栈和队列

栈

- 操作受限的线性表，只能在一端（栈顶）进行插入删除操作
- 先进后出，FILO

栈的实际应用

- 浏览器后退操作、app页面返回操作
- 各种软件的撤销操作
- 函数调用栈

栈的基本操作

- 入栈`push`
- 出栈`pop`
- 栈是否为空`isEmpty`

使用数组实现栈

使用C++面向对象实现

```cpp
#include <iostream>
using namespace std;

class Stack {
private:
    int* _data;
    int _size;
    const int MAXN = 1000;

public:
    Stack() {
        _data = new int[MAXN];
        _size = 0;
    }

    ~Stack() {
        delete[] _data;
    }

    void push(int x) {  //入栈
        _data[_size++] = x;
    }

    int& top() {        //访问栈顶元素
        if (empty()) throw exception("stack is empty");
        return _data[_size - 1];
```

```
    }

    void pop() {        //出栈
        if (empty()) throw exception("stack is empty");
        _size--;
    }

    bool empty() {
        return _size == 0;
    }

    int size() {
        return _size;
    }

    void clear() {
        _size = 0;
    }
};

int main() {
    Stack S;
    for (int i = 1; i <= 5; i++) {
        S.push(i);
    }
    printf("%d\n", S.size());
    S.top() = 100;  //由于top函数返回的是元素的引用，可这样更改栈顶元素的值（与考研无关，仅作了解）
    while (!S.empty()) {
        printf("%d ", S.top());
        S.pop();
    }
    printf("\n");
    return 0;
}
```

关于考研教材上实现栈的方式

上面的代码是用 `size` 变量表示当前栈所含元素的个数，此时判断栈空条件：`size == 0`

对于考研教材：

- 用栈顶指针top表示当前栈顶元素的下标
  - 判断栈空条件：`top == -1`
  - push、pop操作也要做相应改变
- pop操作不仅要弹出当前栈顶元素，还要将其值返回。即将上面代码中的top和pop操作合二为一。
- 由于C语言没有异常处理机制，每次操作是否成功通过函数返回值返回，原本需要返回的结果则通过指针或引用的方式。

```
#include <stdio.h>

typedef int STATUS;
#define ERROR -1
```

```
#define SUCCESS 0

#define MAX_SIZE 3
int data[MAX_SIZE];
int top = -1;   //指示当前栈顶元素的下标

bool empty() {
    return top == -1;
}

STATUS push(int x) {
    if (top >= MAX_SIZE - 1) return ERROR;
    data[++top] = x;
    return SUCCESS;
}

STATUS pop(int *e) {
    if (empty()) return ERROR;
    *e = data[top--];
    return SUCCESS;
}

int main() {
    for (int i = 1; i <= 5; i++) {
        push(i);
    }
    while (empty() == false) {
        int top_elem;
        pop(&top_elem);
        printf("%d ", top_elem);
    }
    printf("\n");
    return 0;
}
```

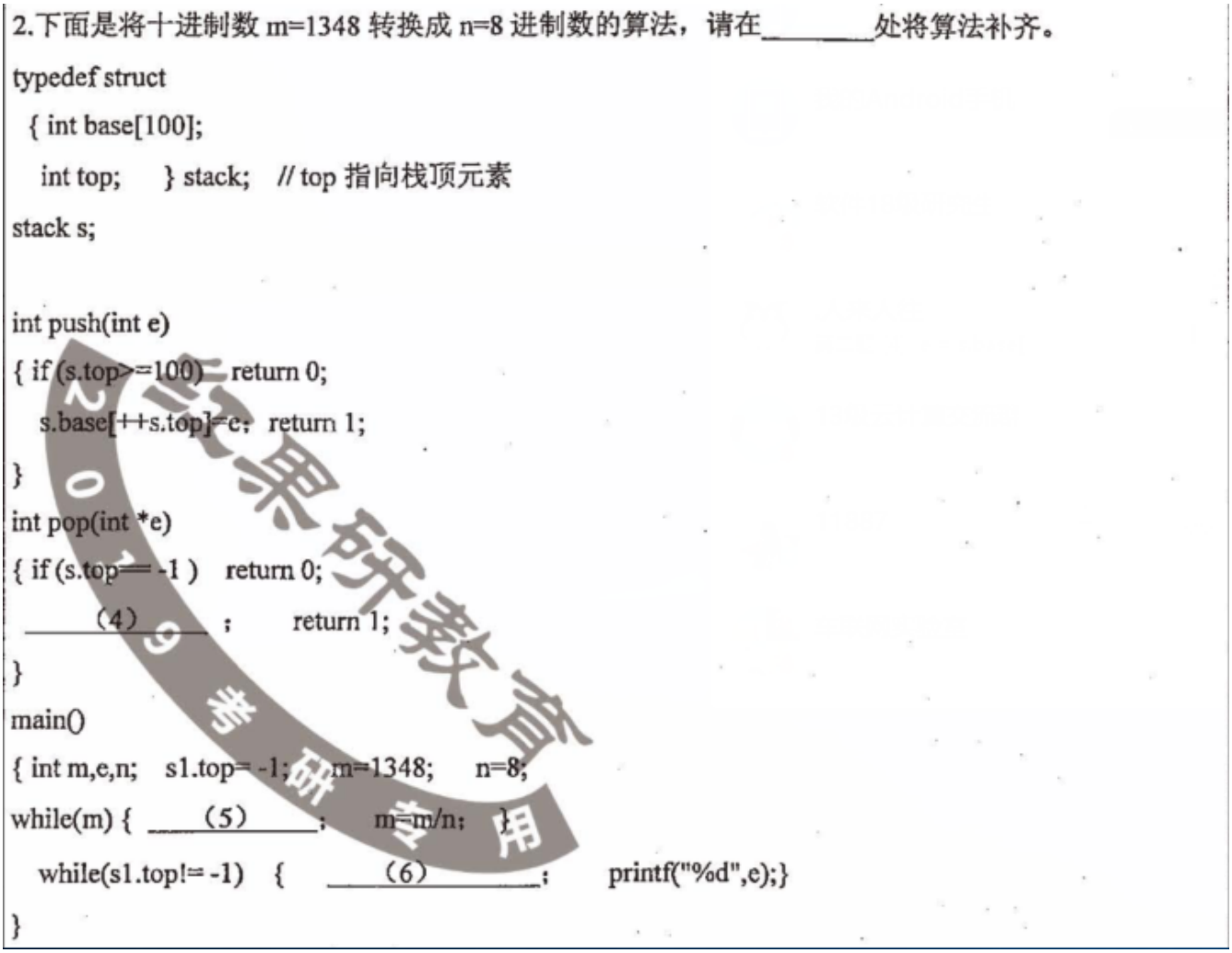
使用（单向）链表实现栈

在单向链表头结点出进行插入删除节点操作

```
#include <iostream>
#include <stack>
using namespace std;

struct Node {
    int data;
    Node* next;

    Node(int x = 0) { data = x; next = NULL; }
};

class Stack {
private:
    Node* head;
```

```
15
16  public:
17      Stack() {
18          head = new Node();
19      }
20
21      ~Stack() {
22          Node* r = head;
23          for (Node* p = head->next; p != NULL; p = p->next) {
24              delete r;
25              r = p;
26          }
27          delete r;
28          printf("stack所占内存已释放\n");
29      }
30
31      bool empty() {
32          return head->next == NULL;
33      }
34
35      void push(int x) {
36          Node* temp = new Node(x);
37          temp->next = head->next;
38          head->next = temp;
39      }
40
41      int& top() {
42          if (empty()) throw exception("stack is empty");
43          return head->next->data;
44      }
45
46      void pop() {
47          if (empty()) throw exception("stack is empty");
48          Node* temp = head->next;
49          head->next = temp->next;
50          delete temp;
51      }
52  };
53
54  int main() {
55      Stack S;
56      for (int i = 1; i <= 5; i++) {
57          S.push(i);
58      }
59      S.top() = 100;
60      while (S.empty() == false) {
61          printf("%d ", S.top());
62          S.pop();
63      }
64      printf("\n");
65      return 0;
66  }
```

栈的应用



2.下面是将十进制数 m=1348 转换成 n=8 进制数的算法，请在_____处将算法补齐。

```
typedef struct
  { int base[100];
    int top;     } stack;   // top 指向栈顶元素
stack s;

int push(int e)
{ if (s.top>=100)  return 0;
   s.base[++s.top]=e;  return 1;
}
int pop(int *e)
{ if (s.top== -1 )   return 0;
   ____(4)____ ;       return 1;
}
main()
{ int m,e,n;   s1.top= -1;  m=1348;     n=8;
while(m) { ____(5)____;     m=m/n;   }
   while(s1.top!= -1)   {    ____(6)____;      printf("%d",e);}
}
```

括号匹配

应用场景：源代码语法正确性检查

样例输入

```
3
()(((()())))
(()
()())(()())
```

样例输出

```
Yes
No
No
```

参考代码

使用C++ STL

```cpp
#include <iostream>
#include <string>
#include <stack>
using namespace std;

bool isMatch(const string &exp) {
    if (exp.length() % 2 != 0) return false;    //机智
    stack<char> S;
    for (int i = 0; i < exp.length(); i++) {
        char ch = exp[i];
        if (ch == '(') {
            S.push(ch);
        } else if (ch == ')') {
            if (S.empty()) return false;
            S.pop();
        }
    }
    return S.empty() == true;
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        string str;
        cin >> str;
        bool ans = isMatch(str);
        printf("%s\n", ans ? "Yes": "No");
    }
    return 0;
}
```

纯C语言

```c
#include <stdio.h>
#include <string.h>
#define MAX_SIZE 1024

int isMatch(char exp[], int n) {
    if (n % 2 != 0) return 0;
    char Stack[MAX_SIZE]; int size = 0; //栈的定义及初始化
    for (int i = 0; i < n; i++) {
        if (exp[i] == '(') {
            Stack[size++] = exp[i];     //push
        } else if (exp[i] == ')') {
            if (size == 0) return 0;    //empty() == true
            size--;                     //pop
        }
    }
    return size == 0;
```

```
}

int main() {
    int T;
    scanf("%d", &T);
    while (T--) {
        char str[MAX_SIZE];
        scanf("%s", str);
        int len = strlen(str);
        int ans = isMatch(str, len);
        printf("%s\n", ans ? "Yes" : "No");
    }
    return 0;
}
```

计算后缀表达式

中缀表达式：`(1 + 2) * 3 - 4 * 5 = -11`

后缀表达式：`1 2 + 3 * 4 5 * -`

前缀表达式：`- * + 1 2 3 * 4 5`

这里stack起到延迟缓冲的作用。

这里为了体现算法原理，简单起见假定输入的后缀表达式只含有`+ - *`3种操作符，且所有的操作数均为`0-9`的整数。

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

//这里为了体现算法原理，简单起见假定输入的后缀表达式只含有"+ - *"这3种操作符，且所有的操作数均为0-9的整数。
int calc(const string &exp) {
    stack<int> S;   //stack只存放操作数
    for (int i = 0; i < exp.length(); i++) {
        char ch = exp[i];
        if (ch == ' ') continue;    //跳过空白字符
        if ('0' <= ch && ch <= '9') {
            S.push(ch - '0');
        } else {
            int b = S.top(); S.pop();
            int a = S.top(); S.pop();
            int ans;
            if (ch == '+') {
                ans = a + b;
            } else if (ch == '-') {
                ans = a - b;
            } else if (ch == '*') {
                ans = a * b;
            }
            S.push(ans);
```

```
        }
    }
    return S.top();
}

int main() {
    string str;
    getline(cin, str);  //读入一行，可能包含空格
    int ans = calc(str);
    printf("%d\n", ans);
    return 0;
}
```

出栈序列判定问题（栈混洗）

定义

一个栈以`1，2，3，4`的顺序入栈，怎样才能得到`3，2，4，1`的出栈序列?

pu, pu, pu, pop, pop, pu, pop, pop

能得到`1，4，2，3`的出栈序列吗?

假如这个栈的最大容量为2，能得到`1，4，3，2`的出栈序列吗?

PAT A1051 Pop Sequence (25)

Given a stack which can keep M numbers at most. Push N numbers in the order of 1, 2, 3, ..., N and pop randomly. You are supposed to tell if a given sequence of numbers is a possible pop sequence of the stack. For example, if M is 5 and N is 7, we can obtain 1, 2, 3, 4, 5, 6, 7 from the stack, but not 3, 2, 1, 7, 5, 6, 4.

Input Specification:

Each input file contains one test case. For each case, the first line contains 3 numbers (all no more than 1000): M (the maximum capacity of the stack), N (the length of push sequence), and K (the number of pop sequences to be checked). Then K lines follow, each contains a pop sequence of N numbers. All the numbers in a line are separated by a space.

Output Specification:

For each pop sequence, print in one line "YES" if it is indeed a possible pop sequence of the stack, or "NO" if not.

Sample Input:

```
//M N K
//M: 栈的容量; N: push序列的长度; K: 让你判断的pop序列的个数
5 7 5
1 2 3 4 5 6 7
3 2 1 7 5 6 4
7 6 5 4 3 2 1
5 6 4 3 7 2 1
1 7 6 5 4 3 2
```

Sample Output:

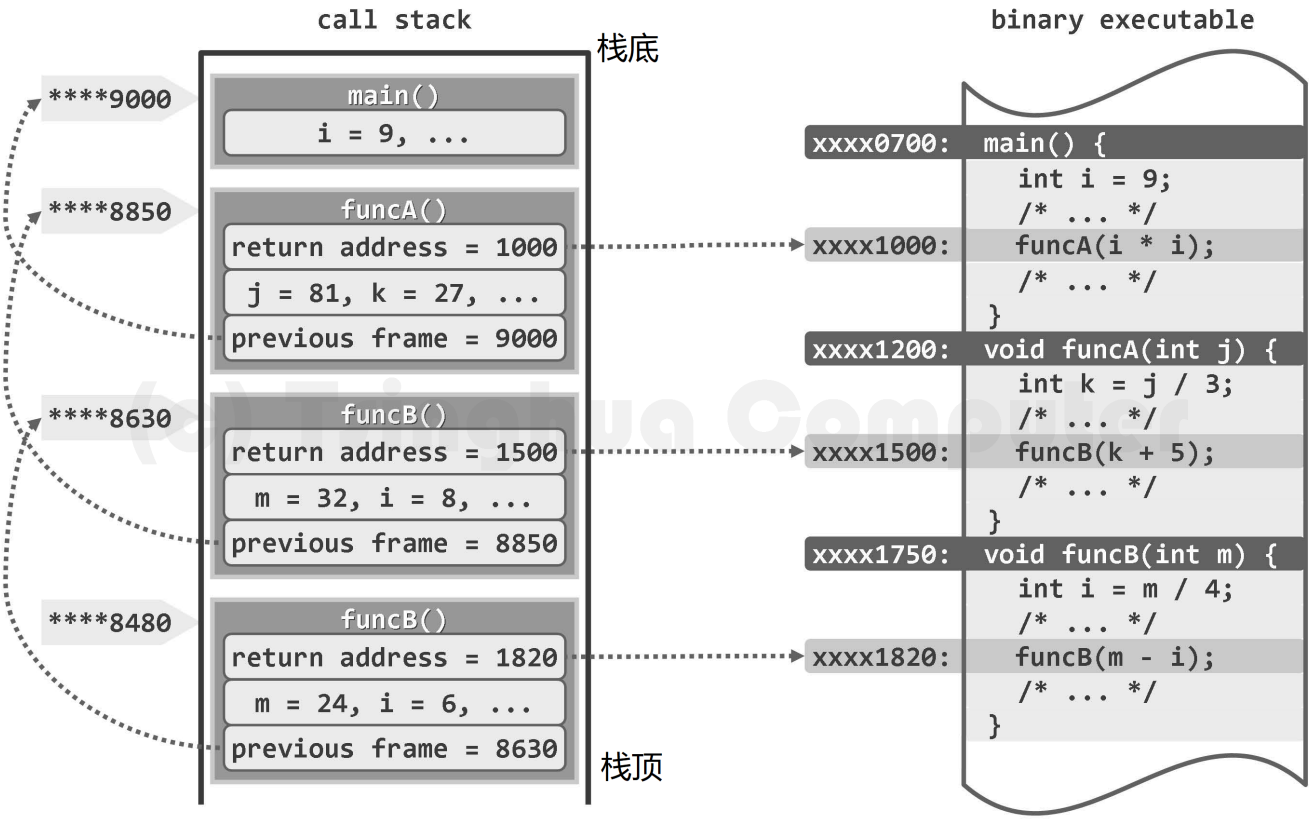
```
YES
NO
NO
YES
NO
```

参考代码:

```cpp
#include <iostream>
#include <stack>
using namespace std;
const int MAXN = 1010;

int num[MAXN];
int M, N, K;

bool judge() {
    stack<int> S;
    int p = 0;
    for (int i = 1; i <= N; i++) {
        S.push(i);
        if (S.size() > M) {
            return false;
        }
        while (!S.empty() && S.top() == num[p]) {
            S.pop();
            p++;
        }
    }
    return S.empty() == true;
}

int main() {
    scanf("%d %d %d", &M, &N, &K);
    for (int k = 0; k < K; k++) {
        for (int i = 0; i < N; i++) {
            scanf("%d", &num[i]);
        }
        bool ans = judge();
        printf("%s\n", ans ? "YES" : "NO");
    }
    return 0;
}
```

函数调用栈（了解）

函数调用栈实例：主函数`main()`调用`funcA()`，`funcA()`调用`funcB()`，`funcB()`再自我调用（递归）



函数调用栈的基本单位是帧（frame）。每次函数调用时，都会相应地创建一帧， 记录该函数实例在二进制程序中的返回地址（return address），以及局部变量、传入参数等， 并将该帧压入调用栈。若在该函数返回之前又发生新的调用，则同样地要将与新函数对应的一帧压入栈中，成为新的栈顶。函数一旦运行完毕，对应的帧随即弹出，运行控制权将被交还给该函 数的上层调用函数，并按照该帧中记录的返回地址确定在二进制程序中继续执行的位置。

在任一时刻，调用栈中的各帧，依次对应于那些尚未返回的调用实例，亦即当时的活跃函数实例（active function instance）。特别地，位于栈底的那帧必然对应于入口主函数main()， 若它从调用栈中弹出，则意味着整个程序的运行结束，此后控制权将交还给操作系统。

此外，调用栈中各帧还需存放其它内容。比如，因各帧规模不一，它们还需记录前一帧的起始地址，以保证其出栈之后前一帧能正确地恢复。

作为函数调用的特殊形式，递归也可借助上述调用栈得以实现。比如在上图中，对应于 `funcB()` 的自我调用，也会新压入一帧。可见，同一函数可能同时拥有多个实例，并在调用栈中 各自占有一帧。这些帧的结构完全相同，但其中同名的参数或变量，都是独立的副本。比如在 `funcB()` 的两个实例中，入口参数 `m` 和内部变量 `i` 各有一个副本。

试探回溯法（了解思想）

了解思想就够了，除非你要考《算法分析与设计》这门课.....

迷宫问题

在迷宫中找出一条从起点到终点的路径，不要求一定是最短路径。

```
//迷宫大小4*5，起点(0，0)，终点(3，4)，'#'表示墙
4 5 0 0 3 4
*****
##*##
****#
**#**
```

参考代码

这代码准备复试机试时再来研究.....

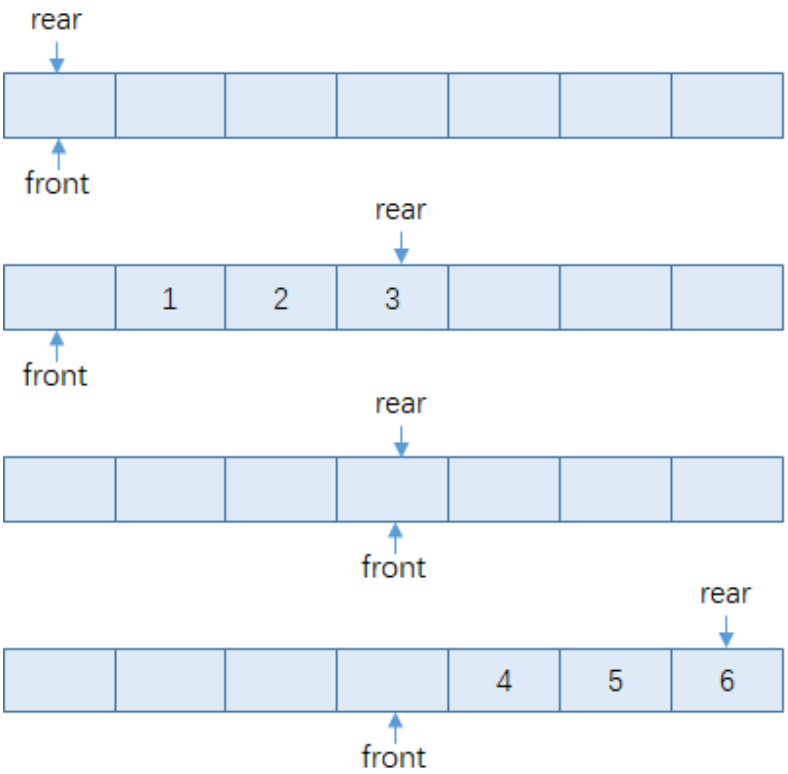
```cpp
#include <iostream>
#include <stack>
using namespace std;
const int MAXN = 105;

struct Point {
    int x, y;
    Point() {}
    Point(int a, int b) { x = a; y = b; }
    bool operator == (Point const &p) {
        return x == p.x && y == p.y;
    }
};

int N, M;
char G[MAXN][MAXN];
bool visited[MAXN][MAXN];   //记录每个点是否访问过

bool judge(int x, int y) {
    if (x < 0 || x >= N) return false;
    if (y < 0 || y >= M) return false;
    if (G[x][y] == '#') return false;
    if (visited[x][y]) return false;
    return true;
}

bool solve_backtracking(const Point &ST, const Point &ED, stack<Point> &path) {
    path.push(ST);
    while (path.empty() == false) {     //除非所有路径都试过了，最后退回到原点，否则都继续尝试走下去
        Point cur = path.top();         //获取当前位置
        if (cur == ED) return true;     //已经到终点了，直接return true
        int deltaXY[4][2] = { {1,0},{-1,0},{0,1},{0, -1} }; //上下左右四个方向的坐标增量
        for (int i = 0; i < 4; i++) {                   //四个方向依次尝试
            int _x = cur.x + deltaXY[i][0];
            int _y = cur.y + deltaXY[i][1];
            if (judge(_x, _y)) {                //判断新的坐标能否过去
                visited[_x][_y] = true;         //标记该坐标已访问
                path.push(Point(_x, _y));       //生成新坐标的结构体，并push
                goto nextLoop;      //中断当前for循环，直接进入下一轮while循环。这里不得已使用一次goto语句...
            }
        }
        path.pop();     //执行到这句说明4个方向都尝试完了，都不能走，只好退回去一步
    nextLoop:
        continue;
    }
    return false;
}
```

```
48
49  void printPath(stack<Point> &path) {
50      //由于栈的特性，这样输出的路径是倒序的
51      while (path.empty() == false) {
52          Point temp = path.top();
53          printf("(%d, %d)\n", temp.x, temp.y);
54          path.pop();
55      }
56  }
57
58  int main() {
59      scanf("%d %d", &N, &M);
60      Point ST, ED;
61      scanf("%d %d %d %d", &ST.x, &ST.y, &ED.x, &ED.y);
62      for (int i = 0; i < N; i++) {
63          scanf("%s", G[i]);  //迷宫的每一行当作字符串来读入。双重循环进行单个字符读入非常容易出错
64      }
65      stack<Point> path;
66      bool flag = solve_backtracking(ST, ED, path);
67      if (flag) {
68          printPath(path);
69      } else {
70          printf("No\n");
71      }
72      return 0;
73  }
```

队列

- 公平地对某种资源进行管理分配，如多个用户使用一台打印机
- 先进先出，FIFO
- 操作受限的线性表，只能在一端插入，另一端删除
- 后续章节 树的层次遍历、图的BFS广度优先搜索算法会用到队列
- 在队列头部`front`做删除操作，队列尾部`rear`做插入操作

顺序队列



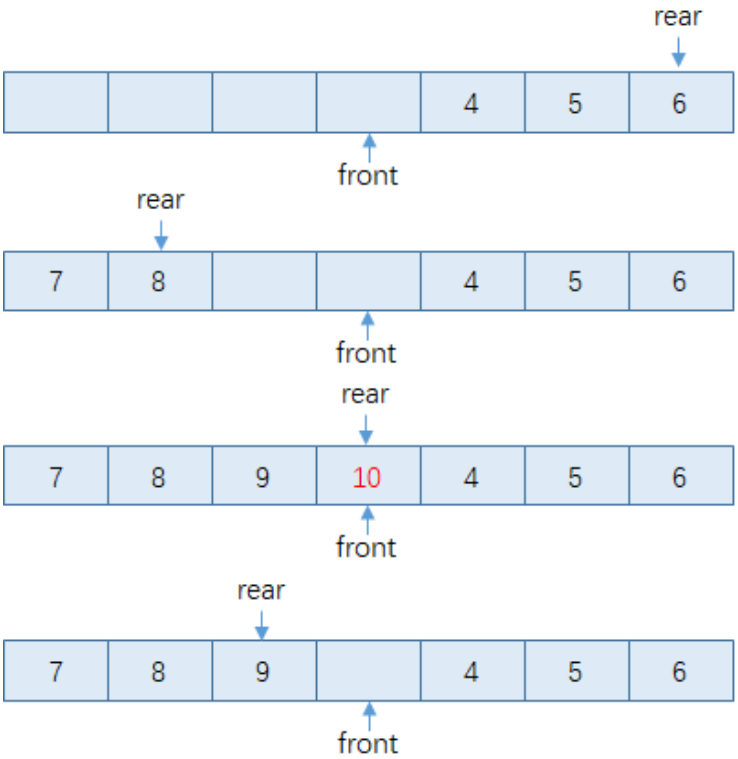
初始化：rear=front=0

push(1) push(2) push(3)

pop(1) pop(2) pop(3)
队列empty条件：`rear == front`

push(4) push(5) push(6)
push(7)？假溢出

循环队列



push(4) push(5) push(6)
push(7)？假溢出

循环队列，将数组想象成一个环
push(7) push(8)

push(9) push(10) 数组彻底装不下了
此时`rear==front`，根据该条件判断，
队列是full还是empty？

浪费一个数组空间，队列full条件：
`(rear+1) % MAX_SIZE == front`

- 队列empty条件：`return rear == front;`
- 队列full条件：`return (rear + 1) % MAX_SIZE == front`
- 入队push操作：

```
void push(int x) {
    rear = (rear + 1) % MAX_SIZE;
    data[rear] = x;
}
```

- 出队pop操作：

```
int pop() {
    front = (front + 1) % MAX_SIZE;
    return data[front];
}
```

- 获取队列大小

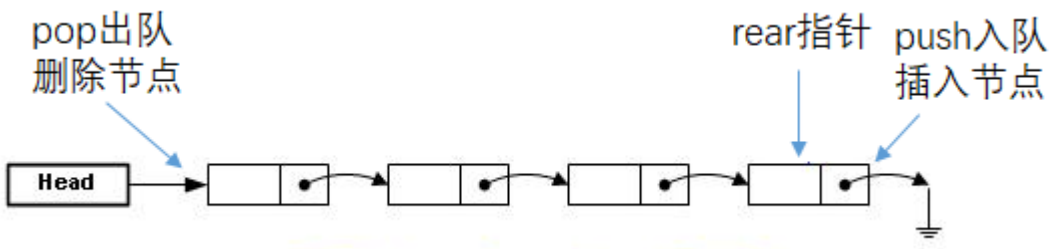
```
int size() {
    if (rear >= front) {    //注意这里必须取等号
        return rear - front;
    } else {
        return rear + MAX_SIZE - front;
    }
}
```

完整代码：

```
#include <iostream>
#include <queue>
using namespace std;

class Queue {
private:
    int* data;
    int front, rear;
    const int MAX_SIZE = 1000;

public:
    Queue() {
        data = new int[MAX_SIZE];
        rear = front = 0;
    }

    bool empty() {
        return rear == front;
    }

    bool full() {
        return (rear + 1) % MAX_SIZE == front;
    }

    void push(int x) {  //入队，enQueue
        if (full()) throw exception("队满了");
        rear = (rear + 1) % MAX_SIZE;
```

```
        data[rear] = x;
    }

    int pop() {          //出队, deQueue
        if (empty()) throw exception("队为空");
        front = (front + 1) % MAX_SIZE;
        return data[front];
    }

    int size() {
        if (rear >= front) {    //注意这里必须取等号
            return rear - front;
        } else {
            return rear + MAX_SIZE - front;
        }
    }
};

int main() {
    Queue Q;
    for (int i = 1; i <= 5; i++) {
        Q.push(i);
    }
    Q.pop();
    printf("size: %d\n", Q.size());
    Q.push(100);
    while (Q.empty() == false) {
        int x = Q.pop();
        printf("%d ", x);
    }
    printf("\n");
    return 0;
}
```

链表队列




```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;

    Node(int x = 0) { data = x; next = NULL; }
```

```cpp
};

class Queue {
private:
    Node* head; //链表的头节点
    Node* rear; //链表的尾指针

public:
    Queue() {
        head = new Node();  //创建头节点
        rear = head;        //尾指针指向头节点
    }

    bool empty() {
        return head->next == NULL;
    }

    void push(int x) {  //入队，在链表尾指针处插入节点
        rear->next = new Node(x);
        rear = rear->next;
    }

    int pop() {         //出队，在链表头结点处删除节点
        if (empty()) throw exception("队列为空");
        int ret = head->next->data;
        Node* t = head->next;
        head->next = t->next;
        delete t;
        return ret;
    }
};

int main() {
    Queue Q;
    for (int i = 1; i <= 5; i++) {
        Q.push(i);
    }
    Q.pop();
    Q.push(100);
    while (Q.empty() == false) {
        int x = Q.pop();
        printf("%d ", x);
    }
    printf("\n");
    return 0;
}
```