目录

实验 1 对指令操作码进行哈夫曼编码

一、问题描述

使用编程工具编写一个程序，对一组指令进行霍夫曼编码，并输出最后的编码结果以及对指令码的长度进行评价。与扩展操作码和等长编码进行比较。

例如，给定一组指令的操作码以及各操作码出现的概率，如下表所示：

表 1-1 操作码表

P1	P2	P3	P4	P5	P6	P7
0.45	0.30	0.15	0.05	0.03	0.01	0.01

对此组指令进行 HUFFMAN 编码正如下图所示：

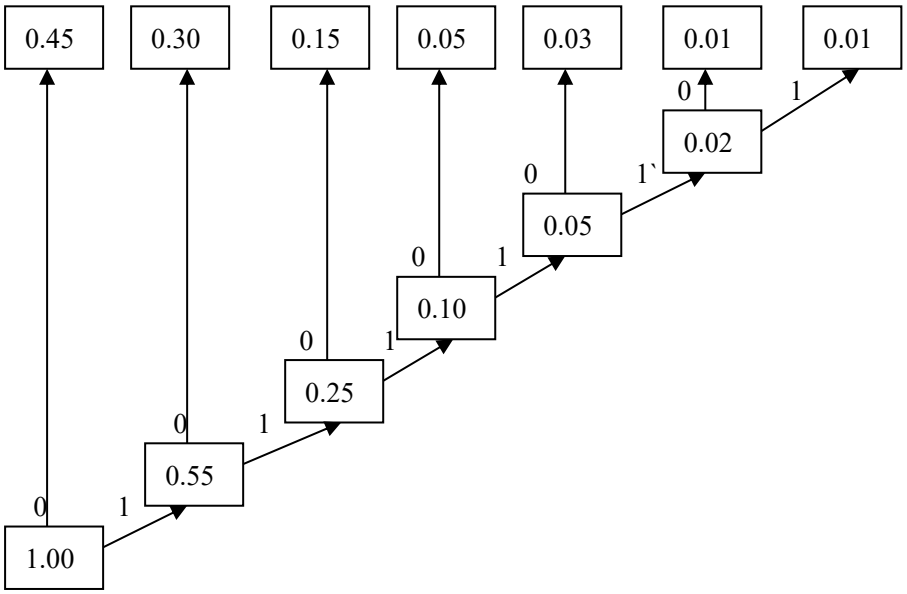


图 1-1

最后得到的 HUFFMAN 编码如下标所示：

表 1-2 哈夫曼编码表

P1	P2	P3	P4	P5	P6	P7
0	10	110	1110	11110	111110	111111
1	2	3	4	5	6	6

最短编码长度为：

H=0.45*1+0.30*2+0.15*3+0.05*4+0.03*5+0.01*6+0.01*6=-1.95.

要对指令的操作码进行 HUFFMAN 编码，只要根据指令的各类操作码的出现概率构造 HUFFMAN 树再进行 HUFFAM 编码。此过程的难点构造 HUFFMAN 树，进行 HUFFAM 编码只要对你所生成的 HUFFMAN 树进行中序遍历即可完成编码工作。

二、设计简要描述

本实验主要要求我们利用已给的指令以及概率进行哈夫曼树的构造以及编码。程序中包含的结构体和类有：哈夫曼结点类（该类包含了哈夫曼结点的孩子节点信息，代表的指令标号，指令对应的概率）。构建哈夫曼树时需要进行最小概率结点的查找函数；删除两最小概率结点，插入新节点的函数；编码函数；计算平均编码字长函数。

（1）函数说明：

```
f_min_p* input_instruct_set();//输入指令集子模块;
huff_p* creat_huffman_tree(f_min_p* head);//构造 huffman 树;
f_min_p* fin_min(f_min_p* h);        //在工作链表中寻找最小概率节点函数。
f_min_p* del_min(f_min_p* h,f_min_p* p);//在工作链表中删除最小概率节点函数。
void insert_n(f_min_p* h,f_min_p* p);// 在工作链表中插入两个最小概率节点生成节点函数。
huff_p*  creat_huffp(f_min_p* p);//生成 HUFFMAN 节点。
void creat_huffman_code(huff_p* h1,huff_code* h);//生成 huffman 编码;
void r_find(huff_p* p1,char code[],int i,huff_code* h);
//遍历 HUFFMAN 树生成指令操作码的 HUFFMAN 编码。
void output_huffman(huff_code* head);//输出 huffman 编码;
void cal_sort_length(huff_code* head);//计算指令用 huffman 编码的平均编码字长
```

（2）程序流程图

该程序的整体流程图如下，

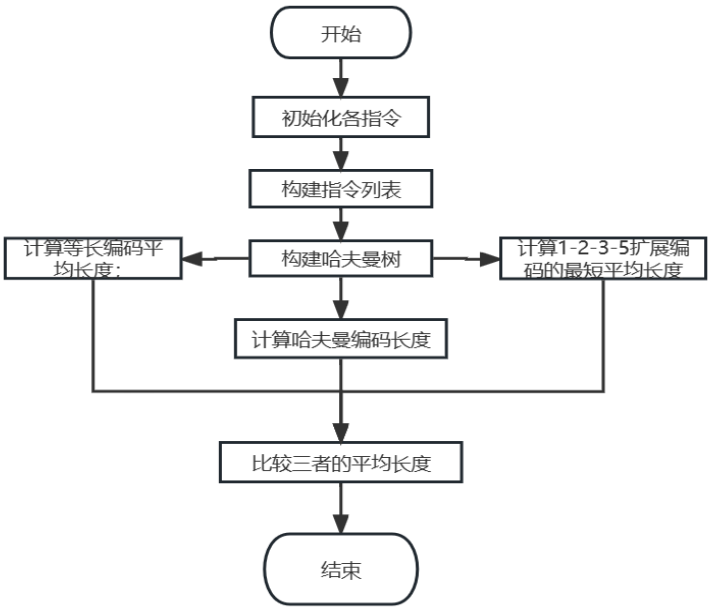


图 1-2

其中哈夫曼树构建过程的程序流程图如下，

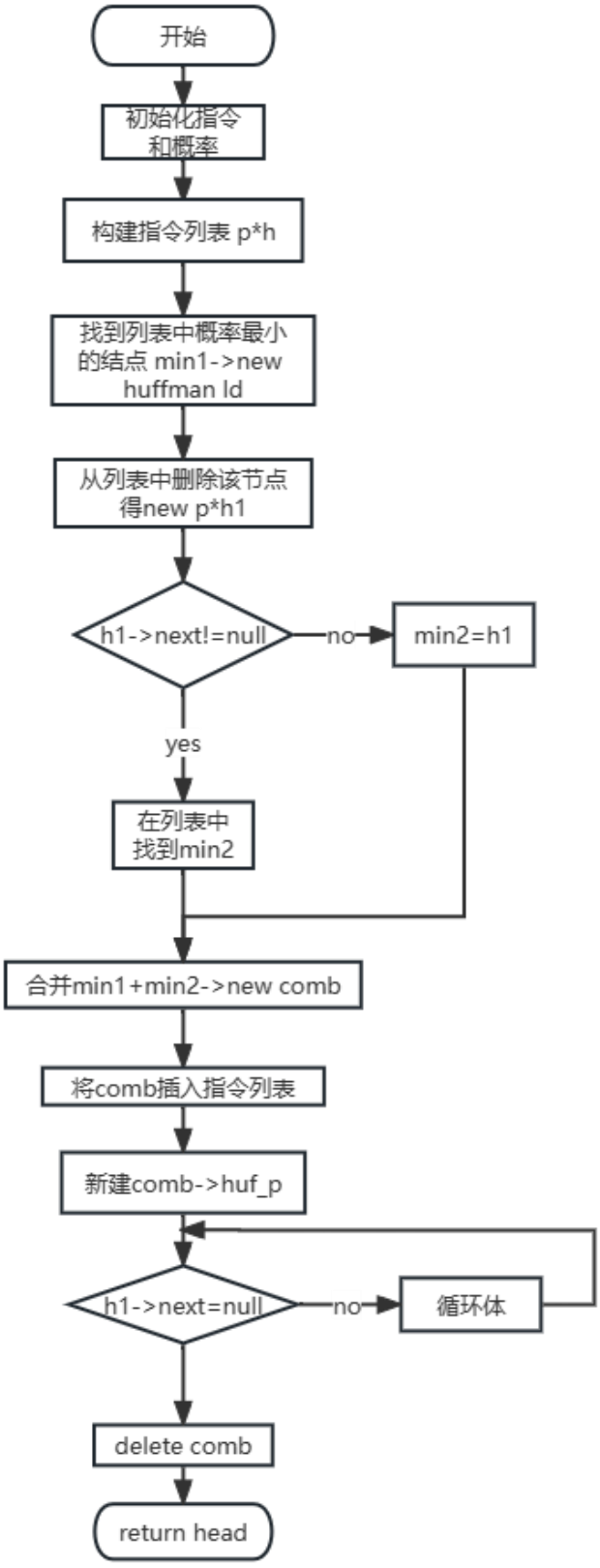


图 1-3

三、程序清单

```
#include<iostream>
#include<cmath>
#include<cstring>

using namespace std;

const int N = 8; // huffman 编码最大长度
class huff_p {
public:
	huff_p* r_child;   // 大概率的节点
	huff_p* l_child;   // 小概率的节点
	char op_mask[3];   // 指令标号
	float p;           // 指令使用概率
};

//f_min_p 链表依次存放着输入的指令
class f_min_p {
public:
	f_min_p* next;
	char op_mask[3];    // 指令标号
	float p;            // 指令使用概率
	huff_p* huf_p;
};

// huff_man code
class huff_code {
public:
	huff_code* next;
	float p;
	char op_mask[3];
	char code[N];       // huffman 编码
};

f_min_p* input_instruct_set(); // 输入指令集子模块；
huff_p* creat_huffman_tree(f_min_p* head); // 构造 huffman 树;
f_min_p* fin_min(f_min_p* h);
f_min_p* del_min(f_min_p* h, f_min_p* p);
void insert_n(f_min_p* h, f_min_p* p);
huff_p* creat_huffp(f_min_p* p);
void creat_huffman_code(huff_p* h1, huff_code* h);// 生成 huffman 编码;
void r_find(huff_p* p1, char code[], int i, huff_code* h);
```

```
void output_huffman(huff_code* head);// 输出 huffman 编码;
void cal_sort_length(huff_code* head);// 计算指令用 huffman 编码的平均编码字长
void print(huff_p* h1);

int main() {
	f_min_p* h, * h1;
	huff_p* root;
	huff_code* head, * pl;

	h = input_instruct_set();
	h1 = h;
	root = creat_huffman_tree(h1);

	head = new huff_code;
	head->next = NULL;
	creat_huffman_code(root, head);

	output_huffman(head);
	cal_sort_length(head);

	pl = head->next;
	while (pl) {
		delete head;
		head = pl;
		pl = pl->next;
	}
}

//构建指令列表
f_min_p* input_instruct_set() {
	f_min_p* head;
	f_min_p* h;
	h = new f_min_p;
	h->next = NULL;
	h->huf_p = NULL;
	head = h;
	int n;

	cout << "请输入指令数:";
	cin >> n;
	while(n<=1){
		cout << "请输入指令数:";
		cin >> n;
	}
```

```
        cout << "请输入指令标号：";
        cin >> h->op_mask;
        cout << "请输入指令的使用概率：";
        cin >> h->p;

        f_min_p* point;
        f_min_p* p1 = head;

        for (int i = 0; i < n - 1; i++) {
            point = new f_min_p;
            cout << "请输入指令标号：";
            cin >> point->op_mask;
            point->op_mask[2] = '\0';
            cout << "请输入指令的使用概率：";
            cin >> point->p;
            point->huf_p = NULL;
            point->next = p1->next;
            p1->next = point;
            p1 = point;
        }
        return head;
}


//构建哈夫曼树
huff_p* creat_huffman_tree(f_min_p* h) {
        f_min_p* h1, * min1, * min2, * comb;
        huff_p* head, * rd, * ld, * parent;
        h1 = h;
        min1 = fin_min(h1);//找到概率最小的结点
        ld = creat_huffp(min1);//新建一个哈夫曼结点
        h1 = del_min(h1, min1);//删除该节点  后得到的新的链表头

        if (h1->next) {

            min2 = fin_min(h1);
        }
        else {
            min2 = h1;
        }

        rd = creat_huffp(min2); //又造一个新节点，与上一个结点合并
        comb = new f_min_p;
```

```
comb->next = NULL;
comb->p = rd->p + ld->p;
comb->op_mask[0] = '\0';
comb->op_mask[1] = '\0';

parent = creat_huffp(comb);

insert_n(h1, comb);
if (h1->next != NULL) {
	h1 = del_min(h1, min2);
}

parent->l_child = ld;
parent->r_child = rd;

comb->huf_p = parent;

head = parent;

while (h1->next != NULL) {
	min1 = fin_min(h1);
	if (min1->huf_p == NULL) {
		ld = creat_huffp(min1);
	}
	else {
		ld = min1->huf_p;
	}
	h1 = del_min(h1, min1);

	if (h1->next) {
		min2 = fin_min(h1);
	}
	else {
		min2 = h1;
	}
	if (min2->huf_p == NULL) {
		rd = creat_huffp(min2);
	}
	else {
		rd = min2->huf_p;
	}
	comb = new f_min_p;
	comb->next = NULL;
	comb->p = rd->p + ld->p;
```

```
            comb->op_mask[0] = '\0';
            comb->op_mask[1] = '\0';
            parent = creat_huffp(comb);

            //此时结点二没删
            if (h1 != NULL) {//如果待取列表还没有取完则将新节点插入
                insert_n(h1, comb);
            }

            if (h1->next != NULL) {
                //如果删了 h2 后还有结点
                h1 = del_min(h1, min2);
            }
            //如果删除第二小之后就再也没有其他节点，
            if(h1->next==NULL){
                if(ld->p<rd->p){
                    huff_p* tmp = ld;
                    ld = rd;
                    rd = tmp;
                }
            }

            parent->l_child = ld;
            parent->r_child = rd;
            comb->huf_p = parent;
            head = parent;

            //
            if (h1->next == NULL) {
                break;
            }
        }
        //循环结束，此时只剩一个结点 comb
        delete comb;
        return head;
    }


    //找到当前链表中概率最小的结点
    f_min_p* fin_min(f_min_p* h) {
        f_min_p* h1, * p1;
        h1 = h;
        p1 = h1;
        float min = h1->p;
```

```
        h1 = h1->next;
        while (h1) {
            if (min > (h1->p)) {
                min = h1->p;
                p1 = h1;
            }
            h1 = h1->next;
        }
        return p1;
}


//删除当前链表中概率最小的结点
f_min_p* del_min(f_min_p* h, f_min_p* p) {
        f_min_p* p1, * p2;
        p1 = h;
        p2 = h;
        if (h == p) {
            h = h->next;
            delete p;
        }
        else {
            while (p1->next != NULL) {
                p1 = p1->next;
                if (p1 == p) {
                    p2->next = p1->next;
                    delete p;
                    break;
                }
                p2 = p1;
            }
        }
        return h;//这里可能删除之后返回的列表为空
}


void insert_n(f_min_p* h, f_min_p* p1) {
        p1->next = h->next;
        h->next = p1;
}


huff_p* creat_huffp(f_min_p* d) {
        huff_p* p1;
```

```
        p1 = new huff_p;
        p1->l_child = NULL;
        p1->r_child = NULL;
        p1->p = d->p;
        p1->op_mask[0] = d->op_mask[0];
        p1->op_mask[1] = d->op_mask[1];
        return p1;
}

void r_find(huff_p* p1, char code[], int i, huff_code* h) {
        if (p1->l_child) {
                code[i] = '1';
                r_find(p1->l_child, code, i + 1, h);
        }
        if (p1->op_mask[0] != '\0') {
                huff_code* p2 = new huff_code;
                p2->op_mask[0] = p1->op_mask[0];
                p2->op_mask[1] = p1->op_mask[1];
                p1->op_mask[2] = '\0';
                p2->p = p1->p;
                int j = 0;
                for (; j < i; j++) {
                        p2->code[j] = code[j];
                }
                p2->code[j] = '\0';
                p2->next = h->next;
                h->next = p2;
        }
        if (p1->r_child) {
                code[i] = '0';
                r_find(p1->r_child, code, i + 1, h);
        }
        delete p1;
}

void creat_huffman_code(huff_p* h1, huff_code* h) {
        int i = 0;
        char code[N] = { '\0' };
        r_find(h1, code, i, h);
}
void output_huffman(huff_code* head) {
        huff_code* h = head->next;
        cout << "指令\t" << "概率\t" << "编码" << endl;
        cout << "---------------------------------" << endl;
```

```
    while (h) {
        h->op_mask[2] = '\0';
        cout << h->op_mask << ":\t" << h->p << "\t" << h->code << endl;
        h = h->next;
    }
    cout << "---------------------------------" << endl;
    cout << endl;
}
void cal_sort_length(huff_code* head) {
    huff_code* h = head->next;
    double j = 0;
    float one_length = 0;
    float per_length = 0;
    float ext_length = 0;//按 1-2-3-5 扩展编码的最小长度为。

    while (h) {
        float length = 0;
        int i = 0;
        while (h->code[i] != '\0') {
            length++;
            i++;
        }
        one_length = h->p * length;
        per_length = per_length + one_length;
        h = h->next;
        j++;
    }
    int i1 = int(j);
    huff_code* p2 = head->next;
    float* p_a = new float[i1];
    //sort 指令概率
    int i0 = 0;
    while (p2) {
        p_a[i0++] = p2->p;
        p2 = p2->next;
    }
    float max, temp;
    int l;
    for (int s = 0; s < i1; s++) {
        max = p_a[s];
        l = s;
        for (int k = s + 1; k < i1; k++) {
            if (max < p_a[k]) {
                max = p_a[k];
```

```
                    l = k;
                }
            }
            temp = p_a[s];
            p_a[s] = max;
            p_a[l] = temp;
        }
        //计算 1-2-3-5 扩展编码的最短平均长度
        float* code_len = new float[i1];
        code_len[0] = 1;
        code_len[1] = 2;
        code_len[2] = 3;
        code_len[3] = 5;
        for (int i = 4; i < j; i++) {
            code_len[i] = 5;
        }
        l = 0;
        while (l < i1) {
            ext_length = ext_length + code_len[l] * p_a[l];
            l++;
        }

        //计算等长编码平均长度；
        int q_length = ceil(log10(j) / log10(2));//向上取整

        cout << "此指令集操作码 huffman 编码的平均长度为：" << per_length << endl;
        cout << "等长编码的平均长度为：" << q_length << endl;
        cout << "按 1-2-3-5 的扩展编码的最短平均编码长度为：" << ext_length;
        cout << endl;
        cout << endl;
        if (q_length >= per_length) {
            cout << "可见 huffman 编码的平均长度要比等长编码的平均长度短" << endl;
        }
        else {
            cout << "huffman 编码有问题请仔细查看算法，以及输入的指令集的概率之和是否大于 1。" << endl;
        }
        if (ext_length > per_length) {
            cout << "可见 huffman 编码的平均长度要比 1-2-3-5 扩展编码的最短平均长度短" << endl;
        }
        else {
            cout << "huffman 编码有问题请仔细查看算法，以及输入的指令集的概率之和是否大于 1。" << endl;
```

```
    }
}
```

四、结果分析

1、原始图示：

输入要求：总指令数（本实验中必须大于 1）、各指令的指令名和指令出现的概率（各指令出现的概率之和必须等于 1）。

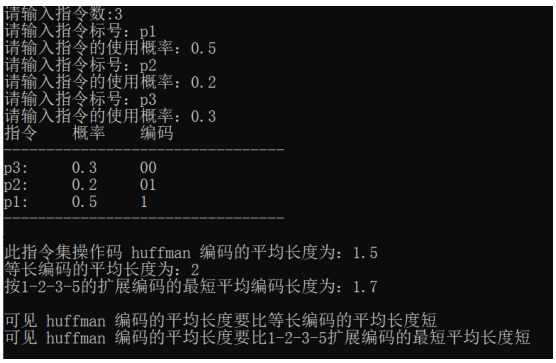
2、测试数据输入及输出

指令数：3

指令编号:p1 指令使用概率：0.5

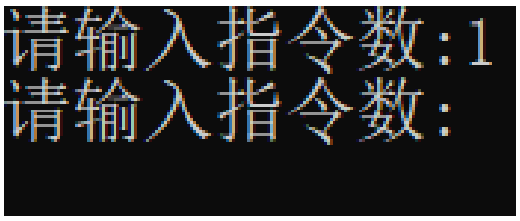
指令编号：p2 指令使用概率：0.2

指令编号：p3 指令使用概率：0.3



指令数：1

当指令数输入为 1 时，程序默认改输入无效，需要重新输入。因为在程序的书写中初次查找两个概率最小的指令数时，**默认指令的个数大于 1**。这也保障了代码的正确性。

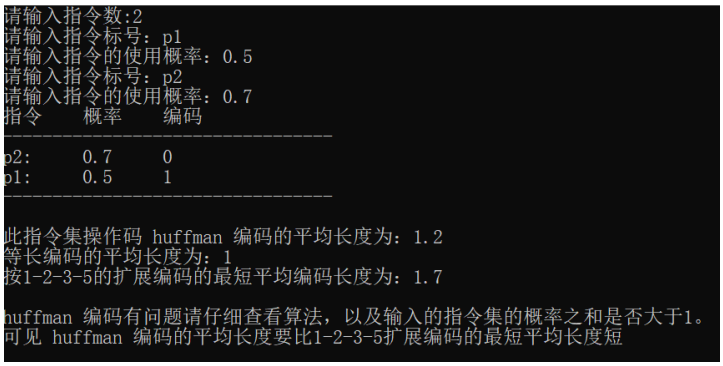


指令数：2

指令编号：p1 指令使用概率：0.5

指令编号：p2 指令使用概率：0.7

此测试数据中输入的指令集使用概率之和大于 1，会显示该编码有问题。

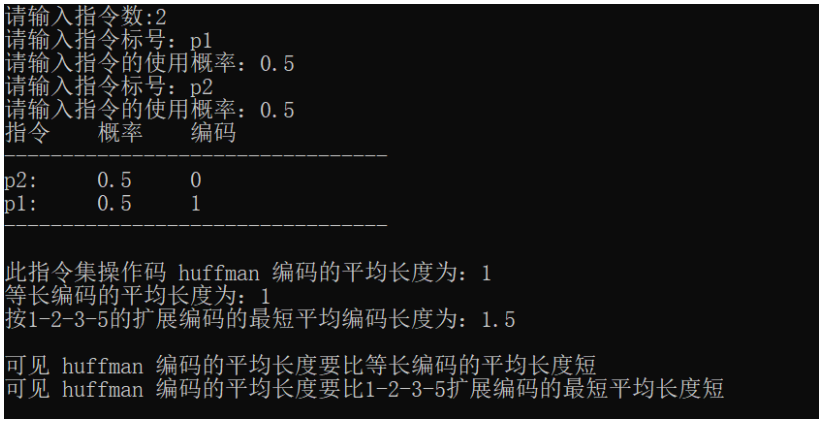


指令数：2

指令编号:p1 指令使用概率：0.5

指令编号：p2 指令使用概率：0.5

当输入的指令数为 2，各概率相等时两叶子节点分别作为根节点的左右孩子。



3、正确性分析

通过对源代码检验，输入不同范围的测试数据并观察其运行结果可以看出，当输入的指令数为 1 时，程序能够识别出并要求重新输入；当输入的指令的概率之和大于 1 时，程序会识别出并指出错误。对于不同的输入得到的 Huffman 编码、等长编码和 1-2-3-5 扩展编码之间的关系经检验也符合事实。可见该程序具有如下特点：

- 正确性
- 鲁棒性

五、调试报告

在调试运行该程序的过程中，也遇到了许多问题，最终一一解决。

1、调试问题分析

问题一：在调试中遇到了一个十分棘手的问题，小范围改变指令个数，并且正确的输入指令的概率之和小于等于 1 却总是出现 huffman 编码有问题的错误提醒。

问题二：当输入的指令条数为 1 时，程序卡顿不动，出现问题。

2、问题解决

在解决问题一时，经过对代码书写的检查发现在进行等长编码计算时并没有向下取整。同时由于输入的指令数较小，所以取整后可能导致二者所得的长度一样。在最后进行错误检验的分支判断中，如果二者长度一样会被判断为错误，故对此判断进行修正。当二者的长度一样时仍会认为是正确的。

当输入的指令条数为 1 时，会出现 bug。发现在进行哈夫曼树构建时，默认结点数超过一个，当指令数只有一个时会出现 null->next 的情况，故限制程序的指令数输入必须大于 1。

六、经验与体会

本实验总共回顾了三种我们学过的编码方式，尤其回顾了在数据结构课程中学习的哈夫曼编码。哈夫曼编码涉及到了哈夫曼树的构建其中还涉及到了中序遍历。在进行本实验前我对哈夫曼编码的理论内容又进行了回顾。

本实验中较大的难度在于指令 huffman 树的构建以及最终编码长度的计算。在调试过程中遇到的绝大多数问题也都在于此。好在通过一步步调试后最终都得到了解决。

但由于本程序输入形式的限制没有尝试更大的指令个数，就测试来看，五个指令以内哈夫曼编码的平均长度要小于等长编码和扩展编码的平均编码长度。

在观摩过别的同学对于同一个程序的理解后，我也意识到本程序的编写 其实是很有多样性的，不该被局限在自己的认知当中。

实验 2 使用 LRU 方法更新 Cache

一、 问题描述

使用 LRU 策略，对一组访问序列进行内部的 Cache 更新。

LRU 置换算法是选择最近最久未使用的页面予以置换。该算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来经历的时间 T，当须淘汰一个页面时，选择现有页面中 T 值最大的，即最近最久没有访问的页面。这是一个比较合理的置换算法。

举例说明此问题，例如：

有一个 CACHE 采用组相连映象方式。每组有四块，为了实现 LRU 置换算法，在快表中为每块设置一个 2 位计数器。我们假设访问序列为“1,1,2,4,3,5,2,1,6,7,1,3”。在访问 CACHE 的过程中，块的装入，置换及命中时，具体情况如下表所示：

表 2-1 块的装入、置换、命中情况

	1	1	2	4	3	5	2	1	6	7	1	3
Cache 块 0	1	1	1	1	1	5	5	5	5	7	7	7
Cache 块 1			2	2	2	2	2	2	2	2	2	3
Cache 块 2				4	4	4	4	1	1	1	1	1
Cache 块 3					3	3	3	3	6	6	6	6
	装入	命中	装入	装入	装入	置换	命中	置换	置换	置换	命中	置换

由该表格可以看出，Cache 块一次是 0、1、2、3 四块，当块有空闲时则按顺序依次放入空闲块。并且为该块设立 T 值。当四个块都已经占满时，则会按顺序查找块中 T 值最大的块进行置换，置换后重设该块的 T 值为 0，表示其具有不被置换的最高优先级。

二、设计简要描述

Cache 类：定义了当前 cache 块的状态，存储的数值，属性 T 值（count 值）。

常量设置：cache 块数为 4，测试的页面数为 12，walk_sort 里存放的是工作集。

功能实现：需要实现了当前 cache 块的是否有空闲块的判断；更新插入块以及其他块的 T；当前数已经存在在 cache 内则直接放入并且更新 T；当前块已满时需要搜索 T 最大的 cache 并将该内容替换；当前块未满且内容不存在时直接放入空闲块。

（1）函数说明

main 函数:整个程序的入口，调用了 up_cache 函数

up_cache 函数： 置换算法实现函数.

（2）程序流程图

该程序的主要功能函数为 up_cache 函数，起到的功能是实现 LRUcache 块置换算法。该程序的主要实现流程图如下，

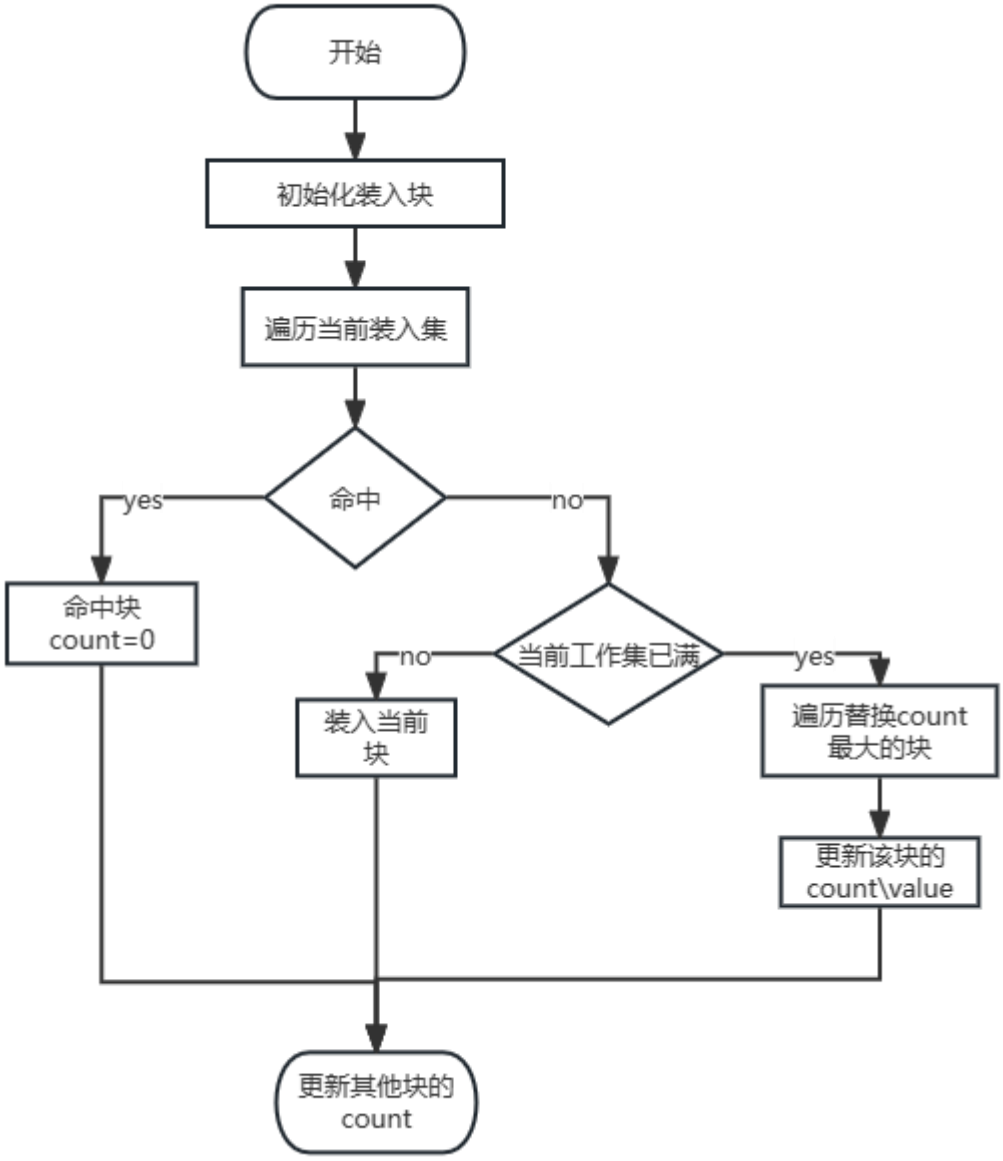


图 2-1 lru 算法单次流程图

三、程序清单

```
#include <iostream>
```

```
using namespace std;

class Cache {
public:
	bool state = false;
	int value = -1;
	int count = 0;
};

const int M = 4; // Cache 块数
Cache cache[M];
const int N = 12;// 测试页面数
int walk_sort[] = {1,1,2,4,3,5,2,1,6,7,1,3};// 测试数据
void up_cache();

int main() {
	up_cache();
}

void up_cache() {
	int i = 0;
	while (i < N) {
		int j = 0;
		// 满么?
		while (j < M) {
			if((cache[j].state == false) && (walk_sort[i] != cache[j].value)) {
				cout << "cache 有空闲块,不考虑是否要置换..." << endl;
				cout << walk_sort[i] << "被调入 cache...." << endl;
				cache[j].value = walk_sort[i++];
				cache[j].state = true;
				cache[j].count = 0;
				int kk = 0;

				for (int x = 0; x < M; x++) {
					cout << "cache 块" << x << ": " << cache[x].value << endl;
				}
				cout << endl;

				// 更新其它 cache 块没使用时间
				while (kk < M) {
					if (kk != j && cache[kk].value != -1) {
						cache[kk].count++;
					}
```

```
                kk++;
            }
            break;
        }
        if (cache[j].value == walk_sort[i]) {
            cout << endl;
            cout << walk_sort[i] << "命中!!!" << endl;

            for (int x = 0; x < M; x++) {
                cout << "cache 块" << x << ": " << cache[x].value << endl;
            }
            cout << endl;

            int kk = 0;
            i++;
            cache[j].count=0;
            //更新其它 cache 块没使用时间
            while (kk < M) {
                if (kk != j && cache[kk].value != -1) {
                    cache[kk].count++;
                }
                kk++;
            }
        }
        j++;
    }

    if (j == M) {
        cout << "cache 已经满了,考虑是否置换..." << endl;
        cout << endl;
        int k = 0;

        while (k < M) {
            if (cache[k].value == walk_sort[i]) {
                cout << endl;
                cout << walk_sort[i] << "命中!!!" << endl;

                for (int x = 0; x < M; x++) {
                    cout << "cache 块" << x << ": " << cache[x].value << endl;
                }

                i++;
                cache[k].count = 0;
                int kk = 0;
```

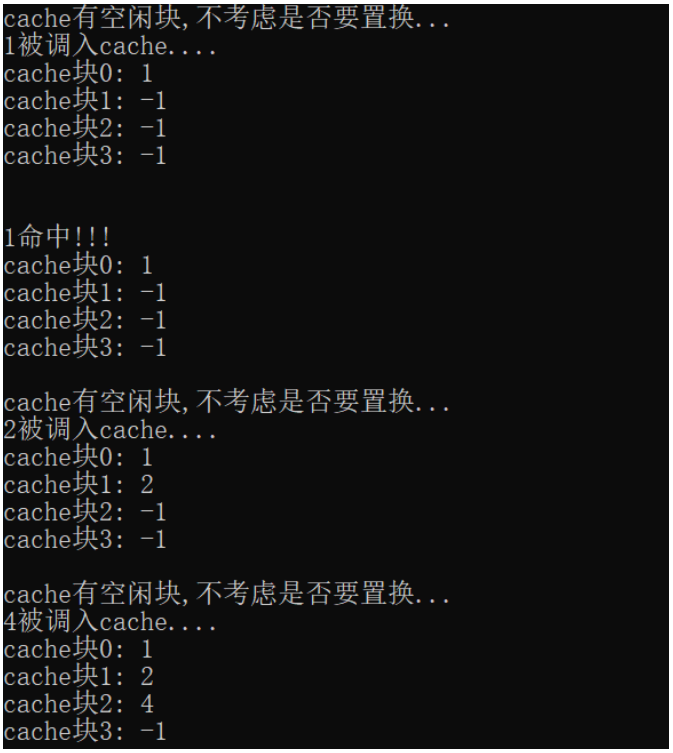
```
                    //更新其它 cache 块没使用时间
                    while (kk < M) {
                        if (kk != k){
                            cache[kk].count++;
                        }
                        kk++;
                    }
                    break;
                }
                k++;
            }
            //考虑置换那一块.
            if (k == M) {
                int ii = 0;
                int t = 0;//要替换的 cache 块号.
                int max = cache[ii].count;
                ii++;
                while (ii < M) {
                    if(cache[ii].count > max) {
                        max = cache[ii].count;
                        t = ii;
                    }
                    ii++;
                }
                //置换
                cout<<cache[t].value<<"被"<<walk_sort[i]<<"在  cache  的"<<t<<"号块置
换..."<<endl;
                cache[t].value=walk_sort[i++];
                cache[t].count=0;

                for (int x = 0; x < M; x++) {
                    cout << "cache 块" << x << ": " << cache[x].value << endl;
                }
                int kk = 0;
                //更新其它 cache 块没使用时间
                while (kk < M) {
                    if (kk != t) {
                        cache[kk].count++;
                    }
                    kk++;
                }
            }
        }
```

```
    }
}
```

四、结果分析

1、原始图示

程序的原始图示如下，由图示可知，程序会判断当前是否有空闲块，当前块是否命中，当前是否需要置换，置换的是哪一块，以及显示当前各块存放的数值情况。



```
cache有空闲块,不考虑是否要置换...
3被调入cache....
cache块0: 1
cache块1: 2
cache块2: 4
cache块3: 3

cache已经满了,考虑是否置换...

1被5在cache的0号块置换...
cache块0: 5
cache块1: 2
cache块2: 4
cache块3: 3

2命中!!!
cache块0: 5
cache块1: 2
cache块2: 4
cache块3: 3
```

```
4被1在cache的2号块置换...
cache块0: 5
cache块1: 2
cache块2: 1
cache块3: 3
cache已经满了,考虑是否置换...

3被6在cache的3号块置换...
cache块0: 5
cache块1: 2
cache块2: 1
cache块3: 6
cache已经满了,考虑是否置换...

5被7在cache的0号块置换...
cache块0: 7
cache块1: 2
cache块2: 1
cache块3: 6

1命中!!!
cache块0: 7
cache块1: 2
cache块2: 1
cache块3: 6
```

2、测试数据输入及输出

Cache 块数为 4

测试页面数为 12

工作集为：{1,1,2,4,3,5,2,1,6,7,1,3}

具体的装入、置换、命中表整理如下，

表 2-2

	1	1	2	4	3	5	2	1	6	7	1	3
Cache 块 0	1	1	1	1	1	5	5	5	5	7	7	7
Cache 块 1			2	2	2	2	2	2	2	2	2	3
Cache 块 2				4	4	4	4	1	1	1	1	1
Cache 块 3					3	3	3	3	6	6	6	6
	装入	命中	装入	装入	装入	置换	命中	置换	置换	置换	命中	置换

Cache 块数为 4

测试页面数为 10

工作集为：{1,2,2,3,7,1,6,1,6,7}

具体的装入、置换、命中表整理如下，

表 2-3

	1	2	2	3	7	1	6	1	6	7
Cache 块 0	1	1	1	1	1	1	1	1	1	1
Cache 块 1		2	2	2	2	2	6	6	6	6
Cache 块 2				3	3	3	3	3	3	3
Cache 块 3					7	7	7	7	7	7
	装入	装入	命中	装入	装入	命中	置换	命中	命中	命中

3、正确性分析

通过对源代码检验，输入不同范围的测试数据并观察其运行结果可以看出该程序具有如下特点：

- 正确性
- 鲁棒性

五、调试报告

1、调试中的问题

调试中发现当 cache 未装满时，会出现一样的数值在两个 cache 中，考虑这个问题是由于命中与否分支数学错误。

在初次调试时发现，第一次装满后置换时总是置换放在 Cache 里的值，这与预先按照 LRU 设想的置换位置不一致。

总体来说当掌握了 LRU 后，代码的书写困难不大，所以调试过程中遇到的问题大都是粗心导致，问题也较少。

2、问题解决

在解决第一个问题时我修改了判断分支的嵌套关系，将最外一层变为对该块是否空闲以及内部值是否等于当前装入数值，具体如下：

```
if((cache[j].state == false) && (walk_sort[i] != cache[j].value))
```

内层改为了对当数值与已有数值相等，具体如下：

```
if (cache[j].value == walk_sort[i]) {
```

在解决第二个问题时，我检查了代码，将问题出现的位置定位到了“若有 cache 空闲则直接装入”处，发现当空闲时并没有在装入后更新其他位置的 T 值，这才导致每次遍历时 cache 中的 T 都为 0 这样每次置换都是置换 cache0。

六、经验与体会

在用 LRU 算法更新 Cache 的过程中，我更加深刻的认识到了 LRU 算法的优点，目前我能通过实验了解到的是当存在热点数据时，LRU 算法的命中率有很强的优势。但是，当 Cache 块数较小时，偶发性的、周期性的批量操作会导致 LRU 命中率急剧下降，缓存污染情况比较严重。

在观摩过别的同学对于同一个程序的理解后，我也意识到本程序的编写 其实是很有多样性的，不该被局限在自己的认知当中。

总体来说 LRU 算法更新 cache 的代码逻辑较为简单，主要是对 LRU 算法的理解，在此实验之外我也会好好了解其他置换算法总结出各个算法的优缺点。

实验 3 单功能流水线调度机构模拟

一、问题描述

通过模拟单功能流水线调度过程，掌握流水线技术，学会计算流水线的吞吐率、加速比、效率。

1、流水线的表示法有三种：

连接图、时空图、预约表。对于线性流水线，主要考虑前二种。

2、流水线的主要特点：

在流水线的每一个功能部件的后面都要有一个缓冲器，称为锁存器、闸门寄存器等，它的作用是保存本流水段的执行结果。各流水段的时间应尽量相等，否则回引起阻塞、断流等。只有连续提供同类任务才能充分发挥流水线的效率。在流水线的每一个流水线段中都要设置一个流水锁存器。流水线需要有“装入时间”和“排空时间”。只有流水线完全充满时，整个流水线的效率才能得到充分发挥。

在本实验中，流水线各部件较为简单，各功能段周期相等且不存在瓶颈段。

二、设计简要描述

由于这是一个实现浮点加的单流水线，每个功能部件周期相同，且没有瓶颈段存在，所以设计较为简单。可以得出各时间段各部件是否工作，正在工作的是第几个指令即可。

常量设置：功能部件数目为 4、需要流水处理的浮点加指令数目为 5，

（1）函数说明

print 函数：输出时空图函数。

pipeline 函数：实现流水线中指令状态转换算法。

main 函数：程序的入口。

三、程序清单

```
#include <iostream>
#include <string>

using namespace std;

const int SPACE = 4;//  功能部件数目
const int NUM = 5;//  需要流水处理的浮点加指令数目
const int TIME = NUM + SPACE - 1;//  存储不同时间段各个功能部件内指令值
```

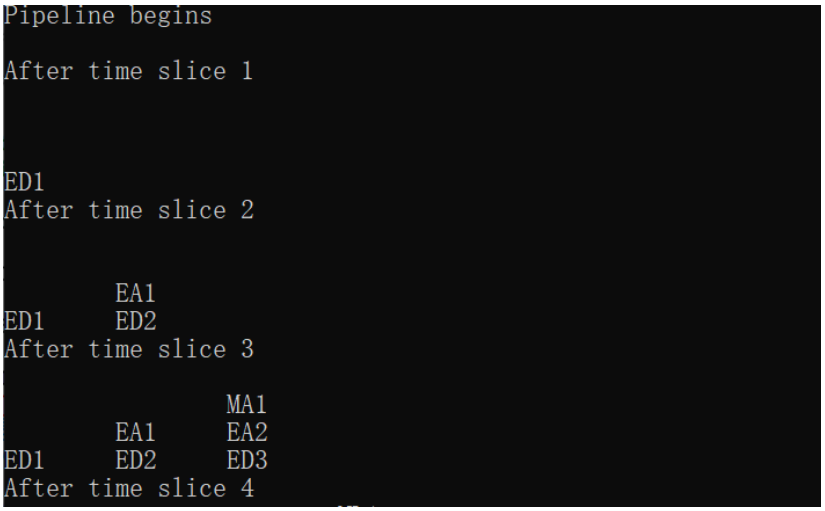
```
// ED：求阶差 EA：对阶 MA：尾数加 NL：规格化
const string INSTRUCTIONS[] = { "NL", "MA", "EA", "ED" };
int ts[SPACE][TIME] = { 0 };// 初始化时空图[4][8]
//ts 里保存的是当前位置属于第几个浮点加指令
int time = 1;/*记录运行时候时间周期*/

void print();// 输出时空图
void pipeline(int ts[SPACE][TIME]);// 流水线中指令状态转换算法

int main() {
	cout << "Pipeline begins" << endl << endl;
	pipeline(ts);
	print();

	cout << endl << "Pipeline ends" << endl << endl;
	cout << "The Through Put of the pipeline is " << (double)NUM / TIME << "t" << endl;
	cout << "The Speedup of the pipeline is " << ((double)NUM*SPACE) / TIME << endl;
	cout << "The Efficiency of the pipeline is " << ((double)NUM*SPACE) / (TIME*SPACE) <<
endl;

	return 0;
}

//打印函数
void print() {
	for (int i = 0; i < TIME; ++i) {	//NUM+SPACE-1
		cout << "After time slice " << i + 1 << endl;
		for (int j = 0; j < SPACE; ++j) {	//部件
			if (i < NUM && ts[j][i] == 0) {	//指令数目
				cout << endl;	//换行
			}
			else {
				for (int k = 0; k < i + 1; ++k) {
					if (ts[j][k] != 0) {

						cout << INSTRUCTIONS[j] << ts[j][k];
					}
					cout << "\t";
				}
				cout << endl;
			}
		}
	}
}
```

```
void pipeline(int ts[SPACE][TIME]) {
    int tempSpace = 0;//  记录处理的指令号
    int tempTime = 0;//  记录时间轴的变化
    for (int s = SPACE - 1; s >= 0; s--) {// 3\2\1\0
        tempSpace = 1;
        for (int t = tempTime; t < TIME; t++) {// 1\...\8
            ts[s][t] = tempSpace++;

            if (s + t >= SPACE + NUM - 1) {
                ts[s][t] = 0;
            }
        }
        tempTime++;
    }
}
```

四、结果分析

1、原始图示

该程序的原始图示如下图，



由图示可以看出，该程序会将时间按周期切片为等距片段，并在每个时间段显示当前流水段的内容。

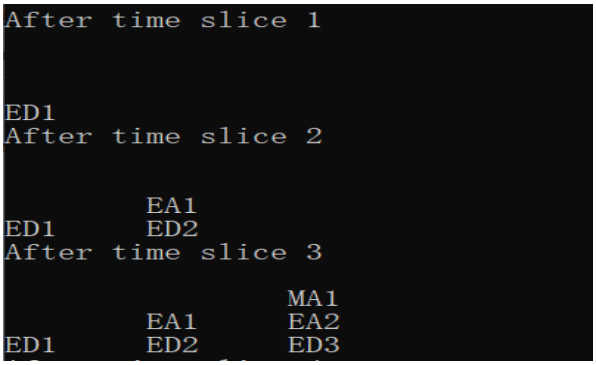
2、测试数据与运行记录

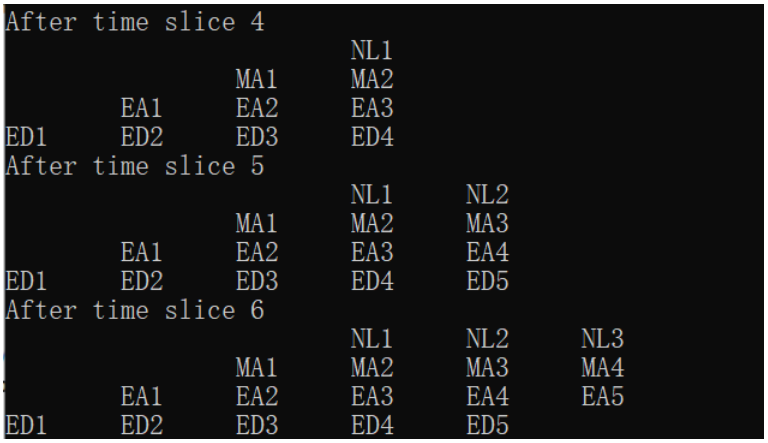
此程序已设置好了部件数目，需流水处理的浮点加指令数目。

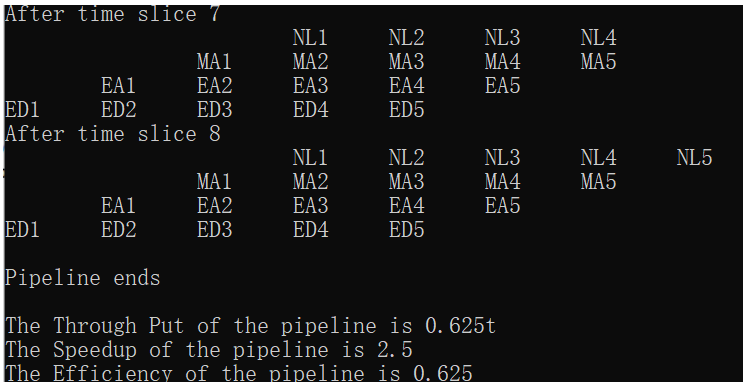
功能部件数目：4

需要流水处理的浮点加指令数目：5

运行时间周期：1







3、正确性分析

通过对源代码检验，输入不同范围的测试数据并观察其运行结果可以看出，当输入的指令数为 1 时，程序能够识别出并要求重新输入；当输入的指令的概率之和大于 1 时，程序会识别出并指出错误。可见该程序具有如下特点：

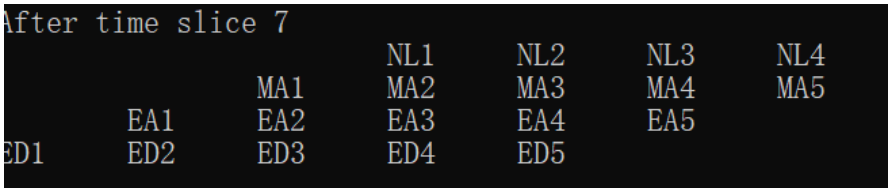
- 正确性
- 鲁棒性

五、调试报告

1、调试中的问题

调试中打印函数最后结果少输出了最后一列，导致单流水线整体最后出错，

怀疑在初始设置常量时出错。错误图如下，



2、问题的解决

在对上述问题进行探究后我发现是在 ts 数组设置的时候出现问题。在程序设计的过程中为了使每一个时间片段都显示对应的流水线序号，所以给每一个设置了 ts[s][t]来记录当前的片段的流水线序号，当该数组元素为 0 时对应输出为一个空格。

```
if (s + t >= SPACE + NUM - 1)
```

例如 s=3 时，t 属于 0 到 4 时该分支都不需要跳转，所以右侧为 8，这样从最底下一行的第六个空格开始对应的数组值为 0，在打印时也就不会打印出流水线片段。刚开始时我将-1 写为了-2，这才导致了错误的发生。

六、经验与体会

通过制作该但功能流水线调度模拟程序，我更加理解了单功能流水线调度的基本原理。也巩固了有关吞吐率、加速比、效率的理解。

但由于本实验制作的是较为单一的但功能流水线，每个功能部件都占有一个 slice，且不存在瓶颈段。所以在实现的时候只需要记录每个段对应功能部件所处理的是第几个流水线，然后再根据流水线调度规律总结出代码。

在观摩过别的同学对于同一个程序的理解后，我也意识到本程序的编写 其实是很有多样性的，不该被局限在自己的认知当中。

做完本实验后我还尝试了写一写存在瓶颈段时的时空图的绘制。我发现加入瓶颈段后各代码段的复杂度都有提高，并且经过计算流水线的各性能指标也都有所下降。实践证明瓶颈段对于流水线的整体效率影响是很大的，因此我们在计算机体系结构这门课中学到的各种调度方法也都有了用武之地。