 #include <stdio.h> // 栈和队列实现图(邻接表)的深度和广度优先搜索

#include <stdlib.h>

#define MaxVex 100

int visited[MaxVex]; // 全局数组，用于记录图中节点访问状态

typedef struct EdgeNode { // 邻接表节点

int adjvex; // 该邻接点在顶点数组中的下标

struct EdgeNode \*next; // 指向下一个邻接点

}EdgeNode;

typedef struct VertexNode { // 顶点节点（图结点）

char data; // 顶点信息

EdgeNode \*firstedge; // 邻接表头指针(指向第一条依附于该顶点的弧的指针)

}VertexNode,AdjList[MaxVex]; // 顶点数组(结构体数组)

typedef struct Graph{ // 图

AdjList adjList; // 存顶点的数组

int numVertexes,numEdges; // 图中当前的结点数以及边数

}Graph,\*GraphAdjList;

void CreateALGraph(GraphAdjList &G) // 建立无向图的邻接表结构

{

test data

7 7

A

C

E

B

D

F

G

0 1

0 2

1 3

2 4

2 5

4 6

5 6

if(G==NULL)

G = (GraphAdjList)malloc(sizeof(Graph));

printf("输入图的结点数以及边数: ");

scanf("%d%d",&G->numVertexes,&G->numEdges);

fflush(stdin);

printf("===========================\n");

printf("输入各个顶点的数据:\n");

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

printf("顶点%d: ",i);

scanf("%c", &(G->adjList[i].data)); // 将顶点数据放入数据域

G->adjList[i].firstedge = NULL; // 边表头指针初始为NULL

fflush(stdin);

}

printf("===========================\n");

int j;

for (int k=0; k<G->numEdges; ++k){ // 输入边的信息并与顶点建立联系

printf("输入(vi,vj)上的顶点序号: ");

scanf("%d %d",&i,&j);

EdgeNode \*ptrEdgeNode = (EdgeNode\*)malloc(sizeof(EdgeNode));

next

j

NULL

next

j

data

ptrEdgeNode->adjvex = j; // 边节点数据域存顶点下标

ptrEdgeNode->next = G->adjList[i].firstedge; // 表头后面插入边节点

G->adjList[i].firstedge = ptrEdgeNode;

firstEdge

ptrEdgeNode = (EdgeNode\*)malloc(sizeof(EdgeNode));

ptrEdgeNode->adjvex = i; // 无向图再进行一次相反操作

ptrEdgeNode->next = G->adjList[j].firstedge;

G->adjList[j].firstedge = ptrEdgeNode;

}

}

1 图和数据结构

|  |  |  |
| --- | --- | --- |
| 图（前面的数字代表顶点序列，  后面的字母表示顶点对应的数据）  0A  / \  1C 2E  / / \  3B 4D 5F  \ /  6G | 图用邻接表表示：    A-->2-->1  C-->3-->0  E-->5-->4-->0  B-->1  D-->6-->2  F-->6-->2  G-->5-->4 |  |

2堆栈定义及相关操作(深度优先遍历会用到此栈)

栈，后进先出，栈顶弹出一个节点，并在栈顶压入该节点的邻接点（有去重）

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | 4D |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | 6G | 6G | 6G |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | 5F | 5F | 5F | 5F | 5F |  |  |  |  |  |  |  |  | F | G |  |  |  |  |  |  |  |  |  |  |  |
|  | 2E | 2E | 2E | 2E | 2E | 2E | 2E |  | 1C | 3B |  |  |  | E | D | D | D |  |  |  |  | C | B |  | D | G |  |
| 0A | 0A | 0A | 0A | 0A | 0A | 0A | 0A | 0A | 0A | 0A | 0A |  | A | C | C | C | C | C | B |  | A | E | E | E | F | F | F |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 0A | 2E | 5F | 6G | 4D |  |  |  |  | 1C | 3B |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

前进就压入节点，回溯就弹出节点。称为深度优先搜索。

int Stack[MaxVex]; // 自定义栈

int Stackcount=-1; // 堆栈指针

int StackEmpty(){ // 判断栈空

return Stackcount==-1;

}

int StackFull(){ // 判断栈满

return Stackcount==MaxVex-1;

}

void Push(int e){ // 入栈

if(!StackFull())

Stack[++Stackcount]=e;

else

printf("Full");

}

void Pop(){ // 出栈

if(!StackEmpty())

Stackcount--;

else

printf("Empty");

}

3 深度优先遍历(堆栈实现)

void DFSTraverseStack(GraphAdjList &G){ //深度优先遍历(堆栈实现)

for (int i=0; i<G->numVertexes; ++i)

visited[i]=0; // 初始化访问状态

i=0; // 从i号顶点开始遍历

visited[i] = 1;

printf("%c ", G->adjList[i].data);

Push(i); // 将起始节点进栈，以便将来正确返回

while(!StackEmpty())

{

EdgeNode \*p=G->adjList[Stack[Stackcount]].firstedge; // 指向栈顶元素的邻接表头

while(p)

{

if(!visited[p->adjvex]) // 若当前邻接顶点没有被访问过，则进行访问并入栈

{

printf("%c ",G->adjList[p->adjvex].data);

visited[p->adjvex]=1;

Push(p->adjvex); // 访问顶点进栈

break; // 邻接表是横向的，不横向探索，因为有入栈，可以在循环中访问下一个元素

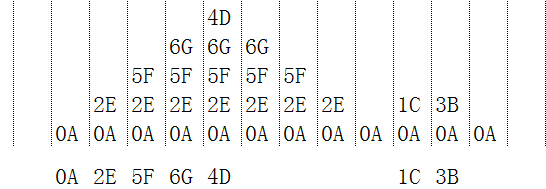
}

else // 若当前邻接顶点已经被访问过，则沿边找到下一个顶点

p=p->next;

}

if(p==NULL) // 若某一方向被访问完，则回溯寻找未被访问的顶点

 Pop();

}

}

4 递归深度优先遍历(递归实现)

void DFSRecursion(GraphAdjList &G, int i) // DFSTraverseRecursion()的子函数

{

visited[i] = 1; // 改变访问状态

printf("%c ",G->adjList[i].data); // 输出顶点

EdgeNode \*p=G->adjList[i].firstedge;

while(p){

if(!visited[p->adjvex]) // 若节点尚未访问

DFSRecursion (G,p->adjvex); // 递归深度遍历

p=p->next; // 边节点指针后移

}

}

void DFSTraverseRecursion(GraphAdjList &G)

{

for (int i=0; i<G->numVertexes;++i)

visited[i] = 0; // 初始化访问数组visited的元素值为0

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

if(!visited[i]) // 节点尚未访问

DFSRecursion (G,i); // 调用遍历函数

}

5 队列定义及相关操作(广度优先遍历会用到此循环队列)

队列，后进后出，头部弹出一个节点，并在尾部加入该节点的邻接点（有去重）

|  |  |  |
| --- | --- | --- |
| 0A  / \  1C 2E  / / \  3B 4D 5F  \ /  6G | front-------rear  A  ----------------  E C  ----------------  C F D  ----------------  F D B  ----------------  D B  ----------------  B G  ----------------  G  ---------------- | int Queue[MaxVex]; // 实现顺序队列  int front=0,rear=0; // 队头和队尾指针  int QueueEmpty(){  return front == rear;  }  int QueueFull(){  return rear == MaxVex-1;  }  void EnQueue(int e){ // 队尾插入元素  if(!QueueFull())  Queue[rear++] = e;  }  void DeQueue(int \*e){ // 队头删除元素  if(!QueueEmpty())  \*e = Queue[front++];//访问区间[front,rear]  } |

6广度优先搜索(使用队列)

void BFSTraverse(GraphAdjList &G) // 图的广度优先遍历

{

for (int i=0; i<G->numVertexes; ++i)

visited[i] = 0; // 初始化访问状态

i=0; // 从i号顶点开始遍历

visited[i] = 1;

printf("%c ", G->adjList[i].data);

EnQueue(i);

while (!QueueEmpty())

{

DeQueue(&i);

EdgeNode \*p = G->adjList[i].firstedge; // 指向队头元素的邻接表头

while (p)

{

if (!visited[p->adjvex]) // 若当前邻接顶点没有被访问过，则进行访问并入队

{

visited[p->adjvex] = 1;

printf("%c ", G->adjList[p->adjvex].data);

EnQueue(p->adjvex);

}

p = p->next; // 访问下一个相连的顶点

}

}

}

void ShowVlist(GraphAdjList &G) // 邻接表输出

{

int i;

EdgeNode\* curp;

printf("===========================\n邻接表输出:\n");

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

{

printf("\%c",G->adjList[i].data);

curp=G->adjList[i].firstedge; // 边节点指针指向第一个边节点

while(curp!=NULL)

{

printf("-->%d",curp->adjvex);

curp=curp->next; // 依次往后遍历

}

printf("\n");

}

}

int main()

{

GraphAdjList G = NULL;

CreateALGraph(G); // 创建邻接表

ShowVlist(G); // 输出邻接表

printf("\n图的深度优先遍历(堆栈实现)为:\t");

DFSTraverseStack(G);

printf("\n图的深度优先遍历(递归实现)为:\t");

DFSTraverseRecursion(G);

printf("\n图的广度优先遍历为:\t\t");

BFSTraverse(G);

printf("\n\n");

system("pause");

return 0;

}

图的深度优先遍历(堆栈实现)为: A E F G D C B

图的深度优先遍历(递归实现)为: A E F G D C B

图的广度优先遍历为: A E C F D B G

深度、广度优先搜索

无论是深度还是广度搜索，都是通过对一个线性表进行处理，只不过是先处理头部还是尾部的问题罢了。

处理头部优先的时候，也就是先加入的先探索，就是广度优先了，因为，头部的都是兄弟节点；

而尾部的则是深度优先，因为放入尾部的都是刚刚生产出来的节点，

后加入的先探索——也就是所谓一条路走到底。 同理可以联想到启发式搜索。

启发式搜索就是先以你自定义的优先级处理，然后再以广度为优先级处理。

所以，归根结底，所谓的搜索，就是一种定义了优先级的枚举。