# 作业三

## T1

> 编号为1、2、3、4、5的五辆列车，顺序开进一个栈式结构的站点，问开出车站的顺序有多少种可能？请具体写出所有可能的出栈序列。

首先根据卡特兰公式可以直接计算出总方案数：

$$\frac{1}{n+1}C_{2n}^{n}=\frac{1}{5+1}C_{10}^{5}=42$$

接下来我们编程求解所有具体的方案。对于一辆车，只有两种状态：

- 在车站内，此时可以出站，也可以不出站
- 在车站外，此时可以进站，也可以不进站

我们利用深度优先搜索解决这个问题，代码如下：

```
void testCatlan() {
    SeqStack<int, 100> station;
    vector<int> out;
    int n = 5;
    get_train_seq(station, out, n);
}

void get_train_seq(SeqStack<int, 100>& station, vector<int>& out, int n) {
    /**
     * @note get all the train sequence
     * @param station train station
     * @param out the train sequence
     * @param n the number of the train
     */

    if (out.size() == 5) {
        // all the train has been out
        for (int i = 0; i < out.size(); i++) {
            cout << out[i] << " \n"[i == out.size() - 1];
        }
        return;
    }
```

```
23
24      if (n > 0) {
25          // condition 1: there are trains not in the station, push the train
    into the station
26          station.Push(n);
27          get_train_seq(station, out, n - 1);
28          station.Pop();
29      }
30
31      if (!station.Empty()) {
32          // condition 2: there are trains in the station, pop the train out
33          int train = station.Top();
34          station.Pop();
35          out.push_back(train);
36          get_train_seq(station, out, n);
37          out.pop_back();
38          station.Push(train);
39      }
40  }
```

最终可以得到下面的运行结果，刚好 42 种方案：

```
 1  1 2 3 4 5
 2  2 1 3 4 5
 3  2 3 1 4 5
 4  2 3 4 1 5
 5  2 3 4 5 1
 6  3 1 2 4 5
 7  3 2 1 4 5
 8  3 2 4 1 5
 9  3 2 4 5 1
10  3 4 1 2 5
11  3 4 2 1 5
12  3 4 2 5 1
13  3 4 5 1 2
14  3 4 5 2 1
15  4 1 2 3 5
16  4 2 1 3 5
17  4 2 3 1 5
18  4 2 3 5 1
19  4 3 1 2 5
20  4 3 2 1 5
```

```
4 3 2 5 1
4 3 5 1 2
4 3 5 2 1
4 5 1 2 3
4 5 2 1 3
4 5 2 3 1
4 5 3 1 2
4 5 3 2 1
5 1 2 3 4
5 2 1 3 4
5 2 3 1 4
5 2 3 4 1
5 3 1 2 4
5 3 2 1 4
5 3 2 4 1
5 3 4 1 2
5 3 4 2 1
5 4 1 2 3
5 4 2 1 3
5 4 2 3 1
5 4 3 1 2
5 4 3 2 1
```

## T2

> 利用栈实现把十进制整数转换为二至十六之间的任一进制数输出的功能。

取余再相除即可。

```cpp
void testConvert() {
    int num, base;
    cin >> num >> base;

    SeqStack<int, 100> a;
    while (num >= base) {
        int mod = num % base;
        a.Push(mod);
        num = (num - mod) / base;
    }
    a.Push(num);

```

```
    while (!a.Empty()) {
        int t = a.Top();
        a.Pop();
        if (t < 10) {
            cout << t;
        } else {
            cout << (char)('A' + t - 10);
        }
    }
}
```

## T3

> 假设表达式中允许包含3种括号：圆括号、方括号和大括号。试编写一个算法，检查表达式中括号是否配对，若能够全部配对则返回1，否则返回0。

扫描表达式即可。匹配逻辑为：遇到数字或运算符则直接略过，遇到左括号则入栈，遇到右括号则进行匹配的判断。最后需要检查括号栈中是否为空，防止有多余的左括号。时间复杂度 $O(n)$

```
bool testMatch(std::string s = "{10+9*12+[10/(2+90)+1]*(1+2)}") {
    SeqStack<char, 100> a;
    string op = "{[()]}";
    int i = 0;

    while (i < s.size()) {
        if (isdigit(s[i]) || op.find(s[i]) == string::npos) {
            // skip numbers or operators
            i++;
        } else if (s[i] == '{' || s[i] == '[' || s[i] == '(') {
            // left bracket
            a.Push(s[i++]);
        } else {
            // right bracket
            if (a.Empty()) {
                // too many right brackets
                return false;
            } else if (s[i] == '}' && a.Top() == '{' ||
                       s[i] == ']' && a.Top() == '[' ||
                       s[i] == ')' && a.Top() == '(') {
                a.Pop();
                i++;
```

```
            } else {
                // mismatch
                return false;
            }
        }
    }

    return a.Empty();
}
```

## T4

> 设有一维数组stack[StackMaxSize]，分配给两个栈S1和S2使用，如何分配数组空间，使得对任何一个栈，当且仅当数组空间全满时才不能插入。试说明你的分配方法。并分别给出两个栈各自的入栈和出栈算法。

借鉴双端队列 `deque` 的思路，我们使用双端栈。即数组**左端**实现一个栈的压入与弹出，数组**右端**实现另一个栈的压入与弹出。通过**双指针**进行内存空间的动态调度。设计一个双端栈来实现双栈的压入、弹出算法。

类声明

```
template<class T, int MaxSize>
class SeqDoubleStack {
private:
    T data[MaxSize];
    int l, r;

public:
    SeqDoubleStack() : l(-1), r(MaxSize) {}
    void Push_front(T x);
    T Pop_front();
    void Push_back(T x);
    T Pop_back();
    void Output();
};
```

左入栈

```cpp
template<class T, int MaxSize>
void SeqDoubleStack<T, MaxSize>::Push_front(T x) {
    if (l + 1 == r) {
        cerr << "full size" << endl;
        exit(1);
    }
    data[++l] = x;
}
```

左出栈

```cpp
template<class T, int MaxSize>
T SeqDoubleStack<T, MaxSize>::Pop_front() {
    if (l == -1) {
        cerr << "empty left stack" << endl;
        exit(1);
    }
    T now = data[l--];
    return now;
}
```

右入栈

```cpp
template<class T, int MaxSize>
void SeqDoubleStack<T, MaxSize>::Push_back(T x) {
    if (l + 1 == r) {
        cerr << "full size" << endl;
        exit(1);
    }
    data[--r] = x;
}
```

右出栈

```cpp
template<class T, int MaxSize>
T SeqDoubleStack<T, MaxSize>::Pop_back() {
    if (r == MaxSize) {
        cerr << "empty right stack" << endl;
        exit(1);
    }
    T now = data[r++];
    return now;
}
```

## T5

> 现有中缀表达式 $E = ((A - B)/C + D * (E - F)) * G$ （注：此题在纸上练习，不用提交）
>
> 1. 写出与 E 等价的后缀表达式。
> 2. 用一个操作符栈来模拟表达式的转换过程，画出在将 E 转换成后缀表达式的过程中，栈内容的变化图。
> 3. 用一个操作数栈来模拟后缀表达式的求值过程，画出对（2）中所得到的后缀表达式求值时，栈中内容的变化图。

1. 根据中缀表达式画出一棵表达式树之后，后序遍历一遍就是后缀表达式。当然用栈其实就是存储「模拟递归」过程中的操作数和操作符。
2. **中缀表达式转后缀表达式**。这需要我们了解中缀表达式和后缀表达式的构造和解析的逻辑。在后缀表达式的解析过程中，我们一旦遇到操作符就需要进行运算，这恰恰对应了中缀表达式的构造过程中，遇到右括号或比较操作符优先级时进行运算的过程。因此我们可以利用构造中缀表达式的过程构造出后缀表达式。即：当中缀遇到操作数时构造后缀的操作数，当中缀进行运算时构造后缀的操作符即可。
3. **后缀表达式求值**。此时只需要一个操作数栈即可，因为后缀的顺序为「左操作数+右操作数+操作符」，并不需要额外的容器存储操作符，遇到操作数直接入栈，遇到操作符直接取两个操作数进行运算，再将运算结果压入操作数栈即可。

中缀表达式求值的具体解析可见这篇博客：AcWing 3302. 表达式求值 | 原理解析 - AcWing

## T6

> 假设以带头结点的循环链表表示队列，并且只设一个表尾指针，试编写相应的置队列空、入队和出队操作。

循环链表类在第二章已经实现，此处给出完整类代码：

```cpp
template<class T>
class CircleList {
private:
    Node<T>* tail;

public:
    CircleList();
    ~CircleList();

    void PushBack(T x);  // push element to end
    void PopFront();     // pop front element
    void Clear();        // clear all elements
    int CountNode();     // count node number
};
```

置空

```cpp
template<class T>
void CircleList<T>::Clear() {
    while (tail->next != tail) {
        PopFront();
    }
}
```

入队

```cpp
template<class T>
void CircleList<T>::PushBack(T x) {
    Node<T>* now = new Node<T>(x);
    Node<T>* temp = tail->next;
    tail->next = now;
    tail = now;
    tail->next = temp;
}
```

出队

```cpp
template<class T>
void CircleList<T>::PopFront() {
    if (tail->next == tail) {
        std::cerr << "empty circle list!" << "\n";
        exit(1);
```

```
    }
    if (tail->next->next == tail) {
        // one node
        Node<T>* temp = tail;
        tail = tail->next;
        tail->next = tail;
        delete temp;
    } else {
        // at least two nodes
        Node<T>* temp = tail->next->next;
        tail->next->next = temp->next;
        delete temp;
    }
}
```

## T7

> 假设以一维数组data[m]存储循环队列的元素，若要使这m个分量都得到应用，则另设一辅助标志变量flag判断队列的状态为“空”还是“满”。编写入队和出队算法。

首先，循环顺序队列就是在常规队列的基础之上利用静态数组空间进行元素的存储，同时对下标索引进行取模操作，这样就可以避免「假上溢」的情况。但是这种存储方式会导致无法区分队空与队满。书中示例部分给出了第一种区分策略，即浪费一个数组空间用来进行边界判定。T7 和 T8 分别给出了另外两个区分策略。

对于「判定循环顺序队列队空/满」第二种策略：显然的无论队空还是队满，头尾指针都有 `(tail + 1) % MaxSize == head` 的关系，因此我们才需要引入额外的一个标记变量 `flag` 进行区分。如何进行区分呢？显然的我们需要在入队时检查是否队满，需要在出队时检查是否队空。

- 当入队检查是否队满时：当且仅当 `(tail + 1) % MaxSize == head` 且曾经有元素入队时，才为队满状态；
- 当出队检查是否队空时：当且仅当 `(tail + 1) % MaxSize == head` 且所有元素均出队时，才为队空状态。

因此可以用「当前队列中是否有元素」这个二元状态来唯一区分 `(tail + 1) % MaxSize == head` 的队空/满两种情况。因此 `flag` 变量可以更准确地重命名为 `is_empty`。即当 `flag` 为真时，表示队列为空；当 `flag` 为假时，表示队列非空。

我们定义「使用辅助变量的循环顺序队列」如下：

```
template<class T, int MaxSize>
class CircleSeqQueueWithFlag {
```

```
private:
    T data[MaxSize];
    int head, tail;
    bool is_empty; // flag var

public:
    CircleSeqQueueWithFlag() : head(0), tail(-1), is_empty(true) {}
    void Push(T x);
    void Pop();
    T Front();
    bool Empty() { return is_empty; }
};
```

入队

```
template<class T, int MaxSize>
void CircleSeqQueueWithFlag<T, MaxSize>::Push(T x) {
    if ((tail + 1) % MaxSize == head && !is_empty) {
        cerr << "full queue!\n";
        exit(1);
    }
    data[(tail + 1) % MaxSize] = x;
    tail = (tail + 1) % MaxSize;
    is_empty = false;
}
```

出队

```
template<class T, int MaxSize>
void CircleSeqQueueWithFlag<T, MaxSize>::Pop() {
    if (is_empty) {
        cerr << "empty queue\n";
        exit(1);
    }
    head = (head + 1) % MaxSize;
    if ((tail + 1) % MaxSize == head) {
        is_empty = true;
    }
}
```

## T8

> 假设以一维数组data[m]存放循环队列的元素，同时设变量num表示当前队列中元素的个数，以判断队列的状态为“空”还是“满”。试给出此循环队列满的条件，并编写入队和出队算法。

本题即「判定循环顺序队列队空/满」的第三种策略：通过记录队列元素个数进行判定。这也是最直接的理解方式。那么显然的当计数器 `num == MaxSize` 时表示队满。

我们定义「使用计数变量的循环顺序队列」如下：

```cpp
template<class T, int MaxSize>
class CircleSeqQueueWithNum {
private:
    T data[MaxSize];
    int head, tail;
    int num;

public:
    CircleSeqQueueWithNum() : head(0), tail(-1), num(0) {}
    void Push(T x);
    void Pop();
    T Front();
    bool Empty() { return !num; }
};
```

入队

```cpp
template<class T, int MaxSize>
void CircleSeqQueueWithNum<T, MaxSize>::Push(T x) {
    if (num == MaxSize - 1) {
        cerr << "full queue\n";
        exit(1);
    }
    data[tail] = x;
    tail = (tail + 1) % MaxSize;
    num++;
}
```

出队

```cpp
template<class T, int MaxSize>
void CircleSeqQueueWithNum<T, MaxSize>::Pop() {
    if (!num) {
        cerr << "empty queue\n";
        exit(1);
    }
    head = (head + 1) % MaxSize;
    num--;
}
```

## T9

> 如何用两个栈来实现队列？并写出队列基本操作的算法。

由于队列是先进先出的数据结构，而栈是先进后出的数据结构，对于先入栈的元素，想要先出栈就只能作为栈顶。为了让先入栈的元素成为栈顶，可以借助第二个栈来转置。出栈结束后再返回到第一个栈即可进入待入队状态。时间复杂度：$O(n^2)$

声明列表

```cpp
template<class T, int MaxSize>
class DoubleStack4Queue {
private:
    SeqStack<T, MaxSize> A; // top for queue push
    SeqStack<T, MaxSize> B; // top for queue pop

public:
    DoubleStack4Queue() {}
    void Push(T x);
    void Pop();
    T Front();
    bool Empty() { return A.Empty() && B.Empty(); }
};
```

入队

```
template<class T, int MaxSize>
void DoubleStack4Queue<T, MaxSize>::Push(T x) {
    // Stack already has overflow check
    while (!B.Empty()) {
        A.Push(B.Top());
        B.Pop();
    }
    A.Push(x);
}
```

出队

```
template<class T, int MaxSize>
void DoubleStack4Queue<T, MaxSize>::Pop() {
    while (!A.Empty()) {
        B.Push(A.Top());
        A.Pop();
    }
    if (B.Empty()) {
        cerr << "empty queue\n";
        exit(1);
    }
    B.Pop();
}
```

取队头

```
template<class T, int MaxSize>
T DoubleStack4Queue<T, MaxSize>::Front() {
    while (!A.Empty()) {
        B.Push(A.Top());
        A.Pop();
    }
    if (B.Empty()) {
        cerr << "empty queue\n";
        exit(1);
    }
    return B.Top();
}
```

判空

```cpp
template<class T, int MaxSize>
bool DoubleStack4Queue<T, MaxSize>::Empty() {
    return A.Empty() && B.Empty();
}
```

# 实验三

实验代码：https://github.com/Explorer-Dong/DataStructure/blob/main/Code/chapter3/Experiment_3.cpp

## T1

> 顺序栈的实现与应用。
>
> 1. 编写main()函数对class SeqStack进行测试，要求：使用菜单选择各项功能。
> 2. 利用顺序栈采用算符优先算法编程实现直接计算中缀表达式的值，要求：输入中缀算术表达式，计算表达式的值。
> 3. 利用顺序栈编程实现先将中缀表达式转换成后缀表达式，再计算后缀表达式的值。
>
> 扩展要求：请修改第（2）题中的中缀表达式求值算法，不仅可以对1位数的操作数进行算术运算，还可以对如下所示的更大的整数做计算：123-89*25-960

1. 对于第一题。使用 `while+switch` 语句即可测试 SeqStack 的全部功能。
2. 对于第二题。经典的中缀表达式求值问题，已在作业 T5 中进行了详细的原理阐述，这里直接给出能够兼容不止一个数位运算的代码。
3. 对于第三题。经典的后缀表达式求值问题，同样也已在作业 T5 中进行了详细的原理阐述，这里直接给出对应的代码。

中缀表达式求值：

```cpp
void testCalcMid(const std::string s = "2*(1+3+1)") {
    unordered_map<char, int> pri{{'(', 0}, {'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}};

    SeqStack<int, 100> num;
    SeqStack<char, 100> op;

    auto calc = [&]() -> void {
        int b = num.Top(); num.Pop();
        int a = num.Top(); num.Pop();
```

```
        char o = op.Top(); op.Pop();

        if (o == '+') num.Push(a + b);
        else if (o == '-') num.Push(a - b);
        else if (o == '/') num.Push(a / b);
        else num.Push(a * b);
    };

    for (int i = 0; i < s.size(); i++) {
        if (isdigit(s[i])) {
            int x = 0, j = i;
            while (j < s.size() && isdigit(s[j])) {
                x = x * 10 + s[j++] - '0';
            }
            num.Push(x);
            i = j - 1;
        } else if (s[i] == '(') {
            op.Push(s[i]);
        } else if (s[i] == ')') {
            while (op.Top() != '(') {
                calc();
            }
            op.Pop();
        } else {
            while (!op.Empty() && pri[op.Top()] >= pri[s[i]]) {
                calc();
            }
            op.Push(s[i]);
        }
    }

    while (!op.Empty()) {
        calc();
    }

    cout << "result: " << num.Top() << "\n";
}
```

中缀表达式转后缀表达式：

```
std::string Mid2Post(const std::string& s) {
    string post;
```

```
    unordered_map<char, int> pri{{'(', 0}, {'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}};
    SeqStack<char, 100> op;

    for (int i = 0; i < s.size(); i++) {
        if (isdigit(s[i])) {
            int x = 0, j = i;
            while (j < s.size() && isdigit(s[j])) {
                x = x * 10 + s[j++] - '0';
            }
            i = j - 1;
            post += to_string(x) + " ";
        } else if (s[i] == '(') {
            op.Push(s[i]);
        } else if (s[i] == ')') {
            while (op.Top() != '(') {
                post += string(1, op.Top()) + " ";
                op.Pop();
            }
            op.Pop();
        } else {
            while (!op.Empty() && pri[s[i]] <= pri[op.Top()]) {
                post += string(1, op.Top()) + " ";
                op.Pop();
            }
            op.Push(s[i]);
        }
    }

    while (!op.Empty()) {
        post += string(1, op.Top()) + " ";
        op.Pop();
    }

    return post;
}
```

后缀表达式求值：

```
int CalcPost(const std::string& post) {
    SeqStack<int, 100> num;
```

```
    for (int i = 0; i < post.size(); i++) {
        if (isdigit(post[i])) {
            int x = 0, j = i;
            while (j < post.size() && isdigit(post[j])) {
                x = x * 10 + post[j++] - '0';
            }
            num.Push(x);
            i = j - 1;
        } else if (post[i] == ' ') {
            continue;
        } else {
            int b = num.Top(); num.Pop();
            int a = num.Top(); num.Pop();
            if (post[i] == '+') num.Push(a + b);
            else if (post[i] == '-') num.Push(a - b);
            else if (post[i] == '/') num.Push(a / b);
            else num.Push(a * b);
        }
    }

    return num.Top();
}
```

## T2

> 队列的实现与应用。
>
> 1. 编写main()函数对class SeqQueue进行测试，要求：使用菜单选择各项功能。
> 2. 编写一个程序，模拟患者在医院等待就诊的情况，主要模拟两件事：
>    1. 患者到达诊室，将病历交给护士，排到等待队列中候诊；
>    2. 护士从等待队列中取出下一位患者的病历，该患者进入诊室就诊。
>
> 程序采用菜单方式，其选项及功能说明如下：
>
> 1. 排队：输入排队患者的病历号（随机产生），加入到就诊患者排队队列中；
> 2. 就诊：患者队列中最前面的病人就诊，并将其从队列中删除；
> 3. 查看：从队首到队尾列出所有排队患者的病历号；
> 4. 下班：退出运行。

1. 对于第一题。使用 `while+switch` 语句即可测试 SeqQueue 的全部功能。

2. 对于第二题。使用任意支持「入队、出队、查看队头」的队列模拟即可，这里使用上文提到的循环队列进行模拟。

## T3

> 对一给定的迷宫，求其从入口到出口的最短路径。

迷宫约定：仅限于二维矩阵，0 表示可达，1 表示不可达，且可达点之间的转移路径均长度均为 1，可转移的方向只有上下左右四个方向。

原理解析：首先最短路径长度的求解比较容易想到，使用 bfs 维护一个距离矩阵 dist 即可。那么如何求解最短路呢？我们可以利用前面求出的 dist 距离矩阵逆推最短路。从终点出发，每次将旧点 `(i,j)` 迭代到四个方向上任一合法的新点 `(ni,nj)` 即可。这里的「合法」体现在：不是障碍点、新点到起点的距离比旧点到起点的距离小 1。

```
pair<vector<pair<int, int>>, int>
getShortestPath(vector<vector<int>>& g, pair<int, int>& st, pair<int, int>& en)
{
    int m = g.size(), n = g[0].size();
    vector<vector<int>> d(m, vector<int>(n, INT_MAX >> 1));
    int sti = st.first, stj = st.second;
    int eni = en.first, enj = en.second;

    int dx[] = {-1, 1, 0, 0};
    int dy[] = {0, 0, -1, 1};
    bool find_path = false;

    // update dist matrix with bfs algorithm
    CircleSeqQueueWithFlag<pair<int, int>, 100> q;
    q.Push({sti, stj});
    d[sti][stj] = 0;
    while (!q.Empty()) {
        auto now = q.Front();
        q.Pop();

        int i = now.first, j = now.second;
        if (i == eni && j == enj) {
            find_path = true;
            break;
        }
        for (int k = 0; k < 4; k++) {
```

```
            int ni = i + dx[k], nj = j + dy[k];
            if (ni < 0 || ni >= m || nj < 0 || nj >= n || g[ni][nj] || d[ni][nj] != INT_MAX >> 1) {
                continue;
            }
            d[ni][nj] = d[i][j] + 1;
            q.Push({ni, nj});
        }
    }

    // edge case
    if (!find_path) {
        cerr << "no valid path!\n";
        exit(1);
    }

    // get path from end point
    vector<pair<int, int>> path;
    int i = eni, j = enj;
    while (i != sti || j != stj) {
        path.push_back({i, j});
        for (int k = 0; k < 4; k++) {
            int ni = i + dx[k], nj = j + dy[k];
            if (ni < 0 || ni >= m || nj < 0 || nj >= n || g[ni][nj] || d[ni][nj] != d[i][j] - 1) {
                continue;
            }
            i = ni, j = nj;
            break;
        }
    }
    path.push_back({sti, stj});
    reverse(path.begin(), path.end());

    return make_pair(path, d[eni][enj]);
}
```