数据结构与程序设计专题实验报告

实验一表达式求值

一 实验目的

熟练掌握栈、队列等基本操作及其在实际问题中的应用。

二 实验要求

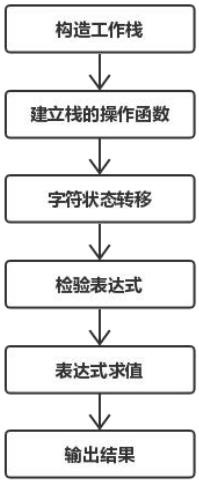
2.1 基本要求：实现栈的基本操作，及表达式求值的实现。
2.2 实现要求

① 实现栈的 push、pop 基本操作；
② 检测表达式的输入是否是正确的数学表达式；
③ 对于正确的数学表达式求取其值。

注：1）数学表达式的判读与求值需支持加减乘除、小括号、中括号、大括号、幂运算。
2）不正确的表达式可能包括：字母、非运算符号、括号不匹配，运算符的排列不符合表达式形式、分母为 0 等多种情况。
3）表达式通过在命令行参数输入输出获取。
4）输出要求：（1）表达式正确的情况下输出结果；（2）命令行参数不正确输出 ERROR_01；（3）表达式存在格式错误输出 ERROR_02；（3）表达式在计算过程中出现逻辑错误输出 ERROR_03。

三 设计思路

为了实现算符优先算法。可以使用两个工作栈。一个称为 OPTR，用以寄存运算符，另一个称做 OPND，用以寄存操作数或运算结果。
1.首先置操作数栈为空栈，表达式起始符"#"为运算符栈的栈底元素；
2.依次读入表达式，若是操作符即进 OPND 栈，若是运算符则和 OPTR 栈的栈顶运算符比较优先权后作相应的操作，直至整个表达式求值完毕（即 OPTR 栈的栈顶元素和当前读入的字符均为"#"）。



3.1 数据存储结构设计

由于表达式是由操作符，运算符和界限符组成的。如果只用一个 char 类型栈，不能满足 2 位以上的整数，所以还需要定义一个 int 类型的栈用来寄存操作数。

```
typedef struct { //运算符栈
    char* base;//在栈构造之前和销毁之后，base的值为null
    char* top;//栈顶指针
    int stacksize;
}StackC;

typedef struct {//运算数栈
    double* base;//在栈构造之前和销毁之后，base的值为null
    double* top;//栈顶指针
    int stacksize;
   }StackN;
```

3.2 不同功能实现

3.2.1 算符优先算法

若表达式正确，则采用算符优先算法对表达式进行求值运算。用 EvaluateExpression 函数实现该功能。本函数将依次读取表达式中每个字符，并进行相应判断，直到读取到最后一位"#"，结束读取，计算得出相应表达式求值的结果。

```
long double EvaluateExpression(char* expression, int& Ver) {//表达式求值
    StackC OPTR;//运算符栈
    StackN OPND;//运算数栈
    int i = 0, ver = 1, z = 0;
    long double a, b, j, k;
    char c, x, theta;
    InitStackC(OPTR);//初始化运算符栈
    PushStackC(OPTR, '#');//#的优先级最小，便于之后 pop 第一个运算符进入符栈
    InitStackN(OPND);//初始化运算数栈
```

```
c = expression[i];
while (c != '#' || GetTopStackC(OPTR) != '#')
{
        j = 0, k = 1;
        while (ChartoInt(c) == 13)
        {//整数位读取
                z++;
                j *= 10;
                j += (c - '0');
                c = expression[++i];
                if (c == '.')
                {//小数位读取
                        c = expression[++i];
                        while (ChartoInt(c) == 13)
                        {
                                k /= 10;
                                j += (c - '0') * k;
                                c = expression[++i];
                        }
                }//if
        }//while
        if (z != 0)
                PushStackN(OPND, j);//将读取到的运算数入运算数栈
        z = 0;
        switch (Precede(GetTopStackC(OPTR), c))
        {
        case -1://栈顶运算符优先级低
                PushStackC(OPTR, c);
                c = expression[++i];
                break;
        case 0://脱括号并接收下一字符
                PopStackC(OPTR, x);
                c = expression[++i];
                break;
        case 1://栈顶元素优先级高
                PopStackC(OPTR, theta);
                PopStackN(OPND, b);
                PopStackN(OPND, a);
                PushStackN(OPND, Operate(a, theta, b, ver));
                if (ver == 0) Ver = 0;
                break;
        default:
                break;
        }//switch
```

```
    }//while
    return GetTopStackN(OPND);//返回计算结果

}//EvaluateExpression
```

3.2.2 两个运算符的优先级算法

采用矩阵判断两个运算符的优先级。1 表示 θ_1 优先权大于 θ_2；0 表示 θ_1 优先权等于 θ_2；−1 表示 θ_1 优先权小于 θ_2。

```
iint Precede(char a, char b) {//利用优先级矩阵，比较运算符号的优先级
    int i, j;
    i = ChartoInt(a) − 1;
    j = ChartoInt(b) − 1;
    int table[12][12] = {
    {1,1,−1,−1,−1,−1,−1,−1,1,1,1,1},
    {1,1,−1,−1,−1,−1,−1,−1,1,1,1,1},
    {1,1,1,1,−1,−1,−1,−1,1,1,1,1},
    {1,1,1,1,−1,−1,−1,−1,1,1,1,1},
    {1,1,1,1,1,−1,−1,−1,1,1,1,1},
    {−1,−1,−1,−1,−1,−1,−2,−2,0,−2,−2,−2},
    {−1,−1,−1,−1,−1,−1,−1,−2,−2,0,−2,−2},
    {−1,−1,−1,−1,−1,−1,−1,−1,−2,−2,0,−2},
    {−2,−2,−2,−2,−2,−2,−2,−2,−2,−2,−2,−2},
    {−2,−2,−2,−2,−2,−2,−2,−2,−2,−2,−2,−2},
    {−2,−2,−2,−2,−2,−2,−2,−2,−2,−2,−2,−2},
    {−1,−1,−1,−1,−1,−1,−1,−1,−1,−1,−1,0}
    };
    return table[i][j];
}//Precede
```

3.3 函数表达式求值功能的实现

3.3.1 字符计算

实现操作数 a 与 b 的数值运算，并检验分母是否为 0 的情况。

```
long double Operate(long double a, char theta, long double b, int& Ver) {//对 a,b 运用符号 theta 进行计算
    switch (ChartoInt(theta))
    {
    case 1:
        return a + b;
        break;
    case 2:
        return a − b;
        break;
    case 3:
        return a * b;
        break;
    case 4:
```

```
            if (b == 0)
            {//若分母为 0，则返回 ERROR
                Ver = 0;//逻辑错误输出字符串 ERROR_03
                //printf("ERROR_03");
                return ERROR;
            }
            else return a / b;
            break;
        case 5:
            if (b == 0)
                return 1.0;
            else return pow(a, b);
            break;
        default:
            break;
        }//switch
}//Operate
```

3.3.2 表达式正确性检验

利用栈检验表达式的正确性。若为运算符"("、"["、"{"则进栈，若为")"、"]"、"}"则出栈，若出栈错误或表达式读取完之后栈不为空，则表达式错误，输出 ERROR_02。

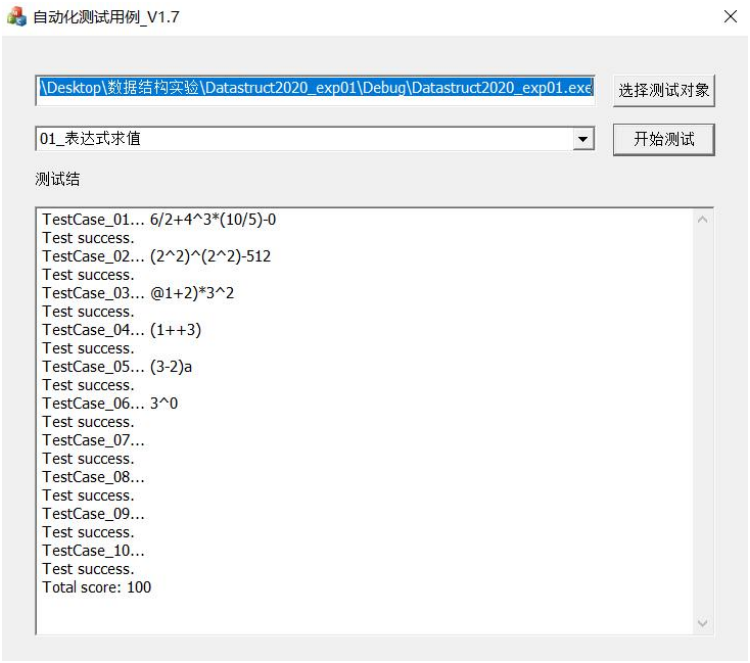
```
int Vertify(char* SString, int Str) {//表达式的正确性检验  正确返回 1  错误返回 0
    int i, k = 0;
    char e;
    if (State(SString[0]) != 1 && State(SString[0]) != 3)//开头必须为数字或左括号
        return 0;
    StackC(Sym);
    InitStackC(Sym);
    if (State(SString[0]) == 3)
        PushStackC(Sym, SString[0]);
    for (i = 1; i < Str - 1; i++)
    {
        if (SString[i] == '.')
        {
            if (State(SString[i - 1]) != 1 || State(SString[i + 1]) != 1)
                return 0;//小数点前后为数字
            continue;
        }
        if (State(SString[i]) == 3)
            PushStackC(Sym, SString[i]);//左括号入栈
        if (State(SString[i]) == 4) {
            if (ChartoInt(SString[i]) - 3 != ChartoInt(GetTopStackC(Sym)) || !GetTopStackC(Sym))//遇到右括号，符栈栈顶必为左括号
                return 0;
```

```
                else {
                    if (ChartoInt(SString[i]) - 3 == ChartoInt(GetTopStackC(Sym)))
                        PopStackC(Sym, e);
                    else return 0;
                }
                if (ChartoInt(SString[i]) == 0 || ChartoInt(SString[i]) == 12)
                    return 0;//不可能出现#或其他字符
            }
            if (Check(State(SString[k]), State(SString[i])) == 0)
                return 0;
            k = i;
        }//for
        if (State(SString[i - 1]) != 1 && State(SString[i - 1]) != 4)
            return 0;//数学表达式以数字或右括号结束
        if (SString[i] != '#')
            return 0;
        if (Str == 1)
            return 1;
        return 1;
}//Vertify
```

检验表达式中运算符"+"、"-"、"/"、"*"以及其与括号之间格式的正确性。

```
int Check(int a, int b) {//字符之间关系检查，正确返回 1，错误返回 0
        if (a == 1 && (b == 1 || b == 2 || b == 4 || b == 5))
            return 1;
        if (a == 2 && (b == 1 || b == 3))
            return 1;
        if (a == 3 && (b == 1 || b == 3))
            return 1;
        if (a == 4 && (b == 2 || b == 4))
            return 1;
        if (a == 5 && b == 1)
            return 1;
        return 0;
}//Check
```

四 运行结果



五 问题与思考

将表达式求值函数中的小数处理部分单独提出作为一个函数；malloc 之后需要 free；程序中有很多较长的 if 判断语句，导致一些输入上出错的问题；在定义变量时，导致一个变量重复定义的问题，可定义局部变量。

六 cpp 源码

Datastruct2020_exp01.cpp

实验二基于串的模式匹配

一 实验目的

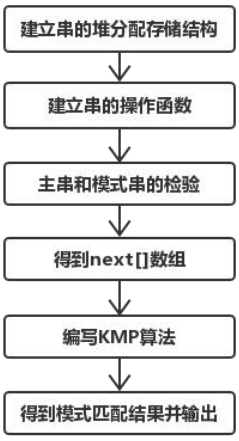
通过本次实验，熟练掌握抽象数据类型串的实现，学会使用串及其模式匹配算法解决具体应用问题，从而体会串的特点。

二 实验要求

① 使用串及其模式匹配算法。
② 用 KMP 算法实现模式匹配。
注：1）中文字符在字符串中占 2 位。
2）输出要求：（1）关键字在字符中的位置，如果不存在则输出-1；（2）命令行参数不正确输出 ERROR_01。

三 设计思路

为了实现串的模式匹配，要进行串的定义，采用堆分配数据结构，并有 next 函数的建立。



3.1 数据结构

由于串长度未知，故采用串的堆分配存储结构，便于灵活分配存储空间以及调整串的长度。

```
typedef struct{//串的堆分配
    char* ch;
    int length;
}HString;
```

3.2 next[]数组的求得

next[]数组的获取可以使用递推的策略完成，由于 next[]数组中存放的数值为当前位置最

长公共前后缀的长度，也即最长公共前缀之后一个字符的下标位置，显然如果之前位置字符与之前位置的最长公共前缀的下一个字符相同，那么当前位置的最长公共前后缀的长度只需增加一即可(特别的，如果当前位置和当前位置最长公共前后缀的后一个字符相等，说明这次比较必然失败，于是应该取其此处不相等的最长公共前后缀)；而如果这两个字符不同，我们希望找到之前字符稍短的一个公共前后缀，也即在之前字符的 next[]值上再取一次 next[]的值，再比较这个值处的字符和之前位置处字符，如果相等取此值加一即可，如果不相等，则继续循环，由于 next[]数组中的值必然比数组内值小，所以循环一直继续下去必然收敛于 0。当值为 0 时。取当前值为 0 即可。

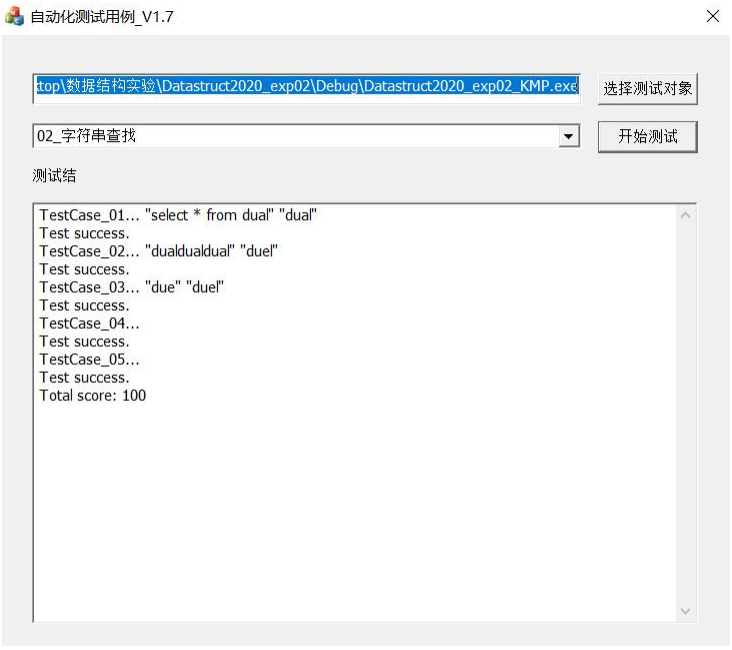
```
void Getnext(HString t, int next[]) {//得到 next 数组
    int i=1, j=0;
    next[1] = 0;
    while (i < t.length) {
        if (j == 0 || t.ch[i] == t.ch[j]){
            ++i; ++j;
            if (t.ch[i] != t.ch[j])
                next[i] = j;
            else next[i] = next[j];
        }//if
        else j = next[j];
    }//while
}// Getnext
```

3.3 KMP 算法实现

为了简化难度，此时设定了两个字符数组，分别为 s（主数组）和 t（待匹配数组）。利用 KMP 算法实现主串和模式串之间的匹配。

```
int Index_KMP(HString S, HString T, int pos) {//用模式串 T 的 NEXT 函数求主串中与模式串相对应的位置
    int i = 0, j = 1;
    i = pos;
    int next[INIT_LENGTH];
    memset(next,0, sizeof(next));
    Getnext(T, next);
    while (i <= S.length && j <= T.length){
        if (j == 0 || S.ch[i] == T.ch[j]){
            ++i; ++j;
        }
        else j = next[j];
    }
    if (j > T.length)
        return i – T.length;
    else return FALSE;
}//Index_KMP
```

四 运行结果



五 问题与思考

在将 char 类型的串转换为堆分配形式时额外定义了一个指针，导致占用了更多存储空间；本次作业使用串的堆分配存储表示，可以用函数 malloc()和 free()对串进行灵活操作。

六 cpp 源代码

Datastruct2020_
exp02_KMP.cpp

实验三基于赫夫曼树的编码/译码

一 实验目的

掌握二叉树的生成、遍历等操作，及赫夫曼编码/译码的原理。

二 实验要求

熟练掌握树的操作。

① 通过键盘输入一段字符(长度>=20)，构建霍夫曼树；

② 根据该树求每个字符的编码，并对该段字符串进行编码；

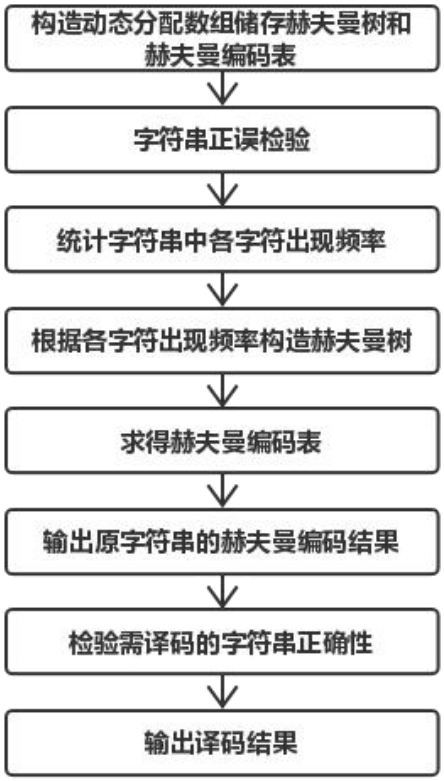
③ 将得到的编码进行译码；

④ 基于该霍夫曼树，实现非递归的先序遍历算法，输出该树所有的节点、节点的权值、节点的度和节点所在的层数；

注：1）在实现时要求霍夫曼树的左右孩子的大小关系满足，左孩子节点权值小于右孩子节点权值。

2)输出要求:(1)命令行参数不正确输出 ERROR_01;(2)编码失败输出 ERROR_02;（3）译码失败输出 ERROR_03。

三 设计思路



3.1 数据结构

由于编码和译码过程需要知道每个结点的双亲的信息和孩子结点的信息，所以设定以下存储结构。

```
typedef struct {
    unsigned int weight;
    unsigned int parent, lchild, rchild;
} HTNode, * HuffmanTree;//动态分配数组储存赫夫曼树
typedef char** HuffmanCode;//动态分配数组储存赫夫曼编码表
```

3.2.1 构造赫夫曼树并求出赫夫曼编码

首先初始化前 n 个具有元素的节点其权重及其父母节点，左右孩子节点，然后初始化后 n+1 到 2n-1 个空节点的父母节点及左右孩子节点。接着，为构筑赫夫曼树，先挑选最小的两个组成左孩子和右孩子，并组建第 n+1 个节点，与其构成树，依次循环直至最后一个 2n-1 节点构造。接下来是进行编码过程：首先每个元素的码都是在一个数组内从后往前赋值‘0’或‘1’，采用从叶子节点到父母节点到根节点的过程，每一个循环如下：如果其节点为 c，令 f 为 c 节点的父母，如果 c 为 f 的左孩子，则该元素码进 0，若为其右孩子，则码进 1，然后把 c 的父母节点 f 给 c，f 重新赋值为 c 的父母的父母节点，重新进行判断左右孩子，最后直到主根节点的父母为 0，此叶子节点的编码即该 cd，把 cd 拷贝给 HC[i]接着大循环 i，直到所有 n 个元素的编码给定。

```
//n：输入，权重非零字符数
//HT：输出，赫夫曼树
//HC：输出，赫夫曼树对应的赫夫曼编码
```

```
//w：输出，字符对应权值（>0）组成的数组
//作用：生成赫夫曼树和赫夫曼编码
void HuffmanCoding(HuffmanTree* HT, HuffmanCode* HC, int* w, int n) {//w存放n个字符权值（>0）构造赫夫曼树HT并求出对应赫夫曼编码HC
    if (n <= 0)return;
    int m = 2 * n - 1;//赫夫曼树结点个数
    (*HT) = (HuffmanTree)malloc(m * sizeof(HTNode));
    HuffmanTree p = (*HT);
    int i;
    for (i = 0; i < n; ++i, ++p, ++w) {//初始化p，且每个节点的左右孩子为-1
        p->weight = *w;
        p->parent = 0;
        p->lchild = p->rchild = -1;
    }
    for (; i < m; ++i, ++p) {//初始化p，生成节点的左右孩子为0
        p->weight = p->parent = p->lchild = p->rchild = 0;
    }
    for (i = n; i < m; ++i) {//构建赫夫曼树
        unsigned int s1, s2;
        Select((*HT), i, &s1, &s2);
        (*HT)[s1].parent = i; (*HT)[s2].parent = i; (*HT)[i].lchild = s1; (*HT)[i].rchild = s2;
        (*HT)[i].weight = (*HT)[s1].weight + (*HT)[s2].weight;
    }
    (*HC) = (HuffmanCode)malloc(n * sizeof(char*));//分配n个字符编码的头指针向量
    char* cd = (char*)malloc(n * sizeof(char));//分配求编码的工作空间
    cd[n - 1] = '\0';//编码结束符
    int start, f, c;
    for (i = 0; i < n; ++i) {//逐个字符求赫夫曼编码
        start = n - 1;
        for (c = i, f = (*HT)[i].parent; f != 0; c = f, f = (*HT)[f].parent) {//逆向求编码
            if ((*HT)[f].lchild == c) cd[--start] = '0';//左右子树分别为0和1
            else cd[--start] = '1';
        }
        (*HC)[i] = (char*)malloc((n - start) * sizeof(char));//为第i个字符分配空间
        strcpy((*HC)[i], &cd[start]);
    }
    free(cd);//释放工作空间
}//HuffmanCoding
```

3.2.2 构造函数 Select()

函数 Select()用于找出目前无双亲结点，且权值最小的两个结点。

3.3 根据赫夫曼编码表进行译码

各字符的权值通过函数 frequency()得到并按 ASCII 所对应大小按顺序存放在 count[]数组中，根据 count[]数组将赫夫曼编码表也按照 ASCII 转换为二维数组 res[][]，通过读取需要译

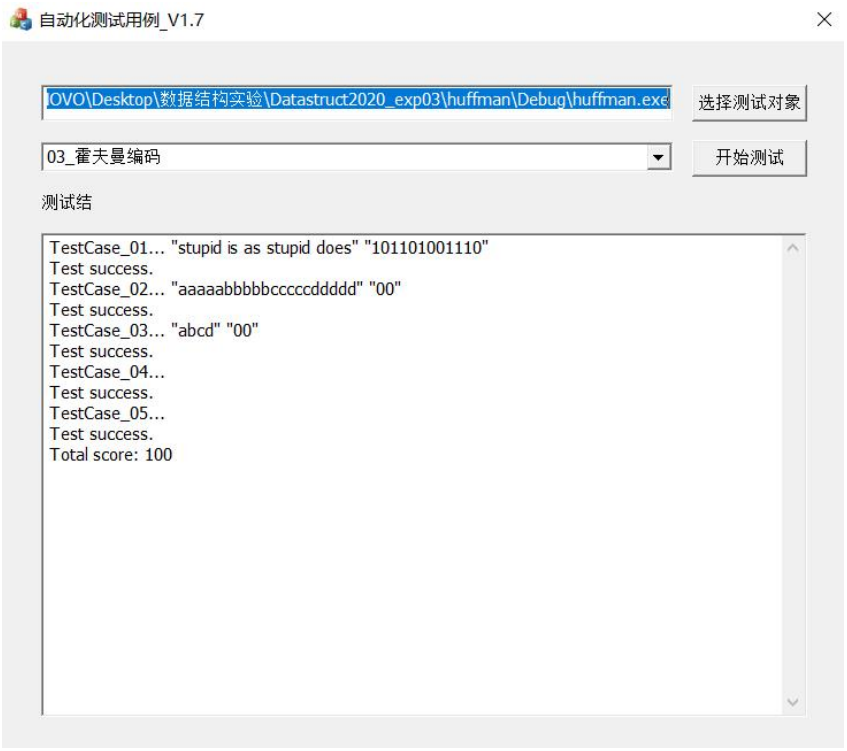
码的字符串，对应的 ASCII 值大小，即可找到对应译码所在 res 数组中的位置，并得到该字符的赫夫曼编码。

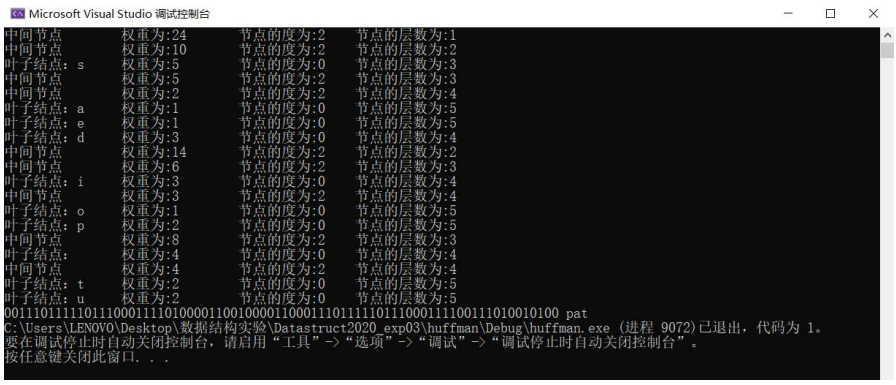
```
char dec[MAXSIZE] = {0};//解码字符串初始化
strcpy_s(dec, MAXSIZE, argv[2]);
dec[strlen(dec)] = '\0';
char* s = dec;
int p;
while (*s != '\0') {
    //解码失败
    if (*s != '0' && *s != '1'){
        printf("\nERROR_03");//解码失败，输出ERROR_03
        return 0;
    }
    p = 2 * n - 2;
    while (HT[p].lchild != -1 || HT[p].rchild != -1) {
        if (*s == '0') p = HT[p].lchild;
        else p = HT[p].rchild;
        ++s;
    }
    printf("%c", c[p]);//输出计算结果：解码结果
}
//解码失败
if (HT[p].lchild != -1 || HT[p].rchild != -1) {//解码失败，输出ERROR_03
    printf("\nERROR_03");
    return 0;
}
```

3.4 实现先序遍历赫夫曼树

除了叶子结点外，其余结点均由字符"#"代替。由于所建立的赫夫曼树已包含各结点的双亲结点和孩子结点的位置信息，且赫夫曼数组中最后一位为该赫夫曼树的根结点，所以只需从最后一位开始读取。先序遍历特点为：先读取结点信息，再读取左孩子结点信息，再读取右孩子结点信息；此处额外定义一个 ccount1[]数组，用于统计每个结点在遍历时被访问次数，若结点 i 未被访问，则 ccount1[i]的值为 0，若仅访问过该结点信息，ccount[i]的值为 1，并接着访问该结点的左孩子，若左右孩子结点均已访问，则 ccount 为 2，下一步访问该结点的双亲结点。由此可实现赫夫曼树的先序遍历。

四 运行结果





五 问题思考

在 main 函数里直接使用函数 free()未能释放二级指针所指空间，需要对该指针所指的结构体的每一部分调用函数 free()功能，才能成功释放；函数 malloc()和函数 free()使用后需判断是否成功分配空间/释放空间。

六 源代码

huffman.c

实验四无向图最短路径搜索

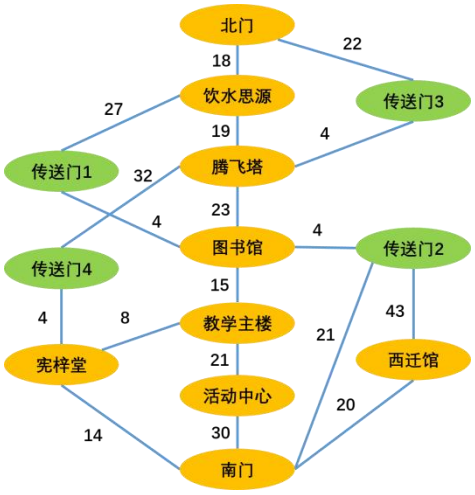
一 实验目的

掌握 Dijkstra 算法的原理。

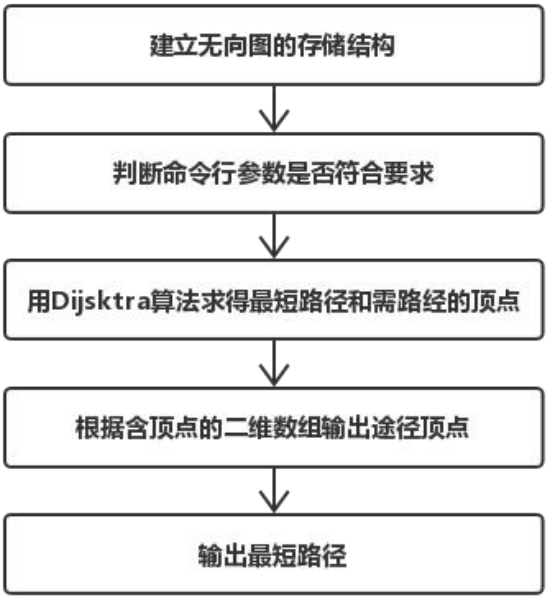
二 实验要求

① 熟练掌握图的操作。

注:(1)命令行参数不正确输出 ERROR_01;(2)获取最短路径失败时输出 ERROR_02;(3)获取最短路径成功时输出最短路径以及路径长度。



三 设计思路



3.1 数据结构

先要建立一个无向图，用于存储该图顶点数以及带权邻接矩阵。

```
typedef struct {                // 邻接矩阵
    int vexnum;                         // 顶点数
    int arcs[MAX][MAX];     //带权邻接矩阵
}MGraph;
```

3.2Dijsktra 算法

首先声明一个一维数组 dist 来保存源点到各个顶点的最短距离和一个保存已经找到了最短路径的顶点的集合的二维数组 pre。初始时，源点 v0 的路径权重被赋为 0（dis[v0]=0）。若对于源点 v0 和顶点 i 存在能直接到达的边（v0,i），则把 dist[i]设为 matrix[v0][i],同时把所有其他（v0 不能直接到达的）顶点的路径长度设为无穷大。

初始时，pre 数组中为 1 的部分只有顶点 v0 和其他能直接到达的顶点。然后，从 dist 数组选择最小值，则该值就是源点 v0 到该值对应的顶点的最短路径，并且把该点加入到 pre 中，此时完成一个顶点，然后，需要检验新加入的顶点是否可以到达其他顶点，并且通过该顶点到达其他点的路径长度是否比源点直接到达短，如果是，那么就替换这些顶点在 dist 中的值。

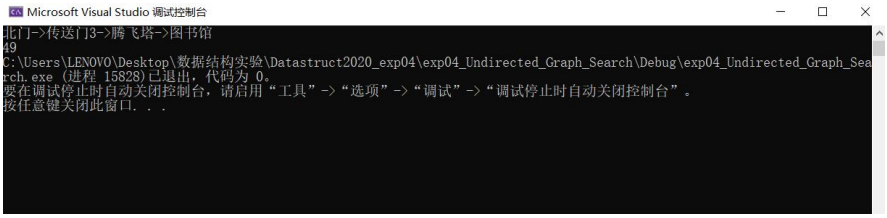
然后，又从 dist 中找出最小值，重复上述动作，直到 T 中包含了图的所有顶点。

```
void ShortestPath_DIJ(MGraph G, int v0, int vd) {//v0 为初始点，vd 为目的地
    int v = 0, w = 0, i = 0;
    int final[MAX] = { 0 };        //final[i]=TRUE 表示已经求得 v0 到 i 的最短路径
    int Path[MAX][MAX] = { 0 };      //Path[i][j]=TRUE 表示从"顶点 v0"到 i 的当前最短路径包含顶点 j
    int D[MAX] = { INF };          // D[i]表示当前从"顶点 v0"到 i 的最短路径长度
    // 初始化
    for (v = 0; v < G.vexnum; ++v) {
        final[v] = FALSE;// 顶点 v 的最短路径还没获取到
        for (w = 0; w < G.vexnum; ++w) Path[v][w] = FALSE; //设空最短路径
```

```
            D[v] = G.arcs[v0][v];// 顶点 v 的最短路径为"顶点 v0"到"顶点 i"的权
            if (D[v] < INF) {
                Path[v][v0] = TRUE;
                Path[v][v] = TRUE;
            }//if
        }//for
        D[v0] = 0;
        final[v0] = TRUE;//初始化 v0,属于 S 集
        //开始主循环，每次求得 v0 到某个顶点的最短路径并加 v 到 S 集
        int min = INF;
        int s = 0;//s 作为标记
        int m = 0;
        for (i = 1; i < G.vexnum; ++i) {// 遍历 G.vexnum-1 次,在未获取最短路径的顶点中，找到离 v0 最近的顶点(k)。
            min = INF;
            for (w = 0; w < G.vexnum; ++w)
                if (final[w] != 1)//w 在 V-S 中
                    if (min > D[w]) {//w 离 v0 更近
                        s = w;
                        min = D[w];
                    }
            final[s] = TRUE;//离 v0 最近的 v 加入 S
            for (w = 0; w < G.vexnum; ++w)//更新当前最短路径及距离
                if (final[w] != 1 && (min + G.arcs[s][w] < D[w])) {
                    D[w] = min + G.arcs[s][w];
                    for (m = 0; m < G.vexnum; m++) Path[w][m] = Path[s][m];
                    Path[w][w] = TRUE;
                }//if
        }//for
        //地名字符串数组
        char name[MAX][MAX] = { "北门","饮水思源","腾飞塔","图书馆","教学主楼","活动中心","南门","宪梓堂","西迁馆","传送门 1","传送门 2","传送门 3","传送门 4" };
        // 打印 dijkstra 最短路径的结果
        if (D[vd] != INF) {//有最短路径
            int k = 1;
            int j = 0;
            int min = INF;
            int tmp = v0;
            if (v0 == vd) {
                printf("%s->%s", name[v0], name[vd]);
            }
            else {
                while (k == 1) {
                    if (tmp != vd) {
```

```
                printf("%s->", name[tmp]);
            }
            else {
                printf("%s", name[tmp]);
            }
            Path[vd][tmp] = 0;//已输出该路径，删除
            k = 0;
            for (j = 0; j < MAX; j++) {//求中间经过地点 tmp
                if (Path[vd][j] == 1 && D[j] != INF && D[j] < min) {
                    min = D[j];
                    tmp = j;
                    k = 1;
                }
            }
            min = INF;
        }//while
    }
    printf("\n%d", D[vd]);
  }
}//ShortestPath_DIJ
```

四 实验结果



五 问题与思考

可采用已有的图，在 main 函数中直接调用该图；由于 Dijsktra 算法求得的是源点到其余所有顶点的最短距离和路径，在输出最短路径思考可将 pre 数组中为 1 的部分替换成按路经顺序的编号（如“1，2，3……”），在最后输出路径时会更加方便。

六 源代码

exp04_Undirect
ed_Graph_Search