数据结构作业（4）

1. 假设CQ[1...10]是一个循环队列，初始状态为front=rear=1，画出做完下列运算后的头尾指针的状态变化情况，若不能入队，请指出元素，并说明理由。

- d, e, b, g, h入队

> 1:(front) -> 2:d -> 3:e -> 4:b -> 5:g -> 6:h(rear), 此时front = 1, rear = 6

- d, e出队

> 3:(front) -> 4:b -> 5:g -> 6:h(rear), 此时front = 3, rear = 6

- i, j, k, l, m入队

> 3:(front) -> 4:b -> 5:g -> 6:h -> 7:i -> 8:j -> 9:k -> 10:l -> 1:m(rear), 此时front = 3, rear = 1

- b出队

> 4:(front) -> 5:g -> 6:h -> 7:i -> 8:j -> 9:k -> 10:l -> 1:m(rear), 此时front = 4, rear = 1

- n, o, p, q, r入队

> 4:(front) -> 5:g -> 6:h -> 7:i -> 8:j -> 9:k -> 10:l -> 1:m -> 2:n -> 3:o(rear) 此时front = 4, rear = 3

p, q, r将无法入队，此时 (rear+1)%10 = 4 = front，队列已满

2. 设循环队列的容量为40（序号从0到39），现经过一些列的入队和出队运算后，有：

- front=11, rear=19;
- front=19, rear=11

请问在这两种情况下，循环队列中各有元素多少?

- front=11, rear=19

在这种情况下循环队列中共有 19 - 11 = 8 个元素

- front=19, rear=11

在这种情况下循环队列中共有 40 - 19 + 11 = 32 个元素

3. 若以1234作为双端队列的输入序列，分别求出满足下列条件的输出序列：

(1) 能由输入受限的双端队列得到，但是不能由输出受限的双端队列得到的输出序列；

(2) 能由输出受限的双端队列得到，但是不能由输入受限的双端队列得到的输出序列；

(3) 既不能由输入受限的双端队列得到，也不能由输出受限的双端队列得到的输出序列；

模拟程序如下，其中 `checkQueue1` 对应输出受限的队列， `checkQueue2` 对应输入受限的队列

```
vector<vector<int>> fullPermulation(vector<int> && nums) {
    if (nums.empty()) return {};
    if (nums.size() == 1) return {nums};

    vector<vector<int>> res;
    for (unsigned i = 0; i < nums.size(); i++) {
        swap(nums[0], nums[i]);

        vector<vector<int>> sub_res =
            fullPermulation(vector<int>(nums.begin() + 1, nums.end()));

        for (auto &v : sub_res) {
            v.insert(v.begin(), nums[0]);
            res.push_back(v);
        }

        swap(nums[0], nums[i]);
    }
    return res;
}

void checkQueue1() {
    // if the queue can enqueue elements at head and tail, but dequeue elements only at head
    // check which of the sequences in combination() can be generated by the queue
    vector<int> nums = {1, 2, 3, 4};
    vector<vector<int>> res = fullPermulation(move(nums)); // the full permutation of nums

    for (auto &seq : res) {
        vector<int> queue_simu;
        unsigned i = 0, j = 0;
        while (true) {
            if (queue_simu.empty()) { // if the queue is empty, enqueue nums[j] directly
                queue_simu.push_back(nums[j]);
            } else {
                // think about the way to enqueue nums[j] to the queue:
                // 2 ways: enqueue at head or enqueue at tail
                // you just need to care about the relative position of nums[j] and
                // any element in the queue (1 ele is enough, cause you just need to
                // care about the ways to enqueue nums[j]. If you want to think all of them,
                // you can break the loop as soon as you find the conflict)
                int ele = queue_simu[0];
                int pos_ele = -1, pos_j = -1; // find the pos of tmp and nums[j] in seq
                for (unsigned k = 0; k < seq.size(); k++) {
                    if (seq[k] == ele) pos_ele = k;
                    if (seq[k] == nums[j]) pos_j = k;
                }
                if (pos_ele > pos_j) {
                    queue_simu.insert(queue_simu.begin(), nums[j]);
                } else {
                    queue_simu.push_back(nums[j]);
```

```
            }
        }

        while (!queue_simu.empty()) {
            if (queue_simu[0] == seq[i]) {
                queue_simu.erase(queue_simu.begin());
                i += 1;
            } else {
                break;
            }
        }

        j += 1;
        if (i == seq.size() || j == nums.size()) break;
    }

    if (i == seq.size()) {
        cout << "seq: ";
        for (auto &i : seq) cout << i << " ";
        cout << "can be generated by the queue" << endl;
    } else {
        cout << "seq: ";
        for (auto &i : seq) cout << i << " ";
        cout << "can not be generated by the queue" << endl;
    }
  }
}


void checkQueue2() {
    // if the queue can dequeue elements at head and tail, but enqueue elements only at tail
    // check which of the sequences in combination() can be generated by the queue
    vector<int> nums = {1, 2, 3, 4};
    vector<vector<int>> res = fullPermulation(move(nums)); // the full permutation of nums

    // this will be much easier than checkQueue1()
    for (auto & seq : res) {
        vector<int> queue_simu;
        unsigned i = 0, j = 0;
        while (true) {
            if (seq[i] != nums[j])
                queue_simu.push_back(nums[j]);
            else
                i += 1;

            while (!queue_simu.empty()) {
                if (queue_simu[0] == seq[i]) {
                    queue_simu.erase(queue_simu.begin());
                    i += 1;
                } else if (queue_simu[queue_simu.size() - 1] == seq[i]) {
                    queue_simu.pop_back();
```

```
                    i += 1;
                } else {
                    break;
                }
            }

            j += 1;
            if (i == seq.size() || j == nums.size()) break;
        }

        if (i == seq.size()) {
            cout << "seq: ";
            for (auto &i : seq) cout << i << " ";
            cout << "can be generated by the queue" << endl;
        } else {
            cout << "seq: ";
            for (auto &i : seq) cout << i << " ";
            cout << "can not be generated by the queue" << endl;
        }
    }
}
```

运行结果如下：

```
# checkQueue1() 输出受限的双端队列
eq: 1 2 3 4 can be generated by the queue
seq: 1 2 4 3 can be generated by the queue
seq: 1 3 2 4 can be generated by the queue
seq: 1 3 4 2 can be generated by the queue
seq: 1 4 3 2 can be generated by the queue
seq: 1 4 2 3 can be generated by the queue
seq: 2 1 3 4 can be generated by the queue
seq: 2 1 4 3 can be generated by the queue
seq: 2 3 1 4 can be generated by the queue
seq: 2 3 4 1 can be generated by the queue
seq: 2 4 3 1 can be generated by the queue
seq: 2 4 1 3 can be generated by the queue
seq: 3 2 1 4 can be generated by the queue
seq: 3 2 4 1 can be generated by the queue
seq: 3 1 2 4 can be generated by the queue
seq: 3 1 4 2 can be generated by the queue
seq: 3 4 1 2 can be generated by the queue
seq: 3 4 2 1 can be generated by the queue
seq: 4 2 3 1 can not be generated by the queue
seq: 4 2 1 3 can be generated by the queue
seq: 4 3 2 1 can be generated by the queue
seq: 4 3 1 2 can be generated by the queue
seq: 4 1 3 2 can not be generated by the queue
seq: 4 1 2 3 can be generated by the queue
```

```
# checkQueue2() 输入受限的双端队列
seq: 1 2 3 4 can be generated by the queue
seq: 1 2 4 3 can be generated by the queue
seq: 1 3 2 4 can be generated by the queue
seq: 1 3 4 2 can be generated by the queue
seq: 1 4 3 2 can be generated by the queue
seq: 1 4 2 3 can be generated by the queue
seq: 2 1 3 4 can be generated by the queue
seq: 2 1 4 3 can be generated by the queue
seq: 2 3 1 4 can be generated by the queue
seq: 2 3 4 1 can be generated by the queue
seq: 2 4 3 1 can be generated by the queue
seq: 2 4 1 3 can be generated by the queue
seq: 3 2 1 4 can be generated by the queue
seq: 3 2 4 1 can be generated by the queue
seq: 3 1 2 4 can be generated by the queue
seq: 3 1 4 2 can be generated by the queue
seq: 3 4 1 2 can be generated by the queue
seq: 3 4 2 1 can be generated by the queue
seq: 4 2 3 1 can not be generated by the queue
seq: 4 2 1 3 can not be generated by the queue
seq: 4 3 2 1 can be generated by the queue
seq: 4 3 1 2 can be generated by the queue
seq: 4 1 3 2 can be generated by the queue
seq: 4 1 2 3 can be generated by the queue
```

由此有答案：

(1) 能由输入受限的双端队列得到，但是不能由输出受限的双端队列得到的输出序列；

- 4 1 3 2

(2) 能由输出受限的双端队列得到，但是不能由输入受限的双端队列得到的输出序列；

- 4 2 1 3

(3) 既不能由输入受限的双端队列得到，也不能由输出受限的双端队列得到的输出序列；

- 4 2 3 1

4. 写出队列ADT的完整形式，即完善各个操作的说明。

```
ADT Queue {
    数据对象: D = {ai | ai ∈ ElemSet, i = 1, 2, …, n, n >= 0}
    数据关系: R1 = {<ai-1, ai> | ai-1, ai ∈ D, i = 2, 3, …, n}
    基本操作:
        InitQueue(&Q):
            操作结果：构造一个空队列Q
        DestroyQueue(&Q):
            初始条件：队列Q已存在
            操作结果：销毁队列Q
        ClearQueue(&Q):
            初始条件：队列Q已存在
            操作结果：将队列Q清空
        QueueEmpty(Q):
            初始条件：队列Q已存在
            操作结果：若队列Q为空队列，则返回true，否则返回false
        GetHead(Q, &e):
            初始条件：队列Q已存在且非空
            操作结果：用e返回队列Q的队头元素
        EnQueue(&Q, e):
            初始条件：队列Q已存在
            操作结果：插入元素e为队列Q的新的队尾元素
        DeQueue(&Q, &e):
            初始条件：队列Q已存在且非空
            操作结果：删除队列Q的队头元素，并用e返回其值
        QueueLength(Q):
            初始条件：队列Q已存在
            操作结果：返回队列Q的元素个数
        QueueTraverse(Q, visit()):
            初始条件：队列Q已存在
            操作结果：依次对队列Q的每个元素调用函数visit()。一旦visit()失败，则操作失败
} ADT Queue
```