小的两个结点下
标
        sum=huftree[s1].weight+huftree[s2].weight;//求和,构造父节点
        huftree[n+i].weight=sum;
        huftree[s1].parent=huftree[s2].parent=n+i;//最小的两个节点的父节点的数组的下标
        huftree[n+i].lchild=s1;//父节点的左孩子下标
        huftree[n+i].rchild=s2;//父节点的右孩子下标
    }
    for(i=1;i<=n;i++)     //开始求每个叶子结点的编码
    {
        c=0;
        for (k=i,f=huftree[i].parent;f!=-1;k=f,f=huftree[f].parent)
        {
            if (huftree[f].lchild==k)
            {
                temp[c]='0' ;
                c++;
            }
            else
            {
                temp[c]='1';
                c++;
            }   //左分枝是0右分枝是1
        }
        cd[i].code=(char *)malloc(c+1);  //产生存储编码的空间
        cd[i].code[c]='\0';
        c--;
        k=0;
        while (c>=0)
            cd[i].code[k++]=temp[c--]; //将temp转置到cd中
        cd[i].leaf=huftree[i].ch;
        cd[i].length=k;
    }
}

/*功能:求哈夫曼树中各个节点的编码*/
/*传入参数:树huftree[],节点个数n,编码数组cd[]*/
void HuftreeCode(Htnode huftree[], CodeType cd[], int n)
{
    int i,c,f,k;
    char temp[N];/*暂存叶子编码字符串,最后需要转置*/
```

```
    for(i=1;i<=n;i++)/*开始求每个叶子结点的编码*/
    {
        c = 0;
        for(k=i,f=huftree[i].parent;f!=-1;k=f,f=huftree[f].parent)
            if(huftree[f].lchild == k)/*左分支是0*/
                temp[c++]='0';
            else
                temp[c++]='1';/*右分支是1*/
        cd[i].code=(char *)malloc(c+1); /*产生存储编码的空间*/
        cd[i].code[c--]='\0';
        k = 0;
        while(c>=0)
            cd[i].code[k++]=temp[c--];//将temp转置到cd中*/
        //cd[i].leaf=huftree[i].ch;
        cd[i].length=k;
    }
}

int main()
{
    Htnode huftree[2*N];   //夫曼树的数组
    CodeType cd[N];    //节点的编码数组
    int w[N],n,i,temp,sum=0;  //节点的临时存放及节点的个数
    printf("输入带权节点的个数:");  //输入带权节点的个数
    scanf("%d",&n);
    if(n < N)
    {
        A:printf("输入带权值的数:\n");   //输入带权值的数
        for(i = 1; i <= n; i++)
        {
            scanf("%d",&temp);
            w[i-1] = temp;
        }
        Hufcoding(huftree,cd,w,n);  //创建哈夫曼树
        printf("HuftreeCode每个节点的编码:\n");  //打印每个节点的编码
        HuftreeCode(huftree, cd, n);
        for(i = 1; i <= n; i++)
        {
            printf("%d 的 HuftreeCode 节点的编码:",huftree[i].weight);
            puts(cd[i].code);
        }
        printf("Huftree 带权路径长度:\n");   //打印WPL(带权路径长度)
        for(i = 1; i <= n; i++)
        {
```

```
            printf("%d * %d + ",huftree[i].weight,cd[i].length);
            sum += huftree[i].weight * cd[i].length;
        }
        printf("0 = %d\n",sum);
    }
    else
    {
        printf("输入错误！\n");
        goto A;
    }
}

/*
 输入带权节点的个数:5
 输入带权值的数:
 1
 2
 3
 4
 5
 HuftreeCode每个节点的编码:
 1 的 HuftreeCode 节点的编码:010
 2 的 HuftreeCode 节点的编码:011
 3 的 HuftreeCode 节点的编码:00
 4 的 HuftreeCode 节点的编码:10
 5 的 HuftreeCode 节点的编码:11
 Huftree 带权路径长度:
 1 * 3 + 2 * 3 + 3 * 2 + 4 * 2 + 5 * 2 + 0 = 33
 */
```

二叉树·遍历·全

```
//
//二叉树·遍历·全
//
#include <stdio.h>
#include <stdlib.h>
//#include <stdbool.h> //一般stdbool.h头文件中，函数bool开头，返回值为1或0，false或true
#define maxsize 100

typedef char datatype;

int count=0;//全局变量
```

```
typedef struct tnode{//二叉树定义，递归遍历
    datatype data;
    struct tnode *lchild;
    struct tnode *rchild;
}tnode,*bintree;

typedef struct snode{//顺序栈定义,非递归遍历
    bintree data[maxsize];
    int top;
}seqstack,*pseqstack;

typedef struct qnode{//顺序队列定义，层次遍历
    bintree data[maxsize];
    int front;
    int rear;
}seqqueue,*pseqqueue;


//*******二叉树部分********//
bintree create(void)
{//构建二叉树
    bintree t;
    datatype ch;
    ch=getchar();
    if(ch=='#')
        t=NULL;
    else
    {
        t=(bintree)malloc(sizeof(tnode));
        t->data=ch;
        t->lchild=create();
        t->rchild=create();
    }
    return t;
}

//*******顺序栈部分********//
pseqstack init(void)
{//创建顺序栈
    pseqstack s;
    s=(pseqstack)malloc(sizeof(seqstack));
    if(s)
        s->top=-1;
```

```
    return s;
}
bool empty(pseqstack s)
{//判断栈是否为空
    if(s->top==-1)
        return 1;
    else
        return 0;
}
bool push(pseqstack s,bintree x)
{//栈顶插入新元素x
    if(s->top==maxsize-1)
        return 0;//栈满无法入栈
    else
    {
        s->top++;
        s->data[s->top]=x;
        return 1;
    }
}
bool pop(pseqstack s,bintree *x)
{//删除栈顶元素，并保存在*x
    if(empty(s))
        return 0;//栈空不能出栈
    else
    {
        *x=s->data[s->top];
        s->top--;
        return 1;
    }
}

//*******队列部分********//
pseqqueue initqueue(void)
{//初始化队列
    pseqqueue q;
    if (!(q=(pseqqueue)malloc(sizeof(seqqueue))))
    {
        printf("内存分配失败！");
        exit(-1);
    }
    q->front=q->rear=-1;
    return q;
}
```

```
bool emptyqueue(pseqqueue q)
{//队列判空
    if(q->front==q->rear)
        return 1;//队空
    return 0;
}
bool inqueue(pseqqueue q,bintree t)
{//入队
    if(q->rear==maxsize-1)
        return 0;//队满
    q->rear++;
    q->data[q->rear]=t;
    return 1;
}
bool outqueue(pseqqueue q,bintree *t)
{//出队
    if(q->front==q->rear)
        return 0;
    q->front++;
    *t=q->data[q->front];
    return 1;
}

//*******遍历部分********//
void perorder1(bintree t)
{//先序遍历的递归算法
    if(t)
    {
        printf("%c ",t->data);
        perorder1(t->lchild);
        perorder1(t->rchild);
    }
}
void inorder1(bintree t)
{//中序遍历的递归算法
    if(t)
    {
        inorder1(t->lchild);
        printf("%c ",t->data);
        inorder1(t->rchild);
    }
}
void postorder1(bintree t)
{//后序遍历的递归算法
```

```
    if(t)
    {
        postorder1(t->lchild);
        postorder1(t->rchild);
        printf("%c ",t->data);
    }
}

void preorder2(bintree t)
{//栈的先序遍历的非递归算法
    pseqstack s;
    bintree p=t;
    s=init();
    while (p||!empty(s))
    {
        if(p)
        {
            push(s,p);
            printf("%c ",p->data);
            p=p->lchild;
        }
        else
        {
            pop(s,&p);
            p=p->rchild;
        }
    }
}
void inorder2(bintree t)
{//栈的中序遍历的非递归算法
    pseqstack s;
    bintree p=t;
    s=init();
    while (p||!empty(s))
    {
        if(p)
        {
            push(s,p);
            p=p->lchild;
        }
        else
        {
            pop(s,&p);
            printf("%c ",p->data);
```

```
            p=p->rchild;
        }
    }
}
void postorder2(bintree t)
{//栈的后序遍历的非递归算法
    pseqstack s1;//最终结果栈
    pseqstack s2;//辅助栈
    bintree p=t;
    s1=init();
    s2=init();
    while (p||!empty(s2))
    {
        if(p)
        {
            push(s1,p);
            push(s2,p);
            p=p->rchild;
        }
        else
        {
            pop(s2,&p);
            p=p->lchild;
        }
    }
    while (!empty(s1))
    {
        pop(s1,&p);
        printf("%c ",p->data);
    }
}

void preorder3(bintree t)
{//先序非递归-“栈”
    bintree p=t;
    bintree s[maxsize];
    int top=-1;
    do{
        while(p){
            top++;
            s[top]=p;//保留当前 p 结点在栈 s 中
            printf("%c ",p->data);
            p=p->lchild;//转到当前 p 结点的左孩子
        }
```

```
        p=s[top];//取 p 的上一个结点
        top--;//“头指针”减 1
        p=p->rchild;//转到当前 p 结点的右孩子
    }while(top!=-1||p);
}
void inorder3(bintree t)
{//中序非递归-“栈”
    bintree p=t;
    bintree s[maxsize];
    int top=-1;
    do{
        while (p)
        {
            top++;
            s[top]=p;//保留当前 p 结点在栈 s 中
            p=p->lchild;//转到当前 p 结点的左孩子
        }
        p=s[top];//取 p 的上一个结点
        top--;//“头指针”减 1
        printf("%c ",p->data);
        p=p->rchild;
    }while(top!=-1||p);
}
void postorder3(bintree t)
{//后序非递归-“栈”
    bintree p=t;
    bintree s1[maxsize];//最终结果栈
    bintree s2[maxsize];//辅助栈
    int top1=-1;
    int top2=-1;
    do{
        while (p){
            top1++;
            top2++;
            s1[top1]=p;//保留当前 p 结点在栈 s1 中
            s2[top2]=p;//保留当前 p 结点在栈 s2 中
            p=p->rchild;//转到当前 p 结点的左孩子
        }
        p=s2[top2];//取 p 的上一个结点
        top2--;//“头指针”减 1
        p=p->lchild;//转到当前 p 结点的左孩子
    }while(top2!=-1||p);
    while (top1!=-1)
    {
```

```
        p=s1[top1];//s1 从后往前输出 p 结点
        printf("%c ",p->data);
        top1--;
    }
}

void levelorder(bintree t)
{//层次遍历
    pseqqueue q;
    q=initqueue();//创建队列
    if(t!= NULL)//执行一次
        inqueue(q,t);//将第一个结点 t 存入队列中
    while (!emptyqueue(q))
    {
        outqueue(q,&t);// 出队时的节点
        printf("%c ",t->data);// 输出节点存储的值
        if(t->lchild!= NULL)//有左孩子时将该节点进队列
            inqueue(q,t->lchild);
        if(t->rchild != NULL)//有右孩子时将该节点进队列
            inqueue(q,t->rchild);
    }
}

int main()
{
    bintree t;
    printf("请输入字符串:");
    t=create();

    printf("\n先序递归: ");
    perorder1(t);
    printf("\n中序递归: ");
    inorder1(t);
    printf("\n后序递归: ");
    postorder1(t);

    printf("\n先序非递归-栈: ");
    preorder2(t);
    printf("\n中序非递归-栈: ");
    inorder2(t);
    printf("\n后序非递归-栈: ");
    postorder2(t);

    printf("\n先序非递归-“栈”: ");
```

```
    preorder3(t);
    printf("\n中序非递归-“栈”: ");
    inorder3(t);
    printf("\n后序非递归-“栈”: ");
    postorder3(t);

    printf("\n层次遍历-队列: ");
    levelorder(t);

    printf("\n");
}
/*
测试数据:
ABDH###E##CF##G##
1248###5##36##7##
1248##9##50###36##7##



请输入字符串:1248##9##50###36##7##

先序递归: 1 2 4 8 9 5 0 3 6 7
中序递归: 8 4 9 2 0 5 1 6 3 7
后序递归: 8 9 4 0 5 2 6 7 3 1
先序非递归-栈: 1 2 4 8 9 5 0 3 6 7
中序非递归-栈: 8 4 9 2 0 5 1 6 3 7
后序非递归-栈: 8 9 4 0 5 2 6 7 3 1
先序非递归-“栈”: 1 2 4 8 9 5 0 3 6 7
中序非递归-“栈”: 8 4 9 2 0 5 1 6 3 7
后序非递归-“栈”: 8 9 4 0 5 2 6 7 3 1
层次遍历-队列: 1 2 3 4 5 6 7 8 9 0

 */
```

串·堆存储结构·静态·标准

```
//
//  main.c
//  串·堆存储结构·静态·标准
//

#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#define maxsize 100
#define SMAX 100

char store[SMAX+1];//堆空间
int free1;//自由区指针

typedef struct
{//带串长度的索引表
    char name[maxsize];//串名
    int length;//串长
    char * stradr;//起始地址
}lnode;

typedef struct
{//末尾指针的索引表
    char name[maxsize];//串名
    char * stradr,* enadr;//起始地址，末尾地址
}enode;

typedef struct
{
    int length;//串长
    int stradr;//起始地址
}hstring,*phstring;

phstring InitString(void)
{//开辟 字符串 空间
    phstring H=(phstring)malloc(sizeof(hstring));
    if(NULL==H)
    {
        printf("Memory allocate is error!");
        system("pause");
        exit(0);
    }
    else
    {
        H->stradr = 0;
        H->length = 0;
        return H;
    }
}

int strassign(phstring s1,char *s2)
```

```
{//将一个字符数组s2中的字符串送入堆stroe中，free1是自由区指针，正常操作返回1
    int i=0;
    int len;
    len=(int)(strlen(s2));
    if(len<0||free1+len > SMAX+1)
        return 0;
    else
    {
        for(i=0;i<len;i++)
            store[free1+i]=s2[i];
        s1->stradr=free1;
        s1->length=len;
        free1=free1+len;    //修改自由区指针
        printf("s串为: ");
        for(i=0;i<len;i++)  //输出
            printf("%c",store[s1->stradr+i]);
        printf("\n");
        return 1;
    }
}

int strcopy(phstring s1,phstring s2)
{//该运算将堆store中的s2复制到一个新串s1中
    int i=0;
    if(free1+s2->length > SMAX+1)
        return 0;
    else
    {
        for(i=0;i<s2->length;i++)
            store[free1+i]=store[s2->stradr+i];
        s1->length=s2->length;
        s1->stradr=free1;
        free1=free1+s2->length;
        printf("拷贝的新串为: ");
        for(i=0;i<s1->length;i++)  //输出
            printf("%c",store[s1->stradr+i]);
        printf("\n");
        return 1;
    }
}

int main(int argc, const char * argv[])
{
    phstring s1=InitString(); //串空间初始化
```

```
    phstring s2=InitString();

    strassign(s1, "love ");  //串进堆
    strassign(s2, "you ");
    strcopy(s2, s1);  //串拷贝，把s2拷贝到s1

    printf("store中全部的串为：");
    for(int i=0;i<free1;i++)
        printf("%c",store[i]);
    printf("\n");
    return 0;
}
/*
 s串为：love
 s串为：you
 拷贝的新串为：love
 store中全部的串为：love you love
 */
```

串·堆存储结构·动态·标准

```
//
//  main.c
//  串·堆存储结构·动态·标准
//

#include <stdio.h>
#include <stdlib.h>

typedef struct
{
    char *p;
    int length;
}hstring,*phstring;

phstring InitString(void)
{//开辟 字符串 空间
    phstring H=(phstring)malloc(sizeof(hstring));
    if(NULL==H)
    {
        printf("内存开辟失败!");
        exit(0);
    }
    else
```

```
    {
        H->p=NULL;
        H->length=0;
        return H;
    }
}

int strassign(phstring s1,char *s2)
{//串常量赋值，将一个字符串常量的值赋值给一个字符串变量
    int i;
    char *pc;
    if(s1->p)
        free(s1->p);
    for(i=0,pc=s2;*pc!='\0';i++,pc++);//求s2长度
    //printf("%d",i);
    if(i==0)
    {
        s1->p=NULL;
        s1->length=0;
        return 0;
    }
    if(!(s1->p=(char *)malloc((i+1) * sizeof(char))))
    {// +1 为了添加字符串结束标识符需要多申请一个字节内存;
        printf("堆空间不足，赋值失败！\n");
        return 0;
    }
    for(int j=0;j<i;j++)
        s1->p[j]=s2[j];
    s1->p[i]='\0';  //字符串结束标识符
    s1->length=i;
    printf("字符串长度为：%d\n",s1->length);
    return 1;//赋值成功
}

int strcopy(phstring s1,hstring s2)
{//s2也可以输入为 phstring s2 ，后面 . 全改为 ->
//赋值一个串,将一个字符串的值赋值给一个字符串变量,将s2赋值给s1
    if(s2.length<=0)
        return 0;
    if(!(s1->p=(char *)malloc((s2.length+1) * sizeof(char))))
    {
        printf("堆空间不足,赋值失败!\n");
        return 0;
    }
```

```
    for(int i=0;i<s2.length;i++)
        s1->p[i]=s2.p[i];
    s1->length=s2.length;
    s1->p[s2.length]='\0';
    //for(int i=0;i<7;i++)     //输出 s1  方法一
        //printf("%c",s1->p[i]);
    printf("赋值串s3后输出: %s\n",s1->p);   //输出 s1  方法二
    return 1;//赋值成功
}

int substring(phstring sub, phstring s, int pos, int len)
{//求子串,用sub返回串s的第pos个字符起长度为len的子串;其中,1≤pos≤strlength(s)且
0≤len≤strlength(s)-pos+1
    int i;
    if(pos<1||pos>s->length||len<0||len>s->length-pos+1)
        return 0;
    if(sub->p)//释放旧空间
        free(sub->p);
    if(!len)
    {
        sub->p=0;
        sub->length=0;//空子串
    }
    else//完整子串
    {
        sub->p=(char *)malloc((len+1) * sizeof(char));
        for(i=0;i<len;i++)
            sub->p[i]=s->p[pos+i-1];
        sub->p[len]='\0';
        sub->length=len;
    }
    printf("求s1第%d位长度%d的子串sub: %s\n",pos,len,sub->p);  //输出子串
    return 1;
}


int strcontact(phstring t,phstring s1,phstring s2)
//int strcontact(hstring *t,hstring s1,hstring s2)
{//串连接,t 保存由字符串 s1 和 s2 连接而成的新串
    int i;
    if(t->p)
        free(t->p);//释放旧空间
    if(!(t->p=(char*)malloc((s1->length+s2->length+1)*sizeof(char))))
        printf("堆空间不足,串连接失败!\n");
```

```
    for(i=0;i<s1->length;i++)
        t->p[i]=s1->p[i];
    t->length=s1->length+s2->length;
    for(i=s1->length;i<t->length;i++)
        t->p[i]=s2->p[i-s1->length];
    t->p[t->length]='\0';
    printf("由s1和s2连接的串t为: %s\n",t->p);  //输出 t
    return 1;
}

int strinsert(phstring s,int pos,phstring t)
{//在目标串的指定位置前插入字符串,1≤pos≤strlength(s)+1,在串s的第pos字符前插入串t
    int i;
    if(pos<1||pos>s->length+1)
        return 0;
    if(t->length==0)//t是空串
        return 1;
    if(!(s->p=(char *)realloc(s->p,(s->length+t->length+1)*sizeof(char))))
    {       //realloc重新分配 s->p 的内存
        printf("堆空间不足,插人失败!\n");
        return 0;
    }
    for(i=s->length-1;i>=pos-1;i--)//让出插人的位置
        s->p[i+t->length]=s->p[i];
    for(i=pos-1;i<=pos+t->length-2;i++)
        s->p[i]=t->p[i-pos+1];
    s->length=s->length+t->length;
    s->p[s->length]='\0';
    printf("在串s2的第%d字符前插入串s3: %s\n",pos,s->p);  //输出 t
    return 1;
}

int initstring(hstring *s)
{//置空串
    s->p=0;//指针置空
    s->length=0;
    return 1;
}

int destorystring(phstring s)
{//销毁串
    if(s->length)
    {
        free(s->p);
```

```
        s->p=0;//指针收起
        s->length=0;
    }
    return 1;
}


int main(int argc, const char * argv[]) {
    phstring s1=InitString();
    phstring s2=InitString();
    phstring s3=InitString();
    phstring sub=InitString();
    phstring t=InitString();

    strassign(s1,"abcdefg");// 将 常量字符串 赋值给 s1
    strassign(s2,"12345678");// 将 常量字符串 赋值给 s2
    strcopy(s3, *s1);  //将s1 赋值给 s3
    substring(sub, s1, 2, 4);  //求 s1第 2位长度 4的子串sub
    strcontact(t, s1, s2);  //将 s1和 s2连接，给 t
    strinsert(s2, 3, sub);   //在串 s2的第 3字符前插入串 sub

    destorystring(s1);
    destorystring(s2);
    destorystring(s3);
    destorystring(sub);
    destorystring(t);
}
/*
 字符串长度为：7
 字符串长度为：8
 赋值串s3后输出：abcdefg
 求s1第2位长度4的子串sub：bcde
 由s1和s2连接的串t为：abcdefg12345678
 在串s2的第3字符前插入串s3：12bcde345678
 */
```

数据结构·栈·顺序存储

```
//
//  main.c
//  数据结构·栈·顺序存储
//
#include <stdio.h>
#include <stdlib.h>
```

```c
#define MAXSIZE 100
typedef int datatype;

typedef struct {
    datatype data[MAXSIZE];
    int top;
}seqstack,*pseqstack;

pseqstack start(void)
{//创建顺序栈
    pseqstack s;
    s=(pseqstack)malloc(sizeof(seqstack));
    if(s)
        s->top=-1;
    return s;
}

int empty(pseqstack s)
{//判断栈是否为空
    if(s->top==-1)
        return 1;
    else
        return 0;
}

int push(pseqstack s,datatype x)
{//栈顶插入新元素x
    if(s->top==MAXSIZE-1)
        return 0;//栈满无法入栈
    else
    {
        s->top++;
        s->data[s->top]=x;
        return 1;
    }
}

int pop(pseqstack s,datatype *x)
{//删除栈顶元素，并保存在*x
    if(empty(s))
        return 0;//栈空不能出栈
    else
    {
        *x=s->data[s->top];
```

```
        s->top--;
        return 1;
    }
}

int gettop(pseqstack s,datatype *x)
{//取出栈顶元素
    if(empty(s))
        return 0;//栈空
    else
    {
        *x=s->data[s->top];//栈顶元素存入*x中
        return (1);
    }
}

void destroy(pseqstack *s)
{//销毁栈
    if(*s)
        free(*s);
    *s=NULL;
    return ;
}

int main()
{
    printf("xxx\n");
    return 0;
}
```

数据结构·栈·链式存储

```
//
//  main.c
//  数据结构·栈·链式存储
//

#include <stdio.h>
#include <stdlib.h>
#define MAXSIZE 100
typedef int datatype;

typedef struct node {
```

```
    datatype data;
    struct node *next;
}stacknode,*pstacknode;

typedef struct {
    pstacknode top;
}linkstack,*plinkstack;

plinkstack start(void)
{//创建链式栈
    plinkstack s;
    s=(plinkstack)malloc(sizeof(linkstack));
    if(s)
        s->top=NULL;
    return s;
}

int empty(plinkstack s)
{
    return (s->top==NULL);
}

int push(plinkstack s,datatype x)
{
    pstacknode p;
    p=(pstacknode)malloc(sizeof(stacknode));
    if(!p)
    {
        printf("内存溢出");
        return 0;
    }
    p->data=x;
    p->next=s->top;
    s->top=p;
    return (1);
}

int pop(plinkstack s,datatype *x)
{
    pstacknode p;
    if(empty(s))
    {
        printf("栈空，不能出栈");
        return  0;
```

```
    }
    *x=s->top->data;
    p=s->top;
    s->top=s->top->next;
    free(p);
    return (1);
}

int gettop(plinkstack s,datatype *x)
{//取出栈顶元素
    if(empty(s))
    {
        printf("栈空");
        return 0;//栈空
    }
    else
    {
        *x=s->top->data;//栈顶元素存入*x中
        return (1);
    }
}

void destroy(plinkstack *s)
{
    pstacknode p,q;
    if(*s)
    {
        p=(*s)->top;
        while (p) {
            q=p;
            p=p->next;
            free(q);
        }
        free(*s);
    }
    *s=NULL;
}


int main()
{
    printf("xxx! \n");
    return 0;
}
```

数据结构·队列·顺序存储

```
//
//  main.c
//  数据结构·队列·顺序存储
//
#include <stdio.h>
#include<stdlib.h>
#define maxsize 100
typedef int datatype;

typedef struct{
    datatype data[maxsize];
    int front,rear;
}seqqueue,*pseqqueue;

pseqqueue init_seqqueue(void)
{//创建队列
    pseqqueue q;
    q=(pseqqueue)malloc(sizeof(seqqueue));
    if(q)
    {
        q->front=0;
        q->rear=0;
    }
    return q;
}

int empty_seqqueue(pseqqueue q)
{//队列判空
    if(q&&q->front==q->rear)
        return 1;
    else
        return 0;
}

int in_seqqueue(pseqqueue q,datatype x)
{//入队
    if((q->rear+1)%maxsize==q->front)
    {
        printf("队满");
        return -1;
    }
    else
```

```
    {
        q->rear=(q->rear+1)%maxsize;
        q->data[q->rear]=x;
        return 1;
    }
}

int out_seqqueue(pseqqueue q,datatype *x)
{//出队
    if(empty_seqqueue(q))
    {
        printf("队空");
        return -1;
    }
    else
    {
        q->front=(q->front+1)%maxsize;
        *x=q->data[q->front];
        return 1;
    }
}

int front_seqqueue(pseqqueue q,datatype *x)
{
    if(q->front==q->rear)
    {
        printf("队空");
        return -1;
    }
    else
    {
        *x=q->data[(q->front+1)]%maxsize;
        return 1;
    }
}

void destroy_seqqueue(pseqqueue *q)
{//销毁队列
    if(*q)
        free(*q);
    *q=NULL;
}
int main(int argc, const char * argv[]) {
    printf("xxx! \n");
```

```
    return 0;
}
```

数据结构·队列·链式存储

```
//
//  main.c
//  数据结构·队列·链式存储
//
#include <stdio.h>
#include<stdlib.h>
#define maxsize 100
typedef int datatype;

typedef struct node{
    datatype data;
    struct node *next;
}qnode,*pqnode;

typedef struct{
    pqnode front,rear;
}linkqueue,*plinkqueue;

plinkqueue init_linkqueue(void)
{//初始化队列
    plinkqueue q;
    q=(plinkqueue)malloc(sizeof(linkqueue));
    if(q)
    {
        q->front=NULL;
        q->rear=NULL;
    }
    return 0;
}

int empty_linkqueue(plinkqueue q)
{//判空
    if(q&&q->front==NULL&&q->rear==NULL)
        return 1;
    else
        return 0;
}

int in_linkqueue(plinkqueue q,datatype x)
{//进队
```

```
    pqnode p;
    p=(pqnode)malloc(sizeof(qnode));
    if(!q)
    {
        printf("内存溢出");
        return 0;
    }
    p->data=x;
    p->next=NULL;
    if(empty_linkqueue(q))
        q->rear=q->front=p;
    else{
        q->rear->next=p;
        q->rear=p;
    }
    return 1;
}

int out_linkqueue(plinkqueue q,datatype *x)
{//出队
    pqnode p;
    if(empty_linkqueue(q))
    {
        printf("队空");
        return 0;
    }
    *x=q->front->data;
    p=q->front;
    q->front=q->front->next;
    free(p);
    if(!q->front)
        q->rear=NULL;
    return 1;
}

int front_linkqueue(plinkqueue q,datatype *x)
{//队头指针
    if(empty_linkqueue(q))
    {
        printf("队空");
        return 0;
    }
    *x=q->front->data;
    return 1;
```

```
}

void destroy_linkqueue(plinkqueue q)
{//销毁队
    pqnode p;
    if(q)
    {
        while (q->front) {
            p=q->front;
            q->front=q->front->next;
            free(p);
        }
        free(q);
    }
    q=NULL;
}

int main() {
    printf("xxx! \n");
    return 0;
}
```

串·顺序存储·标准

```
//
//  main.c
//  串·顺序存储·标准
//

#include <stdio.h>

#define maxsize 256
char s[maxsize];

typedef struct
{
    char data[maxsize];
    int length;
}seqstring;

int strlength(char *s)
{//求串长
    int i=0;
    while(s[i]!='\0')
        i++;
```

```
    return i;
}

int strconcat(char *s1,char *s2,char *s)
{//新串储存在指针s中,串连接
    int i=0,j,len1,len2;
    len1=strlength(s1);
    len2=strlength(s2);
    if(len1+len2>maxsize-1)
        return 0;
    j=0;
    while(s1[j]!='\0')
    {
        s[i]=s1[j];
        i++;
        j++;
    }
    j=0;
    while(s2[j]!='\0')
    {
        s[i]=s2[j];
        i++;
        j++;
    }
    s[i]='\0';
    return 1;
}

int strsub(char *t,char *s,int i,int len)
{//求子串，用t返回串s中第i个字符开始的长度为len的子串，1≤i≤串长
    int slen,j;
    slen=strlength(s);
    if(i<1||i>slen||len>slen-i+1)
    {
        printf("参数不对");
        return 0;
    }
    for(j=0;j<len;j++)
        t[j]=s[i+j-1];
    t[j]='\0';
    return 1;
}

int Strcmp(char *s1,char *s2)
```

```
{//串比较
    int i=0;
    while(s1[i]==s2[i]&&s1[i]!='\0')
        i++;
    return  s1[i]==s2[i];
}

int main(int argc, const char * argv[])
{
    char s1[10]="12345";
    char s2[10]="6789";
    char s[20],t[10];

    int l=strlength(s1);    //求串长
    printf("s1串长为：%d\n",l);

    strconcat(s1, s2, s);  //串连接
    printf("串连接：%s\n",s);

    strsub(t, s1, 1, 3);   //求子串
    printf("求子串：%s\n",t);

    int c=Strcmp(s1, s2);  //串比较
    printf("串比较：%d(相等为1，不等为0)\n",c);  //相等为1，不等为0

    return 0;
}
/*
 s1串长为：5
 串连接：123456789
 求子串：123
 串比较：0(相等为1，不等为0)
 */
```

串·链式存储·标准

```
//
//  main.c
//  串·链式存储·网1

#include <stdio.h>
#include <stdlib.h>

//链串插入 删除 连接
```

```
typedef char datatype;
typedef struct node
{
    datatype  data;
    struct node *next;
}node;
typedef node *linkstr;

void createstr(linkstr *s)//创建字符串
{
    char ch;
    node *p,*r; //p,r,linkstr(*s)都是节点
    *s=NULL;  //*s记录第一个节点
    r=NULL;//初始化节点为空
    while((ch=getchar())!='\n')
    {
        //p=(linkstr)malloc(sizeof(node));
        p=(node*)malloc(sizeof(node));
        p->data=ch;
        if(*s==NULL)//执行第一次
        {
            *s=p;
            r=p;
        }
        else
        {
            r->next=p;//划线
            r=p;
        }
    }
    if(r!=NULL)
        r->next=NULL;
}
linkstr strinsert(linkstr *s,int i,linkstr t)//插入
{//原始字符串；插入位置（第i个字母后面）；要插入的字符串
    int k=1;
    linkstr p,q;
    p = *s;//p为s串的第一个节点
    while(p&&k<i-1)//用p查找第i-1个位置
    {
        p=p->next;
        k++;
    }
```

```
    if(!p)//第i-1个元素不存在,则出错
    {
        puts("error!");
        exit(1);
    }
    else
    {
       q=t;
       while(q&&q->next)  //用q查找t中最后一个元素的位
           q=q->next;
       if(i==1)
       {
           q->next=p;//q的最后一个指向s
           *s=t;//s头结点给t
           return *s;
       }
        q->next=p->next; //n和d   将t连接到s中的第i个位置
        p->next=t;//c和m划线
    }
    return *s;
}

void del(linkstr *s,int i,int length)//删除
{//原始字符串；第i个字符开始删；所删长度为length
    int k=1;
    linkstr p,q,r;
    p=*s;
    q=NULL;  //q=null删除第一个赋初始值
    //第i个位置第一个位置
    while(p&&k<i)  //用p查找s的第i个元素，q始终跟随p的前驱节点
    {
        q=p;
        p=p->next;
        ++k;
    }
    if(!p)//*s从第i个元素不存在则出错
        printf("error!");
    else
    {
        k=1;
        while(k<length&&p)  //p从第i个元素开始查找长度为len子串的最后元素
        {
            p=p->next;
            ++k;
```

```
        }
        if(!p)
            printf("two error!");
        else
        {
            if(!q)    //被删除的子串位于s
            {
                r=*s;//用r记录原字符串第一个节点
                *s=p->next;//*s记录删除后的新起始节点
            }
            else   //被删除的子字符串位于s的中间或者最后的情形
            {
               r=q->next;  //用r记录要删除子串的第一个节点
               q->next=p->next;
            }
            p->next=NULL;
            while(r!=NULL)    //回收子字符串占用的空间
            {
                p=r;
                r=r->next;
                free(p);
            }
        }
    }
}

void concatstr(linkstr *s1,linkstr s2)//连接
{//把串s2连接到s1后
    linkstr p;
    if(!(*s1))//考虑串s1为空串时
    {
        *s1=s2;
        return ;
    }
    else
    {
        if(s2) //s1和s2均不为空串时
        {
            p=*s1;//用p查找s1的最后一个字符的位置
            while(p->next)
            {
                p=p->next;
            }
            p->next=s2;//将串s2连接到串s1之后
```

```
        }
    }
}

linkstr substring(linkstr *s,int i,int len)//取链式串的子串
{//原始字符串；第i个字符开始，取长度len的子串
    int k=1;//p为原字符串第一个节点，用p记录s中的第i个位置
    linkstr p=*s,q,r,t;
    while(p&&k<i)
    {
        p=p->next;
        ++k;
    }
    if(!p)
    {
        printf("error!");
        return NULL;
    }
    else
    {//r保存第一个节点
        r=(node*)malloc(sizeof(node));
        r->data=p->data;
        r->next=NULL;
        k=1;
        q=r;
        while(p->next&&k<len)
        {
            p=p->next;//p第二个节点
            ++k;
            t=(node*)malloc(sizeof(node));//保存第二个节点t++节点
            t->data=p->data;
            q->next=t; //第一个和第二个节点划线
            q=t;      //q指向字符串最后一个位置
        }
        if(k<len)
        {
            printf("two error!");
            return NULL;
        }
        else
        {
            q->next=NULL;
            return r;
        }
```

```
    }
}
void dispaly(linkstr *s)
{
    node *p;
    //linkstr p;
    p=*s;
    while(p)
    {
        printf("%c",p->data);
        p=p->next;
    }
    printf("\n");
}
int main()
{
    linkstr s,t;
    printf("输入初始字符串1: ");
    createstr(&s);//创建原始字符串
    printf("   1.输出创建好字符串s: ");
    dispaly(&s);


    printf("输入要插入的字符串: ");//插入
    createstr(&t);//创建要插入的字符串
    strinsert(&s,5,t);
    printf("   2.输出插入后的字符串s: ");
    dispaly(&s);

    del(&s,2,2);//删除,从第2位开始到第2位后面删掉
    printf("   3.输出删除后的字符串s: ");
    dispaly(&s);
    printf("\n");

    printf("输入要连接的字符串2: ");
    createstr(&t);
    concatstr(&s,t);//连接
    printf("   4.输出连接后的字符串s: ");
    dispaly(&s);

    t=substring(&s,1,1);//取子串，从第1位开始到第1位后面取子串
    printf("   5.输出子串s: ");
    dispaly(&t);
```

```
    return 0;
}
/*
测试数据:

mzt  student
is a sun
 forever

输出结果:

 输入初始字符串1: mzt  student
    1.输出创建好字符串s: mzt  student
 输入要插入的字符串: is a sun
    2.输出插入后的字符串s: mzt is a sun student
    3.输出删除后的字符串s: m is a sun student

 输入要连接的字符串2:  forever
    4.输出连接后的字符串s: m is a sun student forever
    5.输出子串s: m

 哈哈哈哈哈哈哈！发现了，串连接蛮好玩的~

 */
```

二叉树·应用举例·基本操作

```
//
//  main.c
//  二叉树·应用举例·基本操作

#include <stdio.h>
#include <stdlib.h>
#define maxsize 100

typedef char datatype;
int count=0;//全局变量

typedef struct tnode{
    datatype data;
    struct tnode *lchild;
    struct tnode *rchild;
}Tnode,*bintree;
```

```
bintree create(void)
{//构造二叉链表创建二叉树,以加入空结点的先序序列输入
    bintree t;
    datatype ch;
    ch=getchar();
    if(ch=='#')   //读入#时,将相应结点指针置空
        t=NULL;
    else
    {
        t=(bintree)malloc(sizeof(Tnode));//生成结点空间
        t->data=ch;
        t->lchild=create();//构造二叉树的左子树
        t->rchild=create();//构造二叉树的右子树
    }
    return (t);
}

int count_tree(bintree t)
{//求二叉树的结点个数,中序思路
    if(t==NULL)
        return 0;
    if(t)
    {
        count_tree(t->lchild);
        count++;
        count_tree(t->rchild);
    }
    return count;
}

int Count(bintree t)
{//求二叉树的结点个数,后序思路
    int lcount,rcount;
    if (t==NULL)
        return 0;
    lcount=Count(t->lchild);     //求左子树的结点个数
    rcount=Count(t->rchild);     //求右子树的结点个数
    return  lcount+rcount+1;
}

int high(bintree t)
{//求二叉树高度，后序思路
    int l,r;
    if(t==NULL)
```

```
        return 0;
    else
    {
        l=high(t->lchild);//左子树高度
        r=high(t->rchild);//右子树高度
        if(l>r)
            return l+1;
        return r+1;
    }
}

bintree CopyTree(bintree  t)
{//复制二叉树算法
    bintree p,q,s;
    if (t==NULL)
        return NULL;
    p=CopyTree(t->lchild);    /*复制左子树*/
    q=CopyTree(t->rchild);    /*复制左子树*/
    s=(bintree)malloc(sizeof(Tnode));  /*复制根结点*/
    s->data=t->data;
    s->lchild=p;
    s->rchild=q;
    return s;
}

void Levcount(bintree t,int L,int num[])
{//求二叉树每层结点数算法,先序思路
    //求链式存储的二叉树t中每层结点个数L表示当前t所指结点的层次，当t初值为根时，L初值为1，num数组元素初始化0
    if(t)
    {
        printf("%c",t->data); //访问当前结点
        num[L]++;  //当前t所指结点的层次数增加1
        Levcount(t->lchild, L+1, num); //递归左子树
        Levcount(t->rchild, L+1, num); //递归右子树
     }
 }


int main(int argc, const char * argv[])
{
    printf("请输入二叉树：");
    bintree t=create(),s;
    printf("该二叉树结点个数为(中序思路)：%d\n",count_tree(t));
```

```
    printf("该二叉树高度为(后序思路): %d\n",Count(t));
    printf("该二叉树高度为(后序思路): %d\n",high(t));
    s=CopyTree(t);
    int num[10]={0};
    printf("二叉树每层结点数算法(先序思路):");
    Levcount(s, 1, num);
    printf("\n");
    return 0;
}

/*

测试数据:
ABDH###E##CF##G##
1248###5##36##7##
1248##9##50###36##7##


 请输入二叉树: 1248##9##50###36##7##
 该二叉树结点个数为(中序思路): 10
 该二叉树高度为(后序思路): 10
 该二叉树高度为(后序思路): 4
 二叉树每层结点数算法(先序思路):1248950367

 */
```

广义表·基本操作

```
//
//  main.c
//  广义表·基本操作·网

#include <stdio.h>
#include <stdlib.h>
typedef char ElemType;
typedef struct lnode
{   int tag;              //节点类型标识
    union
    {
        ElemType data;
        struct lnode *sublist;
    }val;
    struct lnode *link;     //指向下一个元素
} GLNode;
GLNode *CreateGL(char *s)          //返回由括号表示法表示s的广义表链式存储结构
```

```
{   GLNode *g;
    char ch=*s++;                 //取一个字符
    if (ch!='\0')                 //串未结束判断
    {   g=(GLNode *)malloc(sizeof(GLNode));//创建一个新节点
        if (ch=='(')              //当前字符为左括号时
        {   g->tag=1;             //新节点作为表头节点
            g->val.sublist=CreateGL(s);  //递归构造子表并链到表头节点
        }
        else if (ch==')')
            g=NULL;                //遇到')'字符,g置为空
        else if (ch=='#')           //遇到'#'字符，表示空表
            g->val.sublist=NULL;
        else                       //为原子字符
        {   g->tag=0;               //新节点作为原子节点
            g->val.data=ch;
        }
    }
    else                           //串结束,g置为空
        g=NULL;
    ch=*s++;                       //取下一个字符
    if (g!=NULL)                    //串未结束，继续构造兄递节点
    {
        if (ch==',')                //当前字符为','
            g->link=CreateGL(s);     //递归构造兄递节点
        else                       //没有兄弟了,将兄弟指针置为NULL
            g->link=NULL;
    }
    return g;                      //返回广义表g
}
int GLLength(GLNode *g)             //求广义表g的长度
{
    int n=0;
    g=g->val.sublist;               //g指向广义表的第一个元素
    while (g!=NULL)
    {
        n++;
        g=g->link;
    }
    return n;
}
int GLDepth(GLNode *g)               //求广义表g的深度
{
    int max=0,dep;
    if (g->tag==0)
```

```
        return 0;
    g=g->val.sublist;                 //g指向第一个元素
    if  (g==NULL)                     //为空表时返回1
        return 1;
    while (g!=NULL)                   //遍历表中的每一个元素
    {
        if (g->tag==1)                //元素为子表的情况
        {
            dep=GLDepth(g);           //递归调用求出子表的深度
            if (dep>max) max=dep;      //max为同一层所求过的子表中深度的最大值
        }
        g=g->link;                    //使g指向下一个元素
    }
    return(max+1);                    //返回表的深度
}
void DispGL(GLNode *g)                 //输出广义表g
{   if (g!=NULL)                      //表不为空判断
    {                                 //先输出g的元素
        if (g->tag==0)                 //g的元素为原子时
            printf("%c", g->val.data);  //输出原子值
        else                          //g的元素为子表时
        {   printf("(");               //输出'('
            if (g->val.sublist==NULL)   //为空表时
                printf("#");
            else                      //为非空子表时
                DispGL(g->val.sublist); //递归输出子表
            printf(")");              //输出')'
        }
        if (g->link!=NULL)
        {   printf(",");
            DispGL(g->link);          //递归输出g的兄弟
        }
    }
}
ElemType maxatom(GLNode *g)            //求广义表g中最大原子
{
    ElemType max1,max2;
    if (g!=NULL)
    {
        if (g->tag==0)
        {
            max1=maxatom(g->link);
            return(g->val.data>max1?g->val.data:max1);
        }
```

```
        else
        {
            max1=maxatom(g->val.sublist);
            max2=maxatom(g->link);
            return(max1>max2?max1:max2);
        }
    }
    else
        return 0;
}
int main()
{
    GLNode *g;
    char *str="(b,(b,a,(#),d),((a,b),c,((#))))";
    g=CreateGL(str);
    printf("广义表g:");DispGL(g);
    printf("\n");
    printf("广义表g的长度:%d\n",GLLength(g));
    printf("广义表g的深度:%d\n",GLDepth(g));
    printf("广义表g的最大原子:%c\n",maxatom(g));
}
/*
广义表g:(b,(b,a,((#)),d),((a,b),c,(((#)))))
广义表g的长度:3
广义表g的深度:4
广义表g的最大原子:d
 */
```

查找·哈希表·网

```
//
//  main.c
//  查找·哈希表·网

#include <stdio.h>
#include <stdio.h>
#define MaxSize 100      //定义最大哈希表长度
#define NULLKEY -1       //定义空关键字值
#define DELKEY -2        //定义被删关键字值
typedef int KeyType;     //关键字类型
typedef char *InfoType;  //其他数据类型
typedef struct{
    KeyType key;        //关键字域
    InfoType data;      //其他数据域
    int count;          //探查次数域
```

```
}HashTable[MaxSize];    //哈希表类型

//将关键字k插入到哈希表中
int InsertHT(HashTable ha,int n,KeyType k,int p)
{
    int i,adr;
    adr = k%p;
    if (ha[adr].key==NULLKEY || ha[adr].key==DELKEY)  //x[j]可以直接放在哈希表中
    {
        ha[adr].key=k;
        ha[adr].count=1;
    }
    else      //发生冲突时,采用线性探查法解决冲突
    {
        i=1;  //i记录x[j]发生冲突的次数
        do
        {
            adr = (adr+1)%p;
            i++;
        } while (ha[adr].key!=NULLKEY && ha[adr].key!=DELKEY);
        ha[adr].key=k;
        ha[adr].count=i;
    }
    n++;
    return n;
}

//创建哈希表
void CreateHT(HashTable ha,KeyType x[],int n,int m,int p){
    int i,n1=0;
    for (i=0;i<m;i++)  //哈希表置初值
    {
        ha[i].key=NULLKEY;
        ha[i].count=0;
    }
    for(i=0;i<n;i++)
        InsertHT(ha,n1,x[i],p);
}

//在哈希表中查找关键字k
int SearchHT(HashTable ha,int p,KeyType k){
    int i=0,adr;
    adr = k%p;
    while (ha[adr].key!=NULLKEY && ha[adr].key!=k)
```

```
    {
        i++;            //采用线性探查法找下一个地址
        adr = (adr+1)%p;
    }
    if(ha[adr].key==k)  //查找成功
    {
        printf("  ha[%d].key=%d\n",adr,k);
        return adr;  //返回k所在位置
    }
    else              //查找失败
    {
        printf("  未找到%d\n",k);
        return -1;
    }
}

//删除哈希表中关键字k
int DeleteHT(HashTable ha,int p,int k,int n)
{
    int adr;
    adr = SearchHT(ha,p,k);
    if (adr!=-1)   //在哈希表中找到关键字
    {
        ha[adr].key=DELKEY;
        n--;      //哈希表长度减1
        return n;
    }
    else           //在哈希表中未找到该关键字
        return n;
}

//输出哈希表
void DispHT(HashTable ha,int n,int m)
{
    float avg=0;
    int i;
    printf(" 哈希表地址:\t");
    for(i=0;i<m;i++)
        printf(" %3d",i);
    printf("\n");
    printf(" 哈希表关键字:\t");
    for(i=0;i<m;i++)
    {
        if(ha[i].key==NULLKEY || ha[i].key==DELKEY)
```

```
            printf("    "); //输出4个空格
        else
            printf(" %3d",ha[i].key);
    }
    printf(" \n");
    printf(" 搜索次数:\t");
    for(i=0;i<m;i++)
    {
        if(ha[i].key==NULLKEY || ha[i].key==DELKEY)
            printf("    "); //输出4个空格
        else
            printf(" %3d",ha[i].count);
    }
    printf("\n");
    for(i=0;i<m;i++)
        if(ha[i].key!=NULLKEY && ha[i].key!=DELKEY)
            avg=avg+ha[i].count;
    avg=avg/n;
    printf(" 平均搜索长度ASL(%d)=%.3g\n",n,avg);

}

//查找成功时，平均查找长度
void CompASL(HashTable ha,int m)
{
    int i;
    int s=0,n=0;
    for (i=0;i<m;i++)
    {
        if(ha[i].key!=DELKEY && ha[i].key!=NULLKEY)
        {
            s=s+ha[i].count;
            n++;
        }
    }
    printf(" 查找成功的ASL=%.3g\n",s*1.0/n);
}
int main()
{
    int x[]={16,74,60,43,54,90,46,31,29,88,77};
    int n=11;  //元素个数
    int m=13,p=13,i;
    HashTable ha;
```

```
    int k=29;  //关键字k
    printf("建立哈希表:\n");
    CreateHT(ha,x,n,m,p);  //建立哈希表
    DispHT(ha,n,m);        //输出哈希表

    i=SearchHT(ha,p,k);    //查找关键字

    printf("\n删除关键字%d\n",k);
    n=DeleteHT(ha,p,k,n);  //删除关键字
    DispHT(ha,n,m);

    i=SearchHT(ha,p,k);    //查找关键字

    printf("\n插入关键字%d\n",k);
    n=InsertHT(ha,n,k,p);  //插入关键字
    DispHT(ha,n,m);

    printf("\n");
}
/*
 建立哈希表:
  哈希表地址:      0   1   2   3   4   5   6   7   8   9  10  11  12
  哈希表关键字:     77      54  16  43  31  29  46  60  74  88      90
  搜索次数:      2       1   1   1   1   4   1   1   1   1       1
  平均搜索长度ASL(11)=1.36
   ha[6].key=29

 删除关键字29
   ha[6].key=29
  哈希表地址:      0   1   2   3   4   5   6   7   8   9  10  11  12
  哈希表关键字:     77      54  16  43  31      46  60  74  88      90
  搜索次数:      2       1   1   1   1       1   1   1   1       1
  平均搜索长度ASL(10)=1.1
   未找到29

 插入关键字29
  哈希表地址:      0   1   2   3   4   5   6   7   8   9  10  11  12
  哈希表关键字:     77      54  16  43  31  29  46  60  74  88      90
  搜索次数:      2       1   1   1   1   4   1   1   1   1       1
  平均搜索长度ASL(11)=1.36
 */
```

图·存储结构·邻接表·书

```
//
//  main.c
//  图·存储结构·邻接表·书
//
//  [   vertex   (顶点域)   | firstedg（边表头指针）          ]       [顶点结点]
//  [ adjvertex (邻接点域) |  next（指针域）                 ]       [表结点]
//  [ adjvertex (邻接点域) |  info(边上信息) | next(指针域) ]       [带权图的边表结构]


#include <stdio.h>
#include <stdlib.h>
#define MaxVertexNum 30  /*最大顶点数为 30*/

typedef char VertexType;
typedef int InfoType;


typedef struct node {   //表结点w
    int adjvertex;   //邻接点域,一般是存放顶点对应的序号或在表头向量中的下标
    InfoType info;   //与边(或弧)相关的信息
    struct node * next;   //指向下一个邻接点的指针域
}EdgeNode;

typedef struct vnode {  //顶点结点
    VertexType vertex;    //顶点域
    EdgeNode * firstedge;   //边表头指针
}VertexNode;

typedef struct {
    VertexNode adjlist[MaxVertexNum];   //邻接表
    int vertexNum,edgeNum;   //顶点数和边数
}ALGraph;  //ALGraph是以邻接表方式存储的图类型

ALGraph init_alraph(void)
{//开辟空间
    ALGraph *G;
    G=(ALGraph *)malloc(sizeof(ALGraph));
    if(G)
        return *G;
    exit(0);
}

void CreateALGraph(ALGraph *G)
{//建立 无向图G 的邻接表存储
```

```
    int i,j,k;
    EdgeNode *p,*s;
    printf("邻接表建立：\n");
    printf("  输入顶点数和边数:");
    scanf("%d%d",&(G->vertexNum),&(G->edgeNum));   //读人顶点数和边数
    printf("  输入%d个顶点信息:",G->vertexNum);
    getchar();
    for(i=0;i<G->vertexNum;i++)               //建立有n个顶点的顶点表
    {
        scanf("%c",&(G->adjlist[i].vertex));    //读人顶点信息 v0,v1,v2,v3
        G->adjlist[i].firstedge=NULL;     //顶点的边表头指针设为空
    }
    for(k=0;k<G->edgeNum;k++)            //建立边表
    {
        printf("  输入第%d个边表连接(0<=i,j<=%d)：",k+1,G->vertexNum-1);
        scanf("%d%d",&i,&j);         //读入边<Vi,Vj>的顶点对应序号
        p=(EdgeNode*)malloc(sizeof(EdgeNode));    //生成新边表结点p*
        p->adjvertex=j;          //邻接点序号为j
        p->next=G->adjlist[i].firstedge;   //将新边表结点 p插人到顶点 Vi 的链表头部
        G->adjlist[i].firstedge=p;

        //加入以下代码，为创建无向图
        s=(EdgeNode*)malloc(sizeof(EdgeNode));
        s->adjvertex = i;       //邻接点序号为i
        s->next = G->adjlist[j].firstedge;
        G->adjlist[j].firstedge = s;

    }
}

void show_ALgraph(ALGraph *G)
{//输出邻接表
    int i;
    EdgeNode *p;
    printf("创建的 无向图 邻接表 为：\n");
    for(i=0;i<G->vertexNum;i++)
    {
        printf(" <%c> ",G->adjlist[i].vertex);
        p=G->adjlist[i].firstedge;
        while (p) {
            printf("->%c",G->adjlist[p->adjvertex].vertex);
            p=p->next;
        }
        printf("\n");
```

```
    }
}
int main(int argc, const char * argv[])
{
    ALGraph G=init_alraph();
    CreateALGraph(&G);
    show_ALgraph(&G);
    return 0;
}

/*
 邻接表建立:
   输入顶点数和边数:8 9
   输入8个顶点信息:abcdefgh
   输入第1个边表连接(0<=i,j<=7): 0 1
   输入第2个边表连接(0<=i,j<=7): 0 2
   输入第3个边表连接(0<=i,j<=7): 1 3
   输入第4个边表连接(0<=i,j<=7): 1 4
   输入第5个边表连接(0<=i,j<=7): 3 7
   输入第6个边表连接(0<=i,j<=7): 4 7
   输入第7个边表连接(0<=i,j<=7): 2 5
   输入第8个边表连接(0<=i,j<=7): 2 6
   输入第9个边表连接(0<=i,j<=7): 5 6
 创建的 无向图 邻接表 为:
  <a> ->c->b
  <b> ->e->d->a
  <c> ->g->f->a
  <d> ->h->b
  <e> ->h->b
  <f> ->g->c
  <g> ->f->c
  <h> ->e->d
 */
```

图·存储结构·邻接矩阵·书

```
//
//  main.c
//  图·存储结构·邻接矩阵·书
//

#include <stdio.h>
#include <stdlib.h>
```

```c
#define MaxVertexNum 50

typedef char VertexType;
typedef int Edgetype;

typedef struct
{
    VertexType vertexs[MaxVertexNum];       //顶点向量
    Edgetype arcs[MaxVertexNum][MaxVertexNum];  //邻接矩阵
    int vertexnum,edgenum;                  //图的当前顶点数和边数
    //GraphType type;                     //图的种类标志
}MGraph;

MGraph init_mgraph(void)
{
    MGraph *G;
    G=(MGraph *)malloc(sizeof(MGraph));
    if(G)
        return *G;
    exit(0);
}

void creatgraph(MGraph *G)
{//建立 无向图G 的邻接矩阵存储
    int i,j,k;
    printf("邻接矩阵建立：\n");
    printf("  输入顶点数和边数：");
    scanf("%d %d",&(G->vertexnum),&(G->edgenum));   //输入顶点数和边数
    printf("  输入%d顶点信息:",G->vertexnum);
    getchar();
    for(i=0;i<G->vertexnum;i++)     //输入顶点信息，建立顶点表v0,v1,v2,v3
        scanf("%c",&(G->vertexs[i]));
    for(i=0;i<G->vertexnum;i++)       //初始化邻接矩阵
        for(j=0;j<G->vertexnum;j++)
            G->arcs[i][j]=0;
    for(k=0;k<G->edgenum;k++)     //输入e条边，建立邻接矩阵
    {
        printf("  输入第%d个点的坐标(0<=i,j<=%d)：",k+1,G->vertexnum-1);
        scanf("%d %d",&i,&j);     //输入为 1 的点坐标 (i,j)
        G->arcs[i][j]=1;

        G->arcs[j][i]=1;    //若加入此行，则为无向图的邻接矩阵建立
    }
}
```

```
void show_MGgraph(MGraph *G)
{//输出邻接矩阵
    int i,j;
    printf("创建的 无向图 邻接矩阵 为: \n");
    for(i=0;i<G->vertexnum;i++)
    {
        for(j=0;j<G->vertexnum;j++)
            printf(" %d",G->arcs[i][j]);
        printf("\n");
    }
}
int main(int argc, const char * argv[])
{
    MGraph G=init_mgraph();
    creatgraph(&G);
    show_MGgraph(&G);
    return 0;
}
/*
 邻接矩阵建立:
   输入顶点数和边数: 8 9
   输入8顶点信息:abcdefgh
   输入第1个点的坐标(0<=i,j<=7): 0 1
   输入第2个点的坐标(0<=i,j<=7): 0 2
   输入第3个点的坐标(0<=i,j<=7): 1 3
   输入第4个点的坐标(0<=i,j<=7): 1 4
   输入第5个点的坐标(0<=i,j<=7): 3 7
   输入第6个点的坐标(0<=i,j<=7): 4 7
   输入第7个点的坐标(0<=i,j<=7): 2 5
   输入第8个点的坐标(0<=i,j<=7): 2 6
   输入第9个点的坐标(0<=i,j<=7): 5 6
 创建的 无向图 邻接矩阵 为:
  0 1 1 0 0 0 0 0
  1 0 0 1 1 0 0 0
  1 0 0 0 0 1 1 0
  0 1 0 0 0 0 0 1
  0 1 0 0 0 0 0 1
  0 0 1 0 0 0 1 0
  0 0 1 0 0 1 0 0
  0 0 0 1 1 0 0 0
 */
```

图·有向·AOV 拓扑排序

```
//
//  main.c
//  图·加权有向无环·AOV拓扑排序
//
//    [   indegree  |   vertex  |  firstedge  ]

#include <stdio.h>
#include <stdlib.h>
#define MaxVertexNum 30  //最大顶点数为 30

typedef struct node  //表结点
{
    int adjvertex;  //邻接点域,一般是存放顶点对应的序号或在表头向量中的下标
    struct node * next;  //指向下一个邻接点的指针域
}EdgeNode;

typedef struct vnode  //顶点表结点
{
    int indegree;  //存放顶点人度
    int vertex;  //顶点域
    EdgeNode * firstedge;  //边表头指针
}VertexNode;

typedef struct
{
    VertexNode adjlist[MaxVertexNum];  //邻接表
    int vertexNum,edgeNum;  //顶点数和边数
}ALGraph;  //ALGraph 是以邻接表方式存储

ALGraph init_alraph(void)
{
    ALGraph *G;
    G=(ALGraph *)malloc(sizeof(ALGraph));
    if(G)
        return *G;
    exit(0);
}

void CreateALGraph(ALGraph *G)
{//建立 有向图G 的邻接表存储
    int i,j,k;
    EdgeNode *p;
    printf("有向图邻接表建立: \n");
    printf("  输入顶点数和边数:");
```

```
    scanf("%d%d",&(G->vertexNum),&(G->edgeNum));   //读人顶点数和边数
    printf("  输入%d个顶点信息(数字):\n",G->vertexNum);
    getchar();
    for(i=0;i<G->vertexNum;i++)              //建立有n个顶点的顶点表
    {
        scanf("%d",&(G->adjlist[i].vertex));    //读人顶点信息 v0,v1,v2,v3
        G->adjlist[i].firstedge=NULL;     //顶点的边表头指针设为空
    }
    for(k=0;k<G->edgeNum;k++)            //建立边表
    {
        printf("  输入第%d个边表连接(0<=i,j<=%d): ",k+1,G->vertexNum-1);
        scanf("%d%d",&i,&j);         //读入边<Vi,Vj>的顶点对应序号
        p=(EdgeNode*)malloc(sizeof(EdgeNode));    //生成新边表结点p*
        p->adjvertex=j;          //邻接点序号为j
        p->next=G->adjlist[i].firstedge;   //将新边表结点 p插人到顶点 Vi 的链表头部
        G->adjlist[i].firstedge=p;
    }
}

void FindInDegree(ALGraph *G)  //求各顶点的入度
{
    int i;
    EdgeNode *p;
    for(i=0;i<G->vertexNum;i++)
        G->adjlist[i].indegree=0;
    for(i=0;i<G->vertexNum;i++)
    {
        for(p=G->adjlist[i].firstedge;p;p=p->next)
            G->adjlist[p->adjvertex].indegree++;
    }
    for(i=0;i<G->vertexNum;i++)
        printf("第 %d 个顶点的入度为 %d \n",i+1,G->adjlist[i].indegree);
}
void Top_Sort (ALGraph G)
{//对以邻接链表为存储结构的图 G,输出其拓扑序列
    int i,j,k,count=0;
    int top=-1;  //栈顶指针初始化
    EdgeNode *p;
    FindInDegree(&G);  //求各顶点的入度
    for(i=0;i<G.vertexNum;i++)  //依次将入度为0的顶点压入链式栈
    {
        if(G.adjlist[i].indegree==0)
        {
            G.adjlist[i].indegree=top;
```

```
            top=i;
        }
    }
    printf("邻接链表的拓扑序列为：");
    while(top!=-1)  //栈不空
    {
        j=top;
        top=G.adjlist[top].indegree;  //从栈中退出一个顶点并输出
        printf(" %d",G.adjlist[j].vertex);
        count++;  //排序到的顶点计数
        for(p=G.adjlist[j].firstedge;p;p=p->next)
        {
            k=p->adjvertex;
            G.adjlist[k].indegree--;  //当前输出顶点邻接点的入度减1
            if(G.adjlist[k].indegree==0)  //新的入度为0的顶点进栈
            {
                G.adjlist[k].indegree=top;
                top=k;  //修改栈顶下标
            }
        }
    }
    if(count<G.vertexNum)
        printf("The network has a cycle");
}

void show_ALgraph(ALGraph *G)
{//输出邻接表
    int i;
    EdgeNode *p;
    printf("创建的 有向图 邻接表 为：\n");
    for(i=0;i<G->vertexNum;i++)
    {
        printf(" <%d> ",G->adjlist[i].vertex);
        p=G->adjlist[i].firstedge;
        while (p) {
            printf("->%d",G->adjlist[p->adjvertex].vertex);
            p=p->next;
        }
        printf("\n");
    }
}

int main()
{
```

```
    ALGraph G=init_alraph();
    CreateALGraph(&G);
    show_ALgraph(&G);
    Top_Sort(G);  //这里的拓扑排序可能有些问题，等一点时间，我会处理好
    printf("\n");
}
/*
 有向图邻接表建立:
   输入顶点数和边数:8 9
   输入8个顶点信息(数字）:
 1
 2
 3
 4
 5
 6
 7
 8
   输入第1个边表连接(0<=i,j<=7): 0 1
   输入第2个边表连接(0<=i,j<=7): 0 2
   输入第3个边表连接(0<=i,j<=7): 1 3
   输入第4个边表连接(0<=i,j<=7): 1 4
   输入第5个边表连接(0<=i,j<=7): 3 7
   输入第6个边表连接(0<=i,j<=7): 4 7
   输入第7个边表连接(0<=i,j<=7): 2 5
   输入第8个边表连接(0<=i,j<=7): 2 6
   输入第9个边表连接(0<=i,j<=7): 5 6
 创建的 有向图 邻接表 为:
  <1> ->3->2
  <2> ->5->4
  <3> ->7->6
  <4> ->8
  <5> ->8
  <6> ->7
  <7>
  <8>
 第 1 个顶点的入度为 0
 第 2 个顶点的入度为 1
 第 3 个顶点的入度为 1
 第 4 个顶点的入度为 1
 第 5 个顶点的入度为 1
 第 6 个顶点的入度为 1
 第 7 个顶点的入度为 2
 第 8 个顶点的入度为 2
```

```
邻接链表的拓扑序列为:  1 2 4 5 8 3 6 7

有向图邻接表建立:
  输入顶点数和边数:6 8
  输入6个顶点信息(数字):
0
1
2
3
4
5
  输入第1个边表连接(0<=i,j<=5): 0 1
  输入第2个边表连接(0<=i,j<=5): 0 2
  输入第3个边表连接(0<=i,j<=5): 0 3
  输入第4个边表连接(0<=i,j<=5): 2 1
  输入第5个边表连接(0<=i,j<=5): 2 4
  输入第6个边表连接(0<=i,j<=5): 3 4
  输入第7个边表连接(0<=i,j<=5): 3 5
  输入第8个边表连接(0<=i,j<=5): 5 4
创建的 有向图 邻接表 为:
 <0> ->3->2->1
 <1>
 <2> ->4->1
 <3> ->5->4
 <4>
 <5> ->4
第 1 个顶点的入度为 0
第 2 个顶点的入度为 2
第 3 个顶点的入度为 1
第 4 个顶点的入度为 1
第 5 个顶点的入度为 3
第 6 个顶点的入度为 1
邻接链表的拓扑序列为:  0 2 1 3 5 4
*/
```