

河海大学物联网工程学院

课程实验报告

题 目	<u>数据结构实验报告</u>
学 号	<u>1762910211</u>
授课班号	<u>6241026</u>
学生姓名	<u>黄家名</u>
指导教师	<u>吴云燕</u>

目录

一、一元多项式的表示和相加	3
1.1 任务要求	3
1.2 基本原理	3
1.3 算法思路	3
1.4 数据结构	3
1.5 流程图	4
1.5.1 总体功能流程	4
1.5.2 多项式输出算法流程	5
1.6 源代码	5
1.6.1 多项式创建	5
1.6.2 多项式的销毁	6
1.6.3 多项式输出	6
1.6.4 多项式求长度	7
1.6.5 多项式求和	8
1.7 运行界面	9
1.8 收获	9
二、后缀表达式	10
2.1 任务要求	10
2.2 基本原理	10
2.3 算法思路	10
2.4 数据结构	10
2.5 流程图	11
2.5.1 总体功能流程	11
2.5.2 中缀转后缀	12
2.5.3 后缀求值	13
2.6 源代码	13
2.6.1 创建栈	13
2.6.2 销毁栈	14
2.6.3 栈中添加元素	14
2.6.4 栈中取出元素	14
2.6.5 栈中删除元素	14
2.6.6 栈长度	14
2.6.7 栈是否为空	14
2.6.8 求后缀表达式的值	15
2.6.9 中缀转后缀	15
2.6.10 其他小函数	16
2.7 运行界面	18
2.8 收获	18
三、赫夫曼树	19

3.1 任务要求.....	19
3.2 基本原理.....	19
3.3 算法思路.....	19
3.4 数据结构.....	19
3.5 流程图.....	20
3.5.1 构建赫夫曼树.....	20
3.6 源代码.....	20
3.6.1 构造赫夫曼树.....	20
3.6.2 赫夫曼编码.....	21
3.6.3 赫夫曼译码.....	22
3.6.4 选权值最小两结点.....	22
3.6.5 菜单函数.....	23
3.7 运行界面.....	24
3.8 收获.....	24
四、关键路径	25
4.1 任务要求.....	25
4.2 基本原理.....	25
4.3 算法思路.....	25
4.4 数据结构.....	26
4.5 流程图.....	26
4.5.1 总体功能流程.....	26
4.5.2 拓扑排序.....	27
4.5.3 关键路径.....	27
4.6 源代码.....	28
4.6.1 创建邻接矩阵.....	28
4.6.2 打印邻接矩阵.....	28
4.6.3 拓扑排序.....	28
4.6.4 关键路径.....	29
4.7 运行界面.....	31
4.8 收获.....	31

一、一元多项式的表示和相加

1.1 任务要求

采用面向对象的思想，自行定义一个多项式的类，实现多项式的创建、销毁、输出、求多项式长度以及多项式求和功能。

1.2 基本原理

显然，可以对多项式采用顺序存储结构，使得多项式相加的算法定义十分简洁，然而在通常的应用中，多项式的次数可能很高且变化很大，使得顺序存储结构的最大长度很难确定。在实际应用程序中取用哪一种，要视多项式做何种运算而定，若只对多项式进行“求值”等不改变多项式的系数和指数的运算，则采用类似于顺序表的顺序存储结构即可，否则应采用链式存储表示。在本实验中，选择使用线性链表的基本操作来实现一元多项式的运算。

根据一元多项式相加的运算规则：对于两个一元多项式中所有指数的相同的项，对应系数相加，若其和不为零，则构成多项式中的一项；对于两个二元多项式中所有指数不同的项，则分别复抄到“和多项式”中去。

1.3 算法思路

假设指针 **qa** 和 **qb** 分别指向多项式 **A** 和多项式 **B** 中当前进行比较的某个结点，则比较两个结点中的指数项，有下列三种情况：

- (1) 指针 **qa** 所指结点的指数值<指针 **qb** 所指结点的指数值，则应摘取 **qa** 指针所指节点插入到“和多项式”链表中去；
- (2) 指针 **qa** 所指结点的指数值>指针 **qb** 所指结点的指数值，则应摘取 **qb** 指针所指节点插入到“和多项式”链表中去；
- (3) 指针 **qa** 所指结点的指数值=指针 **qb** 所指结点的指数值，则将两个结点中的系数相加。若和数不为 0，则修改 **qa** 所指结点的系数值，同时释放 **qb** 所指节点；反之，从多项式 **A** 的链表中删除相应的结点，并释放 **qa** 和 **qb** 所指结点。

1.4 数据结构

```
class polyn {  
    float coef;           //系数  
    int expn;             //指数  
    polyn *next;  
public:  
    polyn * CreatePolyn(int m);  
};
```

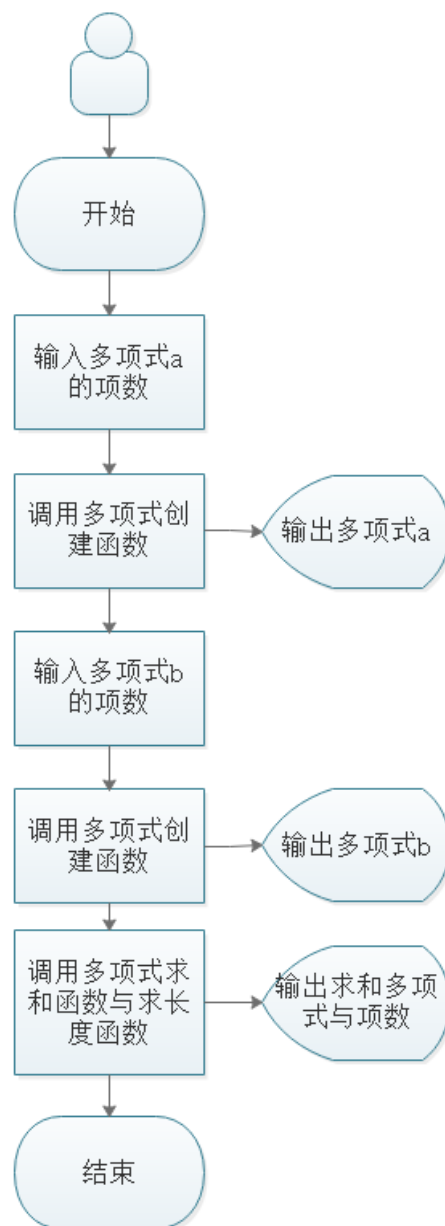
```

void DestroyPolyn(polyn *head);
void PrintPolyn(polyn *head);
int PolyLength(polyn *head);
void AddPolyn(polyn *&ha, polyn *hb);
void MultiplyPolyn(polyn *pa, polyn *pb);
};

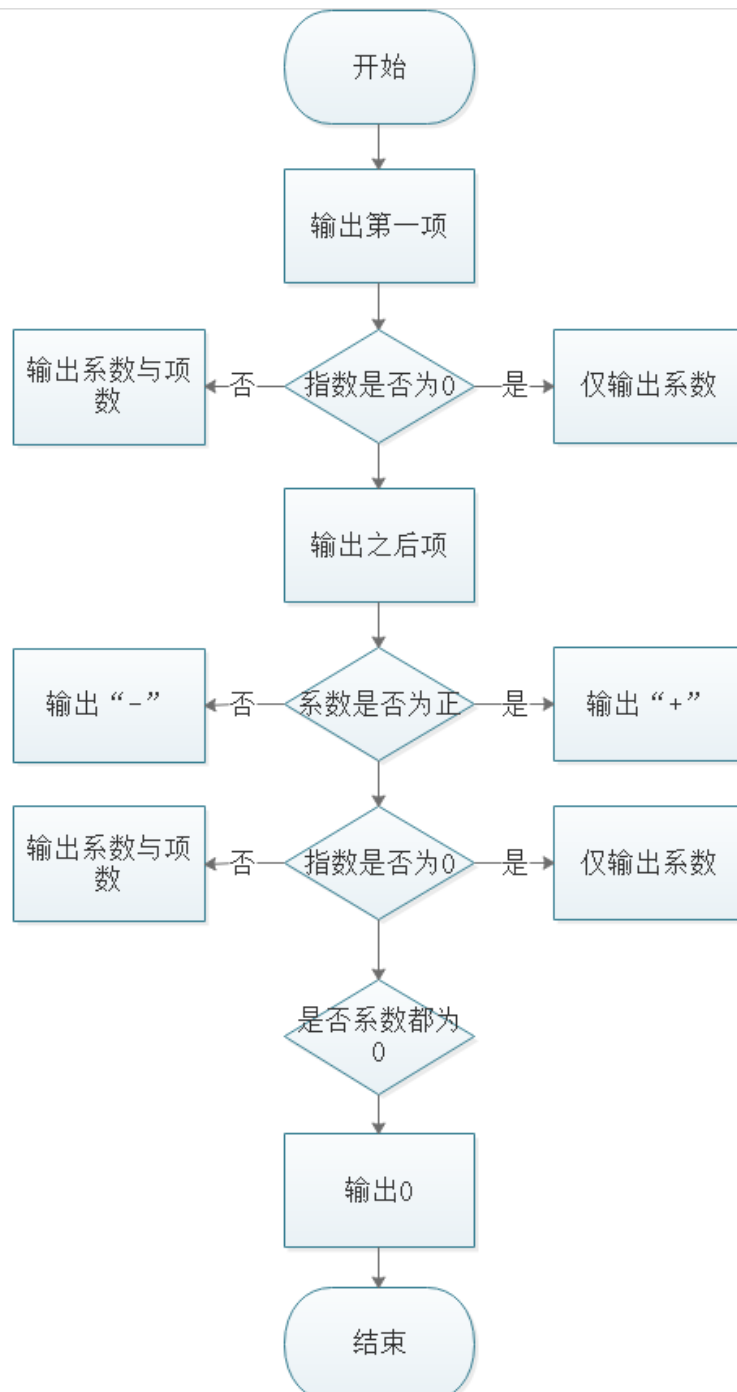
```

1.5 流程图

1.5.1 总体功能流程



1.5.2 多项式输出算法流程



1.6 源代码

1.6.1 多项式创建

需要输入每一项的系数和指数

```

polyn * polyn::CreatePolyn(int m) {
    polyn * head, *p1, *p2;
    head = p2 = 0;
    for (int i = 0; i < m; i++) {
        p1 = new polyn;
        cout << "请输入第" << i+1 << "项系数和指数:" ;
        cin >> p1->coef >> p1->expn;
        if (head == 0) {
            head = p1;
        } else {
            p2->next = p1;
        }
        p2 = p1;
    }
    if (head) {
        p2->next = 0;
    }
    return head;
}

```

1.6.2 多项式的销毁

```

void polyn::DestroyPolyn(polyn *head) {
    polyn *p = head;
    while (head) {
        p = head;
        head = head->next;
        delete p;
    }
}

```

1.6.3 多项式输出

输出时存在几种特殊情况：

- 第一个输出时，前面不需要加符号位。
- 当系数求和等于 0 时，判断使它不输出。
- 当系数为负数时，输出时加上括号。
- 当所有系数都为 0 的时候，输出一个 0。

```

void polyn::PrintPolyn(polyn *head) {
    polyn *p = head;
    int k = 0; //控制输出0的情况
    cout << "y=";
    //第一项的输出

```

```

if (p->coef != 0) {
    if (p->expn == 0) {
        cout << p->coef;
        k++;
    }
    else {
        cout << p->coef << "*x^" << p->expn;
        k++;
    }
}
p = p->next;
//之后项的输出
while (p) {
    if (p->coef > 0) {
        if (p->expn == 0) {
            cout << "+" << p->coef;
            k++;
        } else {
            cout << "+" << p->coef << "*x^" << p->expn;
            k++;
        }
    } else if (p->coef < 0) {
        if (p->expn == 0) {
            cout << "+(" << p->coef << ")";
            k++;
        } else {
            cout << "+(" << p->coef << ")*x^" << p->expn;
            k++;
        }
    }
    p = p->next;
}
//判断是否全为0
if (k == 0) {
    cout << "0";
}
cout << endl;
}

```

1.6.4 多项式求长度

```

int polyn::PolyLength(polyn *head) {
    polyn *p;

```



```

    int length=0;
    p = head;
    while (p != 0) {
        p = p->next;
        length++;
    }
    return length;
}

```

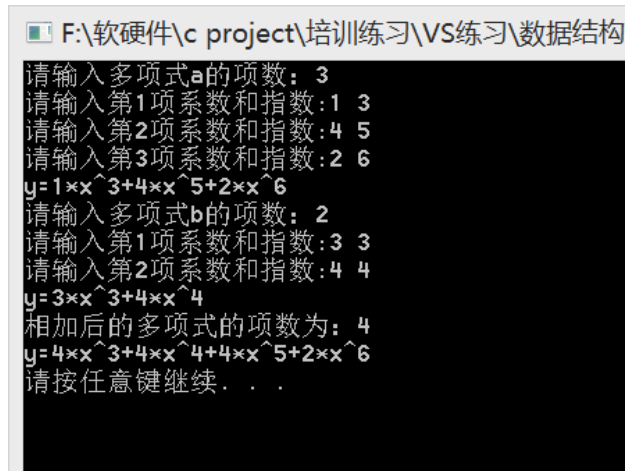
1.6.5 多项式求和

```

void polyn::AddPolyn(polyn *&ha, polyn *hb) {
    polyn *pa, *qa, *pb, *qb;
    pa = qa = ha;
    pb = qb = hb;
    while(pa && pb) {
        //处理求和
        if (pa->expn == pb->expn) {
            pa->coef += pb->coef;
            if (pa != qa) {
                qa = pa;
            }
            pa=pa->next;
            pb=pb->next;
            delete qb;
            qb = pb;
        }
        //处理b中节点的插入
        else if (pa->expn > pb->expn) {
            if (pa==ha) {
                pb= pb->next;
                qb->next = pa;
                ha = qb;
            } else {
                pb = pb->next;
                qa->next = qb;
                qb->next = pa;
                qa = qb;
            }
            qb = pb;
        }
        //处理a移位
        else {
            if (pa != qa) {
                qa = pa;
            }
            pa=pa->next;
        }
    }
    if (pb) {
        qa->next = pb;
    }
}

```

1.7 运行界面



```
F:\软件\c project\培训练习\VS练习\数据结构
请输入多项式a的项数: 3
请输入第1项系数和指数: 1 3
请输入第2项系数和指数: 4 5
请输入第3项系数和指数: 2 6
y=1*x^3+4*x^5+2*x^6
请输入多项式b的项数: 2
请输入第1项系数和指数: 3 3
请输入第2项系数和指数: 4 4
y=3*x^3+4*x^4
相加后的多项式的项数为: 4
y=4*x^3+4*x^4+4*x^5+2*x^6
请按任意键继续. . .
```

1.8 收获

通过本次实验，我熟悉了链表的使用。通过画链表动态图顺利地理清了算法步骤，上机按照思路敲出来之后，还是有很多报错与不合理的地方，经过不断的调试与细节优化，最终完成了本次实验。我的最大感想就是有时只有思路并不能保证可以把这个工程敲出来，而且在调试的过程中，还会遇到很多需要注意的小细节，所以上机实践很重要。很遗憾乘法的那个算法并没有做，因为有思路，又嫌我的思路实现起来太麻烦，还要不知道调试多久，所以就没有做，希望以后有机会可以找到一个最简单的实现思路并将求乘函数补上。

二、后缀表达式

2.1 任务要求

利用面向对象思想，设计一个栈的类与后缀表达式的类，在栈类中实现基本栈的操作。完成中缀表达式转为后缀表达式，同时对后缀表达式求值。

2.2 基本原理

栈是限定仅在表尾进行插入或删除操作的线性表，表尾是栈顶，表头是栈底，不含元素的空表称空栈，特点为先进后出（FILO）或后进先出（LIFO）。

后缀式的转换规则为有中缀表达式 $Exp = a * b + (c - d / e) * f$ 转为后缀式为： $a b * c d e / - f * +$ 其中运算符在式中出现的顺序恰为表达式的运算顺序；每个运算符和在它之前且紧靠它的两个操作数构成一个最小表达式。

后缀表达式的运算规则为连续出现的两个操作数和在它们之后且紧靠它们的运算符构成一个最小表达式，即先找运算符，再找操作数。

2.3 算法思路

由中缀求后缀：每个运算符的运算次序要由它之后的一个运算符来定，在后缀式中，优先数高的运算符领先于优先数低的运算符。运算步骤为：

- (1) 设立运算符栈；
- (2) 设表达式的结束符为“#”（预设运算符栈的栈底为“#”）；
- (3) 若当前字符是操作数，则直接发送给后缀式。
- (4) 若当前运算符的优先数高于栈顶运算符，则进栈；否则，退出栈顶运算符发送给后缀式；
- (5) “(” 对它之前后的运算符起隔离作用，“)” 可视为自相应左括弧开始的表达式的结束符。

由后缀表达式求值运算步骤为：

- (1) 从头到尾找到第一个运算符；
- (2) 找到紧靠运算符之前两个操作数，作为最小表达式，进行运算；
- (3) 将最小表达式替换为该运算结果。

2.4 数据结构

```
class stack{
public:
    char *str_arr;
    int stacksize=0;
```

```

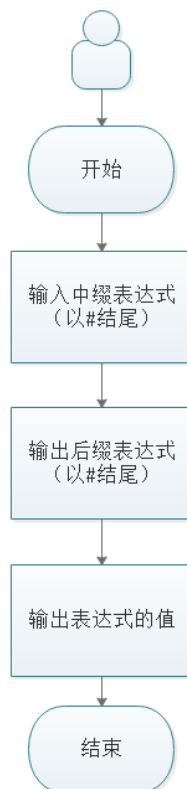
public:
    stack();
    ~stack();
    void push(char e);
    char getTop();
    char pop();
    int stack_size();
    int isEmpty();
};

class suffix {
public:
    int evaluation(char suffix[]);
    char* transform(char str_arr[]);
    int OpMember(char ch);
    char Operate(char a, char ch, char b);
    int precede(char a, char b);
    int getPrioraty(char e);
};

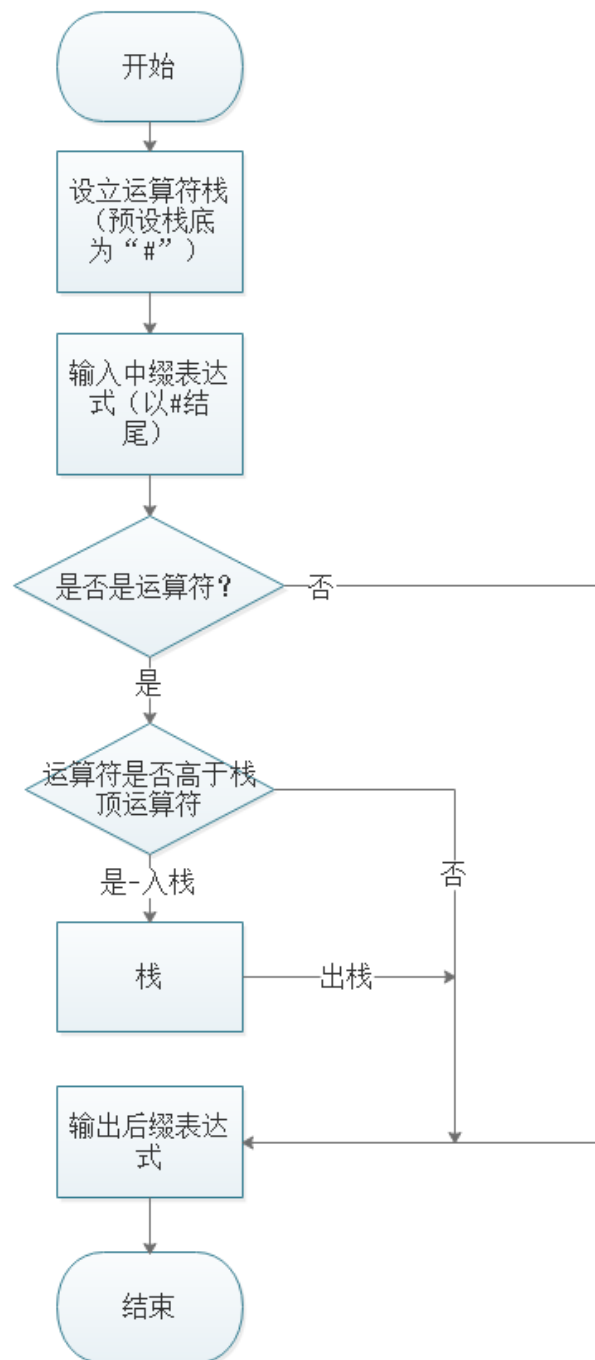
```

2.5 流程图

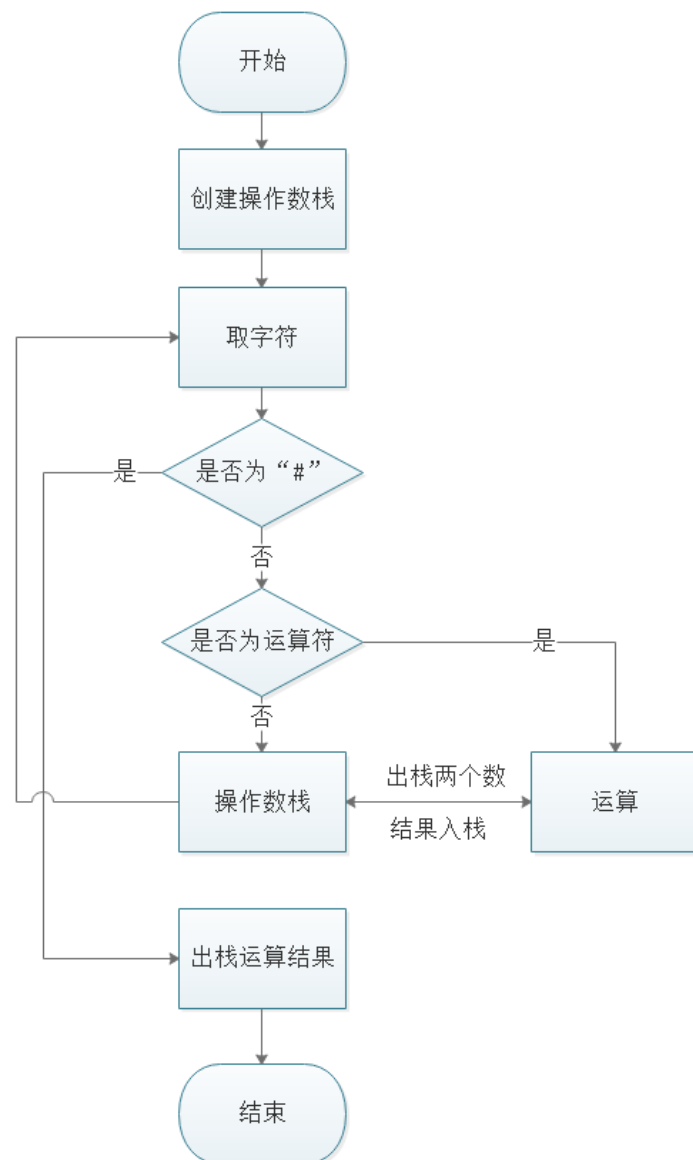
2.5.1 总体功能流程



2.5.2 中缀转后缀



2.5.3 后缀求值



2.6 源代码

2.6.1 创建栈

```
stack::stack() {  
    str_arr = new char[Stack_init_size];  
    if (!str_arr) {  
        cout << "arr malloc error!" << endl;  
    }  
}
```

2.6.2 销毁栈

```
stack::~~stack() {  
    if (str_arr) {  
        delete[] str_arr;  
        str_arr = NULL;  
    }  
}
```

2.6.3 栈中添加元素

```
void stack::push(char e) {  
    str_arr[stacksize++] = e;  
}
```

2.6.4 栈中取出元素

```
char stack::getTop() {  
    return str_arr[stacksize - 1];  
}
```

2.6.5 栈中删除元素

```
char stack::pop() {  
    char top = str_arr[stacksize-1];  
    stacksize--;  
    return top;  
}
```

2.6.6 栈长度

```
int stack::stack_size() {  
    return stacksize;  
}
```

2.6.7 栈是否为空

```
int stack::isEmpty() {  
    return stack_size() == 0;  
}
```

2.6.8 求后缀表达式的值

```
int suffix::evaluation(char suffix[]) {
    stack *s = new stack();
    char a, b;
    char ch = *suffix++;
    while (ch != '#') {
        if (!OpMember(ch))
            s->push(ch); // 非运算符入操作数栈
        else {
            b = s->pop();
            a = s->pop(); // 退出栈顶两个操作数
            s->push(Operate(a, ch, b)); // 作相应运算，将运算结果入栈
        }
        ch = *suffix++; // 继续取下一字符
    }
    char result = s->pop();
    return result-'0';
}
```

2.6.9 中缀转后缀

```
char* suffix::transform(char str_arr[]) {

    char *suffix = new char ;
    stack *s = new stack();
    char c; char *p = str_arr;
    char ch = *p;
    int k = 0;
    s->push('#'); // 预设运算符栈的栈底元素为'#'
    while (!s->isEmpty()) {
        if (!OpMember(ch))
            suffix[k++] = ch; // 操作数直接发送给后缀式
        else {
            switch (ch) {
                case '(':
                    s->push(ch); break; // 左括弧一律入栈
                case ')': {
                    c = s->pop();
                    while (c != '(') { // 自栈顶至（之前运算符发给后缀式
                        suffix[k++] = c;
                        c = s->pop();
                    }
                }
            }
        }
        ch = *p++;
    }
}
```



```

        break;
    }
    default: {
        while (precede(s->getTop(), ch)) {
            suffix[k++] = s->getTop();
            c = s->pop();
            if (c == '#') {
                break;
            }
        } // 将栈中所有优先数不小于当前运算符发送给后缀式
        if (ch != '#') {
            s->push(ch); // 优先数大于栈顶的运算符入栈
        }
        break;
    }
}
}
if (ch != '#')
    ch = *++p;
}
suffix[k] = '\0'; // 添加字符串的结束符
return suffix;
}

```

2.6.10 其他小函数

```

int suffix::OpMember(char ch) {
    int result;
    switch (ch) {
        case '+':
        case '-':
        case '*':
        case '/':
        case '(':
        case ')':
        case '#':
            result = 1; break;
        default:
            result = 0; break;
    }
    return result;
}

```

```

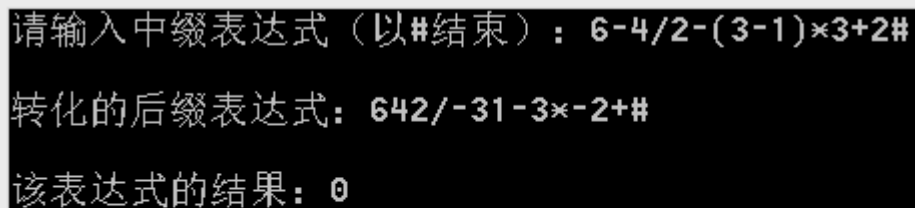
char suffix::Operate(char a, char ch, char b) {
    double result;
    double aa, bb;
    aa = a - '0';
    bb = b - '0';
    switch (ch)
    {
    case '+':
        result = aa + bb; break;
    case '-':
        result = aa - bb; break;
    case '*':
        result = aa*bb; break;
    case '/':
        result = aa / bb; break;
    default:
        break;
    }
    return result + '0';
}

int suffix::precede(char a, char b) {
    return getPrioraty(a) >= getPrioraty(b);
}

int suffix::getPrioraty(char e) {
    switch (e) {
    case '+':
    case '-':
        return 1;
    case '*':
    case '/':
        return 2;
    case '(':
    case ')':
        return 0;
    default:
        return -1;
    }
}

```

2.7 运行界面



```
请输入中缀表达式（以#结束）： 6-4/2-(3-1)*3+2#  
转化的后缀表达式： 642/-31-3*-2+#  
该表达式的结果： 0
```

2.8 收获

在本实验中，我调试的大部分时间花在了栈类的构建中。栈类的构建思路很清晰，但是在用其中方法的时候，遇到了各种报错，说明了在写算法之前，基础函数要写写好，以便在用的时候不会因为工具函数出错而打乱思路。

在由中缀表达式转后缀，然后后缀求值的时候，我一直遇到一个问题，就是转成的后缀与正确结果是一样的，但是在下一步的求值过程中，就是不是那个表达式，我进行了各种尝试，包括在查找原因的过程中也做了很多尝试，历时很久，终于调试成功，还是很开心的。

三、赫夫曼树

3.1 任务要求

用菜单形式实现用户选择功能，构建赫夫曼树，赫夫曼编码、译码。其中赫夫曼树的展示形式采用课本 P149 图 (b)。

3.2 基本原理

设有 n 个权值 $\{w_1, w_2, \dots, w_n\}$ ，构造一棵有 n 个叶子结点的二叉树，每个叶子的权值为 w_i ，则 w_{pl} 最小的二叉树叫 Huffman 树。Huffman 树也称为最优二叉树。

Huffman 编码是根据字符出现频率构造 Huffman 树，然后将树中结点引向其左孩子的分支标“0”，引向其右孩子的分支标“1”；每个字符的编码即为从根到每个叶子的路径上得到的 0、1 序列是“前缀码”。如果在一个编码系统中，任何一个编码都不是其它编码的前缀(最左子串)，则称该编码系统的编码是前缀码。

3.3 算法思路

构造 Huffman 树：

- (1) 根据给定的 n 个权值 $\{w_1, w_2, \dots, w_n\}$ ，构造 n 棵只有根结点的二叉树，令其权值为 w_j ；
- (2) 在森林中选取两棵根结点权值最小的树作左右子树，构造一棵新的二叉树，置新二叉树根结点权值为其左右子树根结点权值之和；
- (3) 在森林中删除这两棵树，同时将新得到的二叉树加入森林中；
- (4) 重复上述两步，直到只含一棵树为止，这棵树即哈夫曼树；

3.4 数据结构

//Huffman树的存储表示

typedef struct

{

int weight;

//权值

int parent, lchild, rchild;

//双亲，左孩子，右孩子的下标

}HTNode, *HuffmanTree;

//动态分配数组存储Huffman树

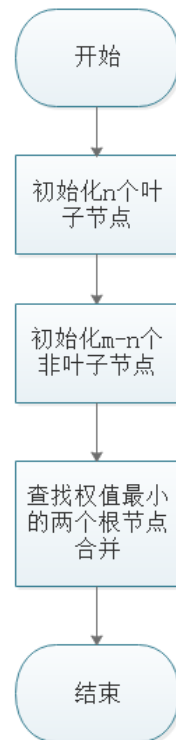
//Huffman编码的存储表示

typedef char **HuffmanCode;

//动态分配数组存储 Huffman 编码表

3.5 流程图

3.5.1 构建赫夫曼树



3.6 源代码

3.6.1 构造赫夫曼树

```
void CreateHuffmanTree(HuffmanTree &hufmtree, int n) {  
    if (n <= 1)  
        return;  
    int m = 2 * n - 1;  
    hufmtree = new HTNode[m + 1];  
    int i;  
    for (i = 1; i <= n; i++) { // 初始化n个叶子节点  
        cin >> hufmtree[i].weight;  
        hufmtree[i].parent = 0;  
        hufmtree[i].lchild = 0;  
        hufmtree[i].rchild = 0;  
    }  
    for (; i <= 2 * n - 1; i++) { // 初始化m-n个非叶子节点
```

```

        huftree[i].weight = 0;
        huftree[i].parent = 0;
        huftree[i].lchild = 0;
        huftree[i].rchild = 0;
    }

    for (int k = n+1; k <= 2 * n - 1; k++) {
        int s1, s2;
        Select(huftree, k, s1, s2);    //查找权值最小的两个根节点合并
        huftree[s1].parent = k;
        huftree[s2].parent = k;
        huftree[k].lchild = s1;
        huftree[k].rchild = s2;
        huftree[k].weight = huftree[s1].weight + huftree[s2].weight;
    }
}

```

3.6.2 赫夫曼编码

```

void CreatHuffmanCode(HuffmanTree HT, HuffmanCode &HC, int n) {
    HC = new char*[n + 1];    //分配n个字符编码的头指针向量
    char *cd = new char[n];    //分配求编码的数据空间
    cd[n - 1] = '\0';
    for (int i = 1; i <= n; i++) {
        int start = n - 1;    //start 开始指向编码结束符位置
        int c = i;
        int f = HT[c].parent;    //f指向结点c的双亲
        while (f != 0) {    //根结点判断
            --start;    //向前回溯，start向前指向一个位置
            if (HT[f].lchild == c)
                cd[start] = '0';
            else
                cd[start] = '1';
            c = f;
            f = HT[f].parent;    //继续向上回溯
        }
        HC[i] = new char[n - start];    //为第i个字符编码分配空间
        strcpy(HC[i], &cd[start]);
    }
    delete cd;
}

```

3.6.3 赫夫曼译码

```
void TranCode(HuffmanTree HT, char a[], char zf[], char b[], int n) {
    int q = 2 * n - 1;
    int k = 0; //记录b数组的下标
    int i = 0;
    for (i = 0; a[i] != '\0'; i++) {
        if (a[i] == '0') {
            q = HT[q].lchild;
        } else if (a[i] == '1') {
            q = HT[q].rchild;
        }
        if (HT[q].lchild == 0 && HT[q].rchild == 0) { //是叶子节点
            b[k++] = zf[q];
            q = 2 * n - 1;
        }
    }
    b[k] = '\0';
}
```

3.6.4 选权值最小两结点

```
void Select(HuffmanTree a, int n, int &s1, int &s2) {
    int temp;
    for (int i = 0; i < n; i++) {
        if (a[i].parent == 0) { //选择第一个parent为0的节点
            s1 = i;
            break;
        }
    }
    for (int i = 0; i < n; i++) { //遍历找到权值最小的作为s1
        if (a[i].parent == 0 && a[s1].weight > a[i].weight)
            s1 = i;
    }
    for (int j = 0; j < n; j++) {
        if (a[j].parent == 0 && j != s1) { //选择第一个parent为0
            且不是s1的节点
            s2 = j;
            break;
        }
    }
    for (int j = 0; j < n; j++) { //遍历找到权值最小且不是s1
        if (a[j].parent == 0 && a[s2].weight > a[j].weight && j != s1)
```

```

        s2 = j;
    }
    if (s1 > s2) {
        temp = s1;
        s1 = s2;
        s2 = temp;
    }
}

```

3.6.5 菜单函数

```

void menu() {
    cout << endl;
    cout << "
*****" <<
endl;
    cout << "          |          ★★★★★★★哈夫曼编码与译码★★★★★★★
★          |" << endl;
    cout << "          |          1. 创建哈夫曼树
" << endl;
    cout << "          |          2. 进行哈夫曼编码
" << endl;
    cout << "          |          3. 进行哈夫曼译码
" << endl;
    cout << "          |          4. 退出程序
" << endl;
    cout << "
*****" <<
endl;
    cout << endl;
}

```


3.7 运行界面

```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
|          ★★★★★★哈夫曼编码与译码★★★★★★          |
|          1. 创建哈夫曼树                          |
|          2. 进行哈夫曼编码                          |
|          3. 进行哈夫曼译码                          |
|          4. 退出程序                                |
|          |                                           |
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

请选择功能(1-创建 2-编码 3-译码 4-退出): 1
请输入字符个数:8
请依次输入8个字符: a b c d e f g
h
请依次输入8个字符的权值: 5 29 7 8 14 23 3 11
创建Huffman树的参数输出:
nodei  char  weight  parent  lchild  rchild
1      a      5        9        0        0
2      b      29       14        0        0
3      c      7        10        0        0
4      d      8        10        0        0
5      e      14       12        0        0
6      f      23       13        0        0
7      g      3        9        0        0
8      h      11       11        0        0
9              8        11        1        7
10             15       12        3        4
11             19       13        8        9
12             29       14        5       10
13             42       15        6       11
14             58       15        2       12
15             100        0       13       14

```

```

请选择功能(1-创建 2-编码 3-译码 4-退出): 2
生成哈夫曼编码表的输出:
结点i  字符  权值  编码
1      a      5     0110
2      b     29     10
3      c      7     1110
4      d      8     1111
5      e     14     110
6      f     23      00
7      g      3     0111
8      h     11     010

请选择功能(1-创建 2-编码 3-译码 4-退出): 3
请输入想要翻译的一串二进制编码: 010010010
译码成功! 翻译结果为: hhh

```

3.8 收获

本次实验主要是对照课本代码进行的调试，从伪代码转为标准 c++也花费了一些时间。对代码的理解还有待深入。

四、关键路径

4.1 任务要求

选择任意图的存储形式，实现求关键路径。

4.2 基本原理

AOV 网是用顶点表示活动，用弧表示活动间优先关系的有向图称为顶点表示活动的网(Activity On Vertex network)，简称 AOV 网。若 $\langle v_i, v_j \rangle$ 是图中有向边，则 v_i 是 v_j 的直接前驱； v_j 是 v_i 的直接后继；若 v_i 到 v_j 有一条有向路径，则 v_i 是 v_j 的前驱； v_j 是 v_i 的后继。AOV 网中不允许有回路，因为回路意味着某项活动以自己为先决条件。

拓扑排序是按照有向图给出的次序关系，将图中顶点排成一个线性序列，对于有向图中没有限定次序关系的顶点，则可以人为加上任意的次序关系。由此所得顶点的线性序列称之为拓扑有序序列。

AOE 网(Activity On Edge)也叫边表示活动的网。AOE 网是一个带权的有向无环图，其中顶点表示事件，弧表示活动，权表示活动持续时间。路径长度是路径上各活动持续时间之和，长度最大的那条路径叫关键路径。其中 $Ve(j)$ 表示事件 V_j 的最早发生时间，是从起点到 V_j 的最大路径长度。

关键路径——必定由关键活动组成，即在有向网中查找所有 $l(i)=e(i)$ 的活动。有时，一个 AOE 网同时有几条关键路径，提前完成非关键活动并不能加快工程的进度。关键活动的速度提高也是有限度的（不能改变网的关键路径）。若网中同时有几条关键路径，为提高整个工程的进度，须同时提高几条关键路径上的活动速度。

4.3 算法思路

拓扑排序：

- (1) 在有向图中选一个没有前驱的顶点且输出之；
- (2) 从图中删除该顶点和所有以它为尾的弧；
- (3) 重复上述两步，直至全部顶点均已输出；或者当图中不存在无前驱的顶点为止。

关键路径：

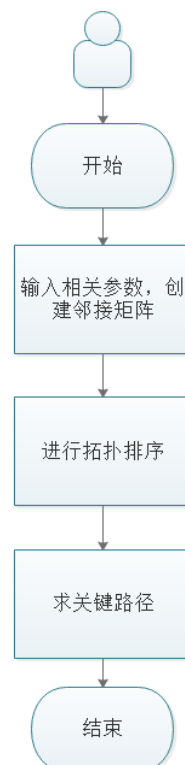
- (1) 求 $Ve(j)$ 。向前递推 $ve(0) = 0$ ； $ve(j) = \text{Max}\{ve(i) + \text{dut}(\langle i, j \rangle)\}$ ；
- (2) 求 $Vl(j)$ 。向后递推 $vl(n-1) = ve(n-1)$ ； $vl(i) = \text{Min}\{vl(j) - \text{dut}(\langle i, j \rangle)\}$ ；
- (3) 求 $e(i)$ 。 $e(i) = Ve(j)$ 。
- (4) 求 $l(i)$ 。 $l(i) = Vl(k) - \text{dut}(\langle j, k \rangle)$ 。
- (5) 计算 $l(i) - e(i)$ ，等于 0 即为关键活动。

4.4 数据结构

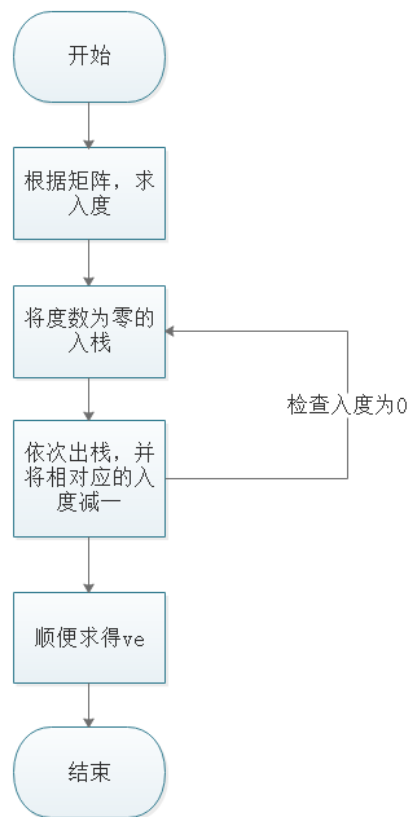
```
struct Graph {  
    int matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];    //邻接矩阵  
    int vexnum;    //顶点个数  
    int arcs;    //弧的个数  
};  
class MGraph {  
public:  
    stack<int> s;  
    int ve[MAX_VERTEX_NUM] = { 0 };    //最早发生时间  
    int vl[MAX_VERTEX_NUM] = { INFINITY };    //最迟发生时间  
public:  
    void CreateGraph(Graph &G);  
    void PrintGraph(Graph G);  
    void TopoSort(Graph G, stack<int> &T);  
    void CriticalPath(Graph G, stack<int> &T);  
};
```

4.5 流程图

4.5.1 总体功能流程



4.5.2 拓扑排序



4.5.3 关键路径



4.6 源代码

4.6.1 创建邻接矩阵

```
void MGraph::CreateGraph(Graph &G) {
    cout << "请输入顶点个数与弧个数:";
    cin >> G.vexnum >> G.arcs;
    for (int i = 0; i < G.vexnum; i++) {
        for (int j = 0; j < G.vexnum; j++)
            G.matrix[i][j] = INFINITY;
    }
    cout << "请分别输入顶点入度与出度的下标与弧的权值:"<<endl;
    for (int k = 0; k < G.arcs; k++) {
        int i, j, w;
        cin >> i >> j >> w;
        G.matrix[i-1][j-1] = w;
    }
}
```

4.6.2 打印邻接矩阵

```
void MGraph::PrintGraph (Graph G){
    cout << "图的邻接矩阵表示:"<<endl;
    for (int i = 0; i < G.vexnum; i++){
        for (int j = 0; j < G.vexnum; j++){
            if (G.matrix[i][j] == INFINITY){
                cout << "*" << '\t';
            }else{
                cout << G.matrix[i][j] << '\t';
            }
        }
        cout << endl;
    }
    cout << endl;
}
```

4.6.3 拓扑排序

```
void MGraph::TopoSort (Graph G , stack<int> &T) {
    int indegree[MAX_VERTEX_NUM] = {0};
    int count = 0;
```

```

//求入度（列和）
for (int i = 0; i < G.vexnum; i++) {
    for (int j = 0; j < G.vexnum; j++) {
        if (G.matrix[j][i] != INFINITY)
            indegree[i]++;
    }
}
for (int i = 0; i < G.vexnum; i++) {
    if (!indegree[i]) {
        s.push(i);
    }
}
cout << "拓扑排序为:";
while (s.size()) {
    int i = s.top();
    s.pop();
    cout << "v" << i + 1 << "\t";
    T.push(i);
    count++;
    for (int j = 0; j < G.vexnum; j++) {
        if (G.matrix[i][j] != INFINITY) {
            if (!(--indegree[j]))
                s.push(j);
            if (ve[i] + G.matrix[i][j] > ve[j]) {
                ve[j] = ve[i] + G.matrix[i][j];
            }
        }
    }
}
cout << endl;
if (count < G.vexnum) {
    cout << "该图有回路! " << endl;
}
s.~stack();
}

```

4.6.4 关键路径

```

void MGraph::CriticalPath(Graph G, stack<int>T) {

    for (int i = 0; i < G.vexnum; i++) {
        vl[i] = INFINITY;
    }
}

```

```

    }
    vl[G.vexnum - 1] = ve[G.vexnum - 1];
    while (!T.empty()) {
        int i = T.top();
        T.pop();
        for (int j = 0; j < G.vexnum; j++) {
            if (G.matrix[j][i] != INFINITY) {
                if (vl[i] - G.matrix[j][i] < vl[j]) {
                    vl[j] = vl[i] - G.matrix[j][i];
                }
            }
        }
    }
}

for (int i = 0; i < G.vexnum; i++) {
    for (int j = 0; j < G.vexnum; j++) {
        if (G.matrix[i][j] != INFINITY) {
            if (ve[i] == vl[j] - G.matrix[i][j]) {
                cout<<"v"<<i+1 << "-v" << j+1 << " " <<
G.matrix[i][j] << endl;
            }
        }
    }
}

cout << "关键路径需要" << vl[G.vexnum - 1] << "天" << endl;
}

```

4.7 运行界面

```
请输入顶点个数与弧个数:9 11
请分别输入顶点入度与出度的下标与弧的权值:
1 2 6
1 3 4
2 5 1
3 5 1
1 4 5
4 6 2
5 7 9
5 8 7
6 8 4
7 9 2
8 9 4
图的邻接矩阵表示:
×      6      4      5      ×      ×      ×      ×      ×
×      ×      ×      ×      1      ×      ×      ×      ×
×      ×      ×      ×      1      ×      ×      ×      ×
×      ×      ×      ×      ×      2      ×      ×      ×
×      ×      ×      ×      ×      ×      9      7      ×
×      ×      ×      ×      ×      ×      ×      4      ×
×      ×      ×      ×      ×      ×      ×      ×      2
×      ×      ×      ×      ×      ×      ×      ×      4
×      ×      ×      ×      ×      ×      ×      ×      ×
拓扑排序为:u1    u4    u6    u3    u2    u5    u8    u7    u9
u1-u2  6
u2-u5  1
u5-u7  9
u5-u8  7
u7-u9  2
u8-u9  4
关键路径需要18天
请按任意键继续. . .
```

4.8 收获

通过本次实验，熟悉了关键路径的求解步骤，以及几个时间节点的求解方法。本来以为不管是从输入还是输出都已经考虑得比较周道了，但是还是被老师批评输出不精致，可以输出一整个过程，也希望自己在以后的编程过程中，能够多用一点心在细节的处理上。