```c
#define _CRT_SECURE_NO_WARNINGS 1
#include <stdio.h>
#include <stdlib.h>
#define StackSize 100
typedef int ElemType;
typedef struct
{
    ElemType data[StackSize];
    int top;
}SqStack;
typedef struct {
        float data[StackSize];
        int top;
}OpStack;
void InitStack(SqStack *S)
{S->top = -1;}
int StackEmpty(SqStack S)
{
    if(S.top==-1){
       // printf("栈已经空\n");
        return 1;
    }
    else return 0;
}
void Push(SqStack *s,ElemType e)
{
    if(s->top==StackSize-1)
        return;
    s->top=s->top+1;
    s->data[s->top]=e;
}
void Pop(SqStack *s,ElemType *e)
{
    if(s->top==-1)
        return;
    *e=s->data[s->top];
    s->top=s->top-1;
}
//取栈顶元素
int GetTop(SqStack S, ElemType *e) {
        if (!StackEmpty(S)){
                *e = S.data[S.top];
                return 1;}
    return 0;
}
// 后缀表达式
void Tohouzhui(SqStack *S,char input[],char output[])
{
    char p;
    ElemType e;
    int i=0,j=0;
    p = input[i++];
    while(p!='\0')
    {
        switch (p)
        {
        case '(':
            Push(S,p);
            break;
        case ')':         //取栈顶元素
                        while (GetTop(*S, &e) && e != '(')
                        {
                                Pop(S, &e);
                                output[j] = e;
                                j++;
                        }
                        Pop(S, &e);//当遇到（时，将符号插入栈中；当遇到）时，将（取出栈
                        break;
        case '+':
                case '-':
                        while (!StackEmpty(*S) && GetTop(*S, &e) && e != '(')
                        {
                                Pop(S, &e);
                                output[j] = e;
                                j++;
                        }
                        Push(S, p);//将当前运算符进栈
                        break;
                case '*':
                case '/':
                        while (!StackEmpty(*S) && GetTop(*S, &e) && e == '/' || e == '*')
                        {
                                Pop(S, &e);
                                output[j] = e;
                                j++;
                        }
                        Push(S, p);//将当前运算符进栈
                        break;
                case ' ':
                        break;
                default:
                        while (p >= '0' && p <= '9')
                        {
                                output[j] = p;
                                j++;
                                p = input[i];
                                i++;
```

```
                        }
                        i--;
                        output[j] = ' ';
                        j++;
                }
                p = input[i];
                i++;
        }
        while (!StackEmpty(*S))
        {
                Pop(S, &e);
                output[j] = e;
                j++;
        }
        output[j] = '\0';
}
//计算后缀表达式
float Caculate(char a[]) {
        OpStack S;
        int i = 0, value;
        float x1, x2;
        float result;
        S.top = -1;
        while (a[i] != '\0')
        {
                if (a[i] != ' ' && a[i] >= '0' && a[i] <= '9')
                {
                        value = 0;
                        while (a[i] != ' ')
                        {
                                value = 10 * value + a[i] - '0';
                                i++;
                        }
                        S.top++;
                        S.data[S.top] = value;
                }
                else
                {
                        switch (a[i])
                        {
                        case '+':
                                x1 = S.data[S.top];
                                S.top--;
                                x2 = S.data[S.top];
                                S.top--;
                                result = x1 + x2;
                                S.top++;
                                S.data[S.top] = result;
                                break;
                        case '-':
                                x1 = S.data[S.top];
                                S.top--;
                                x2 = S.data[S.top];
                                S.top--;
                                result = x2 - x1;
                                S.top++;
                                S.data[S.top] = result;
                                break;
                        case '*':
                                x1 = S.data[S.top];
                                S.top--;
                                x2 = S.data[S.top];
                                S.top--;
                                result = x1 * x2;
                                S.top++;
                                S.data[S.top] = result;
                                break;
                        case '/':
                                x1 = S.data[S.top];
                                S.top--;
                                x2 = S.data[S.top];
                                S.top--;
                                result = x2 / x1;
                                S.top++;
                                S.data[S.top] = result;
                                break;
                        }
                        i++;
                }
        }
}
int main(void)
{
   float result;
   SqStack S;
   InitStack(&S);
   char Orignal[StackSize]={0};
   char Out[StackSize]={0};
   printf("请输入计算式：");
   scanf("%s",&Orignal);
   printf("%s\n",Orignal);
   printf("后缀表达式：");
   Tohouzhui(&S,Orignal,Out);
   printf("%s\n",Out);
   result = Caculate(Out);
   printf("结果是：%f",result);
   return 0;
}
```

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

typedef struct {
    int arrive; //到达时间
    int treat;  //需要办理业务的时间
}ElemType;

typedef struct LinkNode {
    ElemType data;
    struct LinkNode* next;//队列中的元素信息
}LinkNode;

typedef struct  {
    LinkNode* front,*rear;//队列中的元素信息
}LinkQueue;

void InitQueue(LinkQueue *Q)
{
    Q->rear=Q->front=(LinkNode*)malloc(sizeof(LinkNode));
    Q->front->next=NULL;
}

//入队
void EnQueue(LinkQueue *Q,ElemType e)
{
    LinkNode *s = (LinkNode*)malloc(sizeof(LinkNode));
    if (!s)
        exit(0);// 存储分配失败退出
    s->data = e;
    s->next = NULL;
    Q->rear->next = s;
    Q->rear = s;                // rear 指向 s 队尾
}

//出队
int DeQueue(LinkQueue *Q,ElemType *e)
{
    if (Q->front == Q->rear)
        return 0;
    LinkNode *P = Q->front->next;\
    *e = P->data;
    Q->front->next=P->next;
    if(Q->rear==P)
    {
        Q->rear=Q->front;
    }
    free(P);
    return 1;
}

int main()
{
    int sever_wait = 0, client_wait = 0;//sever_wait 业务员等待时间。client_wait 为客户等待时间
    int clock = 0;
    int number = 0;//用于积累客户人数
    int exist = 0;//exist 用于判断是否有未处理的客户 非 0 为存在 0 为不存在

    ElemType temp, e;

    //建造空队列
    LinkQueue Q;
    InitQueue(&Q);

    //txt 文件并打入 '10 20 23 10 45 5'
    FILE* fp = fopen("E:\\Desktop\\queue.txt", "r");
    if (fp == NULL) {
        printf("文件打开失败");
        return 0;
    }

    exist = fscanf(fp, "%d %d", &temp.arrive, &temp.treat);

    do {
        if (exist == 2 && Q.front == Q.rear)//有客户&&队列为空
        {
            sever_wait += (temp.arrive - clock);
            clock = temp.arrive;
            EnQueue(&Q,temp);
            exist = fscanf(fp, "%d %d", &temp.arrive, &temp.treat);
        }
        number++;//积累客户人数+1
        DeQueue(&Q,&e);
        client_wait += (clock - e.arrive);
        clock += e.treat;//时间推进到客户结束时间
        while (temp.arrive <= clock && exist == 2) //上一个没结束，有到达的客户就入队
        {
            EnQueue(&Q,temp);
            exist = fscanf(fp, "%d %d", &temp.arrive, &temp.treat);
        }
    } while (exist == 2 || Q.front != Q.rear);//有未处理的客户&&队列不为空
    printf("业务员等待时间为%d\n客户平均等待时间为%f", sever_wait, (float)client_wait / (float)number);
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
typedef char DataType;
typedef struct Node {
    DataType data; /* 数据域 */
    struct Node* leftChild; /* 左子树指针 */
    struct Node* rightChild; /* 右子树指针 */
} BiTreeNode; /* 结点的结构体定义 */


/*初始化创建二叉树的头结点*/
void Initiate(BiTreeNode** root) {
    *root = (BiTreeNode*)malloc(sizeof(BiTreeNode));
    (*root)->leftChild = NULL;
    (*root)->rightChild = NULL;
}
void Destroy(BiTreeNode** root)
{
    if ((*root) != NULL && (*root)->leftChild != NULL)
        Destroy(&(*root)->leftChild);
    if ((*root) != NULL && (*root)->rightChild != NULL)
        Destroy(&(*root)->rightChild);
    free(*root);
}
/*若当前结点 curr 非空，在 curr 的左子树插入元素值为 x 的新结点*/
/*原 curr 所指结点的左子树成为新插入结点的左子树*/
/*若插入成功返回新插入结点的指针，否则返回空指针*/
BiTreeNode* InsertLeftNode(BiTreeNode* curr, DataType x)
{
    BiTreeNode* s, * t;
    if (curr == NULL) return NULL;
    t = curr->leftChild;/*保存原 curr 所指结点的左子树指针*/
    s = (BiTreeNode*)malloc(sizeof(BiTreeNode));
    s->data = x;
    s->leftChild = t;/*新插入结点的左子树为原 curr 的左子树*/
    s->rightChild = NULL;

    curr->leftChild = s;/*新结点成为 curr 的左子树*/
    return curr->leftChild;/*返回新插入结点的指针*/
}
/*若当前结点 curr 非空，在 curr 的右子树插入元素值为 x 的新结点*/
/*原 curr 所指结点的右子树成为新插入结点的右子树*/
/*若插入成功返回新插入结点的指针，否则返回空指针*/
BiTreeNode* InsertRightNode(BiTreeNode* curr, DataType x)
{
    BiTreeNode* s, * t;
    if (curr == NULL) return NULL;
    t = curr->rightChild;/*保存原 curr 所指结点的右子树指针*/
    s = (BiTreeNode*)malloc(sizeof(BiTreeNode));
    s->data = x;
    s->rightChild = t;/*新插入结点的右子树为原 curr 的右子树*/
    s->leftChild = NULL;
    curr->rightChild = s;/*新结点成为 curr 的右子树*/
    return curr->rightChild;/*返回新插入结点的指针*/
}
/* 前序遍历 */
void PreOrder(BiTreeNode* root, void (*visit)(DataType)) {
    if (root != NULL) {
        visit(root->data);
        PreOrder(root->leftChild, visit);
        PreOrder(root->rightChild, visit);
    }
}
/* 中序遍历 */
void InOrder(BiTreeNode* root, void (*visit)(DataType)) {
    if (root != NULL) {
        InOrder(root->leftChild, visit);
        visit(root->data);
        InOrder(root->rightChild, visit);
    }
}
/* 后序遍历 */
void PostOrder(BiTreeNode* root, void (*visit)(DataType)) {
    if (root != NULL) {
        PostOrder(root->leftChild, visit);
        PostOrder(root->rightChild, visit);
        visit(root->data);
    }
}
/* 查询特定元素 */
int Search(BiTreeNode* root, DataType x) {
    if (root == NULL) {
        return 0; // 未找到
    }
    if (root->data == x) {
        return 1; // 找到
    }
    return Search(root->leftChild, x) | Search(root->rightChild, x);
}
/* 用于打印节点数据的辅助函数 */
void PrintData(DataType data) {
    printf("%c ", data);
}
// 主函数，测试上述功能
int main() {
    BiTreeNode* root = NULL;
    // 初始化根节点
    Initiate(&root);
```

```
    root->data = ' ';
    // 插入 A
    BiTreeNode* A = InsertLeftNode(root, 'A');
    // 插入 B 和 C
    BiTreeNode* B = InsertLeftNode(A, 'B');
    BiTreeNode* C = InsertRightNode(A, 'C');
    // 插入 D
    BiTreeNode* D = InsertLeftNode(B, 'D');
    // 插入 E 和 F
    BiTreeNode* E = InsertLeftNode(C, 'E');
    BiTreeNode* F = InsertRightNode(C, 'F');
    // 插入 G
    BiTreeNode* G = InsertRightNode(D, 'G');
    // InsertLeftNode(root, 'A');
    // InsertLeftNode(root->leftChild, 'B');
    // InsertRightNode(root->leftChild, 'C');
    // InsertLeftNode(root->leftChild->leftChild, 'D');
    // InsertLeftNode(root->leftChild->rightChild, 'E');
    // InsertRightNode(root->leftChild->rightChild, 'F');
    // InsertRightNode(root->leftChild->leftChild->leftChild, 'G');
    printf("         根\n");
    printf("        /\n");
    printf("       A\n");
    printf("      / \\\n");
    printf("     B   C\n");
    printf("    /   / \\\n");
    printf("   D   E   F\n");
    printf("    \\\n");
    printf("     G\n");
    // 前序遍历
    PreOrder(root, PrintData);
    // 中序遍历
    InOrder(root, PrintData);
    // 后序遍历
    PostOrder(root, PrintData);
    // 搜索特定元素
    char searchItem = 'H';
    if (Search(root, searchItem)) {
        printf("Element '%c' found in the tree.\n", searchItem);
    }
    else {
        printf("Element '%c' not found in the tree.\n", searchItem);
    }
    // 释放树的内存 Destroy(&root);
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_VEX 100      // 最大顶点数
#define INFINITY 65535    // 用 65535 来代表无穷大
#define LENGTH(a) (sizeof(a) / sizeof(a[0]))
typedef struct {
    char vexs[MAX_VEX];           // 顶点集合
    int matrix[MAX_VEX][MAX_VEX];  // 邻接矩阵
    int vexnum;                   // 图的顶点数
    int edgnum;                   // 图的边数
} MGraph;
// 获取顶点在数组中的位置
int getPosition(MGraph* G, char ch) {
    for (int i = 0; i < G->vexnum; i++) {
        if (G->vexs[i] == ch)
            return i;
    }
    return -1;
}
// 创建无向图的邻接矩阵
void createMGraph_wuxiang(MGraph* G) {
    char vexs[] = { '0', '1', '2', '3', '4', '5', '6', '7' };
    char edges[][2] = {
        {'0', '1'}, {'0', '2'}, {'1', '3'}, {'1', '4'},
        {'2', '5'}, {'2', '6'}, {'3', '7'}, {'4', '7'},
        {'5', '6'}
    };
    int vlen = LENGTH(vexs);
    int elen = LENGTH(edges);
    G->vexnum = vlen;
    G->edgnum = elen;
    // 初始化顶点
    for (int i = 0; i < G->vexnum; i++) {
        G->vexs[i] = vexs[i];
    }
    // 初始化邻接矩阵
    for (int i = 0; i < G->vexnum; i++) {
        for (int j = 0; j < G->vexnum; j++) {
              G->matrix[i][j] = 0;
        }
    }
    // 填充邻接矩阵
    for (int i = 0; i < elen; i++) {
        int start = getPosition(G, edges[i][0]);
        int end = getPosition(G, edges[i][1]);

        if (start != -1 && end != -1) {
            G->matrix[start][end] = 1;  // 有边为 1
            G->matrix[end][start] = 1;  // 无向图，需要填充两次
        }
    }
}
void createMGraph_youxiang(MGraph* G) {
    char vexs[] = { 'A', 'B', 'C', 'D', 'E', 'F', 'G' };
    char edges[][2] = {
        { 'A', 'B' },
        { 'B', 'C' },
        { 'B', 'E' },
        { 'B', 'F' },
        { 'C', 'E' },
        { 'D', 'C' },
        { 'E', 'B' },
        { 'E', 'D' },
        { 'F', 'G' }
    };
    int vlen = LENGTH(vexs);
    int elen = LENGTH(edges);
    G->vexnum = vlen;
    G->edgnum = elen;
    // 初始化顶点
    for (int i = 0; i < G->vexnum; i++) {
        G->vexs[i] = vexs[i];
    }
    // 初始化邻接矩阵
    for (int i = 0; i < G->vexnum; i++) {
        for (int j = 0; j < G->vexnum; j++) {
              G->matrix[i][j] = 0;
        }
    }
    // 填充邻接矩阵
    for (int i = 0; i < elen; i++) {
        int start = getPosition(G, edges[i][0]);
        int end = getPosition(G, edges[i][1]);

        if (start != -1 && end != -1) {
            G->matrix[start][end] = 1;  // 只设置有向边
        }
    }
}
// 打印图的邻接矩阵
void printMGraph(MGraph* G) {
    for (int i = 0; i < G->vexnum; i++) {
        for (int j = 0; j < G->vexnum; j++) {
            if (G->matrix[i][j] == INFINITY)
                printf("%7s", "INF");
            else
                printf("%7d", G->matrix[i][j]);
```

```
        }
        printf("\n");
    }
}
void DFS(MGraph* G, int i, int* visited) {
    visited[i] = 1;  // 标记为已访问
    printf("%c ", G->vexs[i]);

    for (int j = 0; j < G->vexnum; j++) {
        if (G->matrix[i][j] == 1 && !visited[j])
            DFS(G, j, visited);
    }
}
void DFSTraverse(MGraph* G) {
    int visited[MAX_VEX] = { 0 };

    for (int i = 0; i < G->vexnum; i++) {
        if (!visited[i])
            DFS(G, i, visited);
    }
}
void BFS(MGraph* G, int start) {
    int visited[MAX_VEX] = { 0 };
    int queue[MAX_VEX];
    int front = 0, rear = 0;
    printf("%c ", G->vexs[start]);
    visited[start] = 1;
    queue[rear++] = start;
    while (front != rear) {
        int i = queue[front++];
        for (int j = 0; j < G->vexnum; j++) {
            if (G->matrix[i][j] == 1 && !visited[j]) {
                printf("%c ", G->vexs[j]);
                visited[j] = 1;
                queue[rear++] = j;
            }
        }
    }
}
int main() {
    MGraph G;
    //createMGraph_wuxiang(&G);
    createMGraph_youxiang(&G);
    printf("Graph's Adjacency Matrix:\n");
    printMGraph(&G);
    BFS(&G, 0); //BFS
    DFSTraverse(&G); //DFS
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#define MAX 100
#define isLetter(a) ((((a) >= 'a')&&((a)<='z'))||(((a) >= 'A')&& ((a) <='Z')))
#define LENGTH(a) (sizeof(a) / sizeof(a[0]))
//邻接表中表对应的链表的顶点
typedef struct ENode
{
	int ivex;//该边所指向的顶点的位置
	struct ENode* next_edge;//指向下一条弧的指针
}ENode, * PENode;
// 邻接表中表的顶点
typedef struct _VNode
{
	char data;
	ENode* first_edge;
}VNode;
// 邻接表
typedef struct _LGraph
{
	int vexnum;    //顶点数
	int edgnum; //边数
	VNode vexs[MAX];
}LGraph;
static int getPosition(LGraph G, char ch)
{
	int i;
	for(i = 0; i < G.vexnum; i++)
		if(G.vexs[i].data == ch)
			return i;
	return -1;
}
static void linkLast(ENode* list, ENode* node)
{
	ENode* p = list;
	while(p->next_edge)
		p = p->next_edge;
	p->next_edge = node;
}
//创建邻接表
LGraph* CreateGraph()
{
	char c1, c2;
	// 有向图
	 char vexs[] = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
	 char edges[][2] = {
		{ 'A', 'B' },
		{ 'B', 'C' },
		{ 'B', 'E' },
		{ 'B', 'F' },
		{ 'C', 'E' },
		{ 'D', 'C' },
		{ 'E', 'B' },
		{ 'E', 'D' },
		{ 'F', 'G' } };
	int vlen = LENGTH(vexs);
	int elen = LENGTH(edges);
	int i, p1, p2;
	ENode* node1;
	LGraph* pG;
	if((pG = (LGraph*)malloc(sizeof(LGraph))) == NULL)
		return NULL;
	memset(pG, 0, sizeof(LGraph));
	//初始化顶点数和边数
	pG->vexnum = vlen;
	pG->edgnum = elen;
	//初始化邻接表顶点
	for(i = 0; i < pG->vexnum; i++)
	{
		pG->vexs[i].data = vexs[i];
		pG->vexs[i].first_edge = NULL;
	}
	//初始化邻接表的边
	for(i = 0; i < pG->edgnum; i++)
	{
		//读取边的起始顶点和结束顶点
		c1 = edges[i][0];
		c2 = edges[i][1];

		p1 = getPosition(*pG, c1);
		p2 = getPosition(*pG, c2);

		//初始化 node1
		node1 = (ENode*)calloc(1,sizeof(ENode));
		node1->ivex = p2;
		//将 node1 链接到"p1 所在链表的末尾"
		if(pG->vexs[p1].first_edge == NULL)
			pG->vexs[p1].first_edge = node1;
		else
			linkLast(pG->vexs[p1].first_edge, node1);
	}
	return pG;
}
//打印邻接表图
void printLGraph(LGraph* pG)
```

```
{
	int i, j;
	ENode* node;

	printf("List Graph:\n");
	for(i = 0; i < pG->vexnum; i++)
	{
		printf("%d(%c): ", i, pG->vexs[i].data);
		node = pG->vexs[i].first_edge;
		while(node != NULL)
		{
			printf("%d(%c) ", node->ivex, pG->vexs[node->ivex].data);
			node = node->next_edge;
		}
		printf("\n");
	}
}
//邻接表的广度优先遍历
void BFS(LGraph G)
{
	int head = 0;
	int rear = 0;
	int queue[MAX];
	int visited[MAX];	//顶点访问标记
	int i, j, k;
	ENode* node;

	for ( i = 0; i < G.vexnum; i++)
		visited[i] = 0;
	printf("BFS:");

	for ( i = 0; i < G.vexnum; i++)
	{
		if(!visited[i])
		{
			visited[i] = 1;
			printf("%c", G.vexs[i].data);
			queue[rear++] = i;	//入队列
		}
		while(head != rear)
		{
			j = queue[head++];	//出队列
			node = G.vexs[j].first_edge;
			while(node != NULL)
			{
				k = node->ivex;
				if(!visited[k])
				{
					visited[k] = 1;
					printf("%c", G.vexs[k].data);
					queue[rear++] = k;
				}
				node = node->next_edge;
			}
		}
	}
	printf("\n");
}
//邻接表的深度优先遍历
static void DFS(LGraph G, int i, int* visited)
{
	ENode* node;
	visited[i] = 1;
	printf("%c", G.vexs[i].data);
	node = G.vexs[i].first_edge;
	while(node != NULL)
	{
		if(!visited[node->ivex])
			DFS(G, node->ivex, visited);
		node = node->next_edge;
	}
}
void DFSTraverse(LGraph G)
{
	int i;
	int visited[MAX];	//顶点访问标记

	for ( i = 0; i < G.vexnum; i++)
		visited[i] = 0;
	printf("DFS:");
	for ( i = 0; i < G.vexnum; i++)
	{
		if(!visited[i])
			DFS(G, i, visited);
	}
	printf("\n");
}
int main(void)
{
	LGraph* pG;

	pG = CreateGraph();
	printLGraph(pG);
	BFS(*pG);
	DFSTraverse(*pG);
	return 0 ;
}
```